**Profiling Software Using Intel Advisor**

**The content of this guide is written by adopting from the following sources:**

1. <https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top.html>
2. <https://hpc.llnl.gov/software/development-environment-software/intel-advisor>

**Chapter1: Work flow**

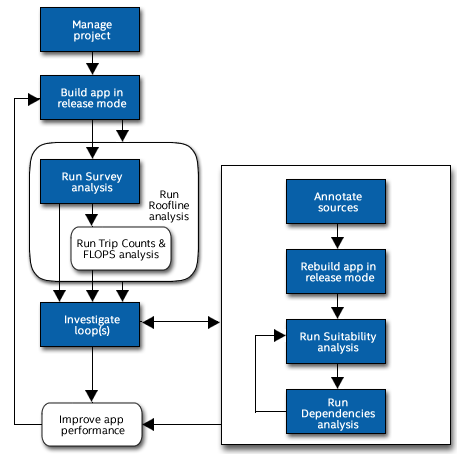
Intel Advisor is a prototyping tool that allows users to analyze their code and determine the costs and benefits of adding various threading models on Intel processors. It works on code written in C, C++, and Fortran, and can model parallelism using OpenMP, Intel Thread Building Blocks, and Intel Cilk Plus. Advisor can also provide guidance to help codes get better vectorization, which is becoming increasingly important with wide vector units in modern processors.

It is composed of two tools:

1. Threading Advisor is a fast-track threading design and prototyping tool that lets you analyze, design, tune, and check threading design options without disrupting your normal development.
2. Vectorization Advisor is a vectorization optimization tool that lets you identify high-impact, under-optimized loops, what is blocking vectorization, and where it is safe to force vectorization. It also provides code-specific how-can-I-fix-this-issue? recommendations.

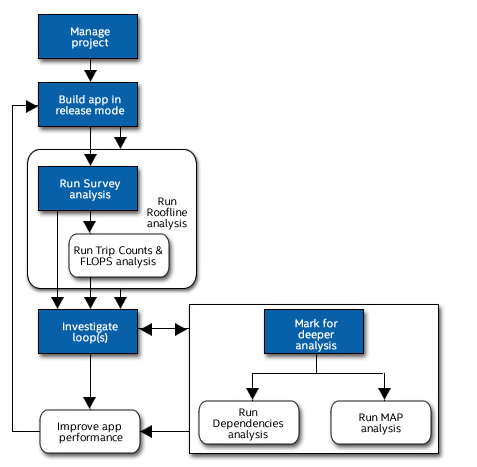
Advisor includes both a graphical user interface (GUI) and a command line (CL) interface that can be accessed with the *advixe-gui* and *advixe-cl* commands, respectively. When running the GUI, begin by creating a new project, entering the executable path and arguments, and setting other options. Once the Advisor project is created, run through the threading and/or vectorization workflows. In this tutorial, we focus on command line that can be available on DevCloud.

The **Threading Workflow** consists of these steps:



1. **Survey Report** - Shows the loops and functions where your application spends the most time. Use this information to discover candidates for parallelization with threads.
2. **Trip Counts analysis** - Shows the minimum, maximum, and median number of times a loop body will execute, as well as the number of times a loop is invoked. Use this information to make better decisions about your threading strategy for particular loops.
3. **Roofline chart** - Helps visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
4. **Annotations** - Insert to mark places in your application that are good candidates for later replacement with parallel framework code that enables threading parallel execution. Annotations are subroutine calls or macros (depending on the programming language) that can be processed by your current compiler but do not change the computations of your application.
5. **Suitability Report** - Predicts the maximum speed-up of your application based on the inserted annotations and a variety of what-if modeling parameters with which you can experiment. Use this information to choose the best candidates for parallelization with threads.
6. **Dependencies Report** - Predicts parallel data sharing problems based on the inserted annotations. Use this information to fix the data sharing problems if the predicted maximum speed-up benefit justifies the effort.

