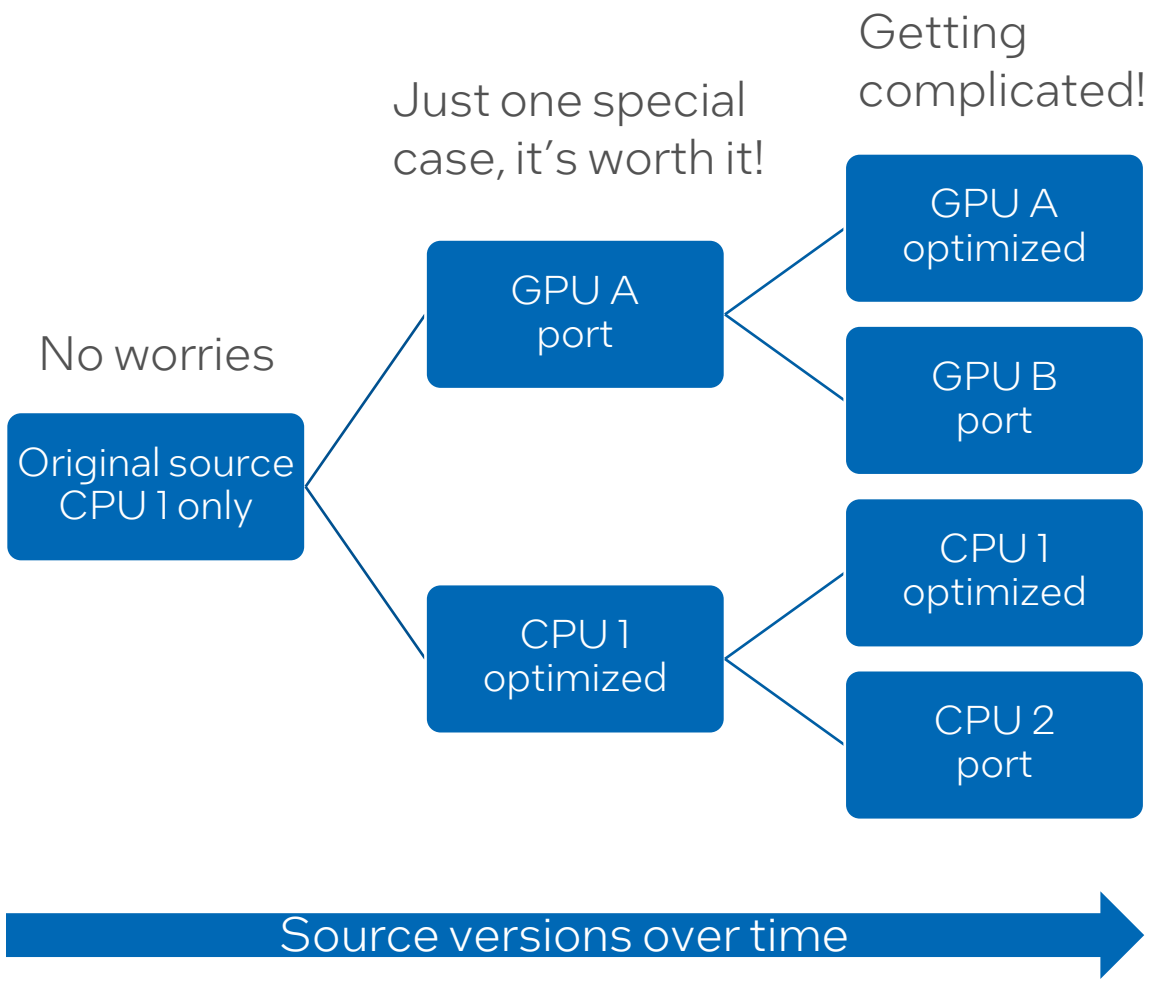
Performance Portability & Productivity in the Porting of Weather Codes to Heterogeneous Platforms