The **Vectorization Workflow** consists of these steps:



1. **Survey Report**- Offers integrated compiler report data and performance data all in one place. Use it to help identify: Where vectorization, or parallelization with threads, will pay off the most
2. **Trip Counts and FLOP analysis** - Dynamically identifies the number of times loops are invoked and execute (sometimes called call count/loop count and iteration count respectively); and measures the number of floating-point and integer operations, and memory traffic. Use to make better decisions about your vectorization strategy for particular loops, as well as optimize already-parallel loops.
3. **Roofline chart** - Helps visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.
4. **Dependencies Report** - For safety purposes, the compiler is often conservative when assuming data dependencies. Use a Dependencies-focused Refinement Report to check for real data dependencies in loops the compiler did not vectorize because of assumed dependencies. If real dependencies are detected, the analysis can provide additional details to help resolve the dependencies. Your objective: Identify and better characterize real data dependencies that could make forced vectorization unsafe.
5. **Memory Access Patterns (MAP) Report**- Use a MAP-focused Refinement Report to check for various memory issues, such as non-contiguous memory accesses and unit stride vs. non-unit stride accesses. Your objective: Eliminate issues that could lead to significant vector code execution slowdown or block automatic vectorization by the compiler.

**Chapter 2: Survey, Trip Counts, FLOPS, and Roofline Analyses**

**Survey analysis** - Identifies:

* Where vectorization, or parallelization with threads, will pay off the most
* If vectorized loops are providing benefit, and if not, why not
* Un-vectorized loops and why they are not vectorized
* Performance problems in general

**Trip Counts analysis** - Dynamically identifies the number of times loops and functions are invoked and executed (also called call count/loop count and iteration count respectively). Use Trip Counts data to:

* Detect loops with too-small trip counts and trip counts that are not a multiple of vector length.
* Analyze parallelism granularity more deeply.

**FLOP analysis** - Dynamically measures floating-point and integer operations, and memory traffic. Use the FLOP analysis to generate application memory usage and performance values that help you make better decisions about your vectorization strategy.

**Roofline analysis** - Helps you visualize actual performance against hardware-imposed performance ceilings, as well as determine the main limiting factor (memory bandwidth or compute capacity), thereby providing an ideal roadmap of potential optimization steps.

Use the Roofline chart to answer the following questions:

* What is the maximum achievable performance with your current hardware resources?
* Does your application work optimally on current hardware resources?
* If not, what are the best candidates for optimization?
* Is memory bandwidth or compute capacity limiting performance for each optimization candidate?

**Chapter 3: Refinement Reports**

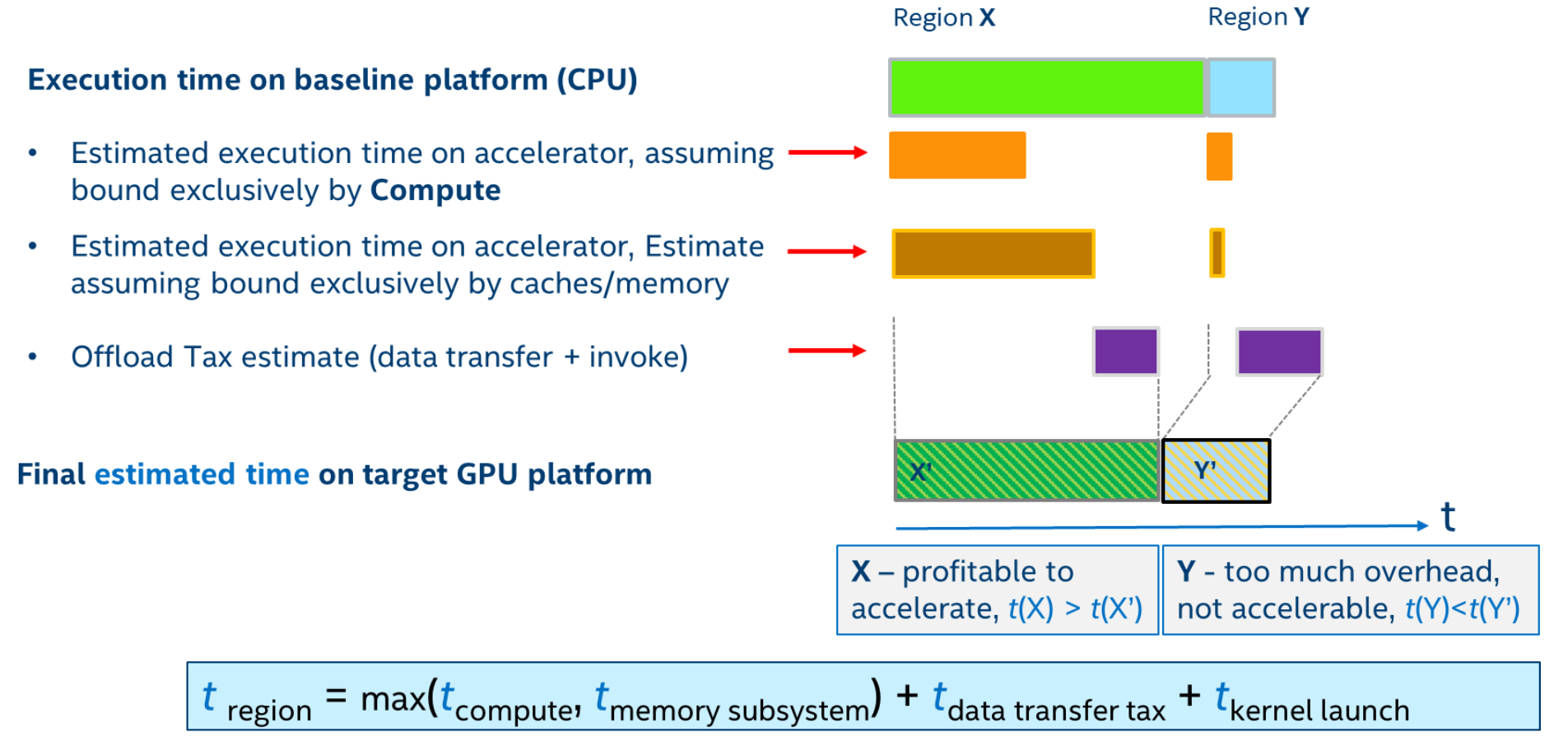
Intel Advisor offers two refinement analyses:

* **Dependencies analysis** (optional) - For safety purposes, the compiler is often conservative when assuming data dependencies. Run a Dependencies analysis to check for real data dependencies in loops the compiler did not vectorize because of assumed dependencies. If real dependencies are detected, the analysis can provide additional details to help resolve the dependencies. Your objective: Identify and better characterize real data dependencies that could make forced vectorization unsafe.
* **Memory Access Patterns** (MAP) analysis (optional) - Run a MAP analysis to check for various memory issues, such as non-contiguous memory accesses and unit stride vs. non-unit stride accesses. Your objective: Eliminate issues that could lead to significant vector code execution slowdown or block automatic vectorization by the compiler.

**Chapter 4: offload advisor**

Intel® Advisor Beta includes Offload Advisor, a tool that allows you to collect performance predictor data in addition to the profiling capabilities of Intel Advisor. Use Offload Advisor to determine what code can be offloaded to a target device (for example, to a GPU), accelerating the performance of your CPU-based application.

Offload Advisor provides metrics and performance data such as projected speedup, a call tree showing offloaded and accelerated regions, and more. It takes into account not only compute and memory limitations, but the time required to transfer data if the code is offloaded. The Offload Advisor produces upper-bound speedup estimates using a bounds and bottlenecks performance model. It takes measured x86 CPU metrics and application characteristics as an input and applies an analytical model to estimate execution time and characteristics on a target GPU.



In the Offload Advisor workflow, you must (1) run performance profiling to collect data about your application, and (2) run performance modeling and generate your report results. These steps are generally performed using the collect.py and analyze.py scripts, respectively.

Run analyze.py to predict your application performance on a target device (for example, on a GPU) and generate Intel® Advisor Beta output results:

*advixe-python <APM>/analyze.py <project-dir> [--options]*

where <APM> is the Offload Advisor environment variable that points to script directory. Replace it with $APM on Linux\* OS or %APM% on Windows\* OS.

By default, Intel® Advisor Beta collections and analysis are optimized to model performance on an integrated Intel® Processor Graphics Gen11. You can switch to integrated Intel® Processor Graphics Gen9 configuration by providing an alternate hardware definition model .toml file as an input to the analyze.py and collect.py / run\_oa.py. For example, for data collection with collect.py and performance modeling with analyze.py for the Intel® Processor Graphics Gen9 architecture:

*advixe-python <APM>/collect.py <project-dir> –-config gen9 [--options] -- <target> [target-options]*

*advixe-python <APM>/analyze.py <project-dir> [--options]*

**Chapter 5: How to use**

You can use the Intel Advisor command line interface, advixe-cl, to run analyses and reports. This makes it possible to automate many tasks as well as analyze an application running on remote hosts. You can then view results using the Intel Advisor GUI or command line reports.