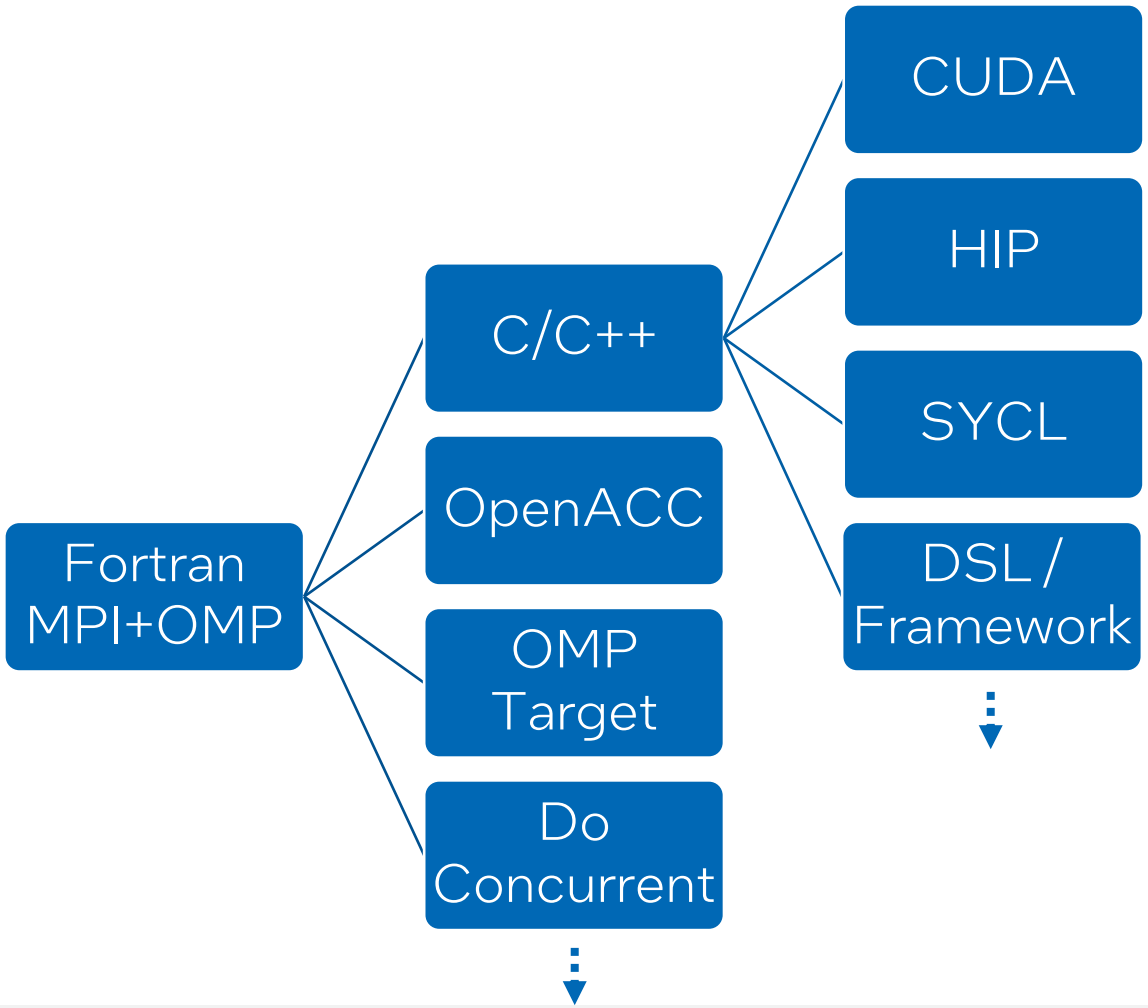
Camilo Moreno
(camilo.a.moreno@intel.com)

Motivation



- Classically it was enough to optimize for one architecture, generally CPU
- New systems can combine architectures including CPU, GPU, AI, FPGA
- Ambitious goals for model detail, collaboration, AI integration, etc. demand support for new architectures, with sufficient performance
- But the complexity and maintenance burden of the resulting codebase must be controlled!

Example: starting from FORTRAN



- Multiple ways to serve a given set of platforms
- Each implementation has:
  - Porting cost
  - Supported platforms
  - Optimization opportunities
  - A relationship to existing code

Kernel Optimization

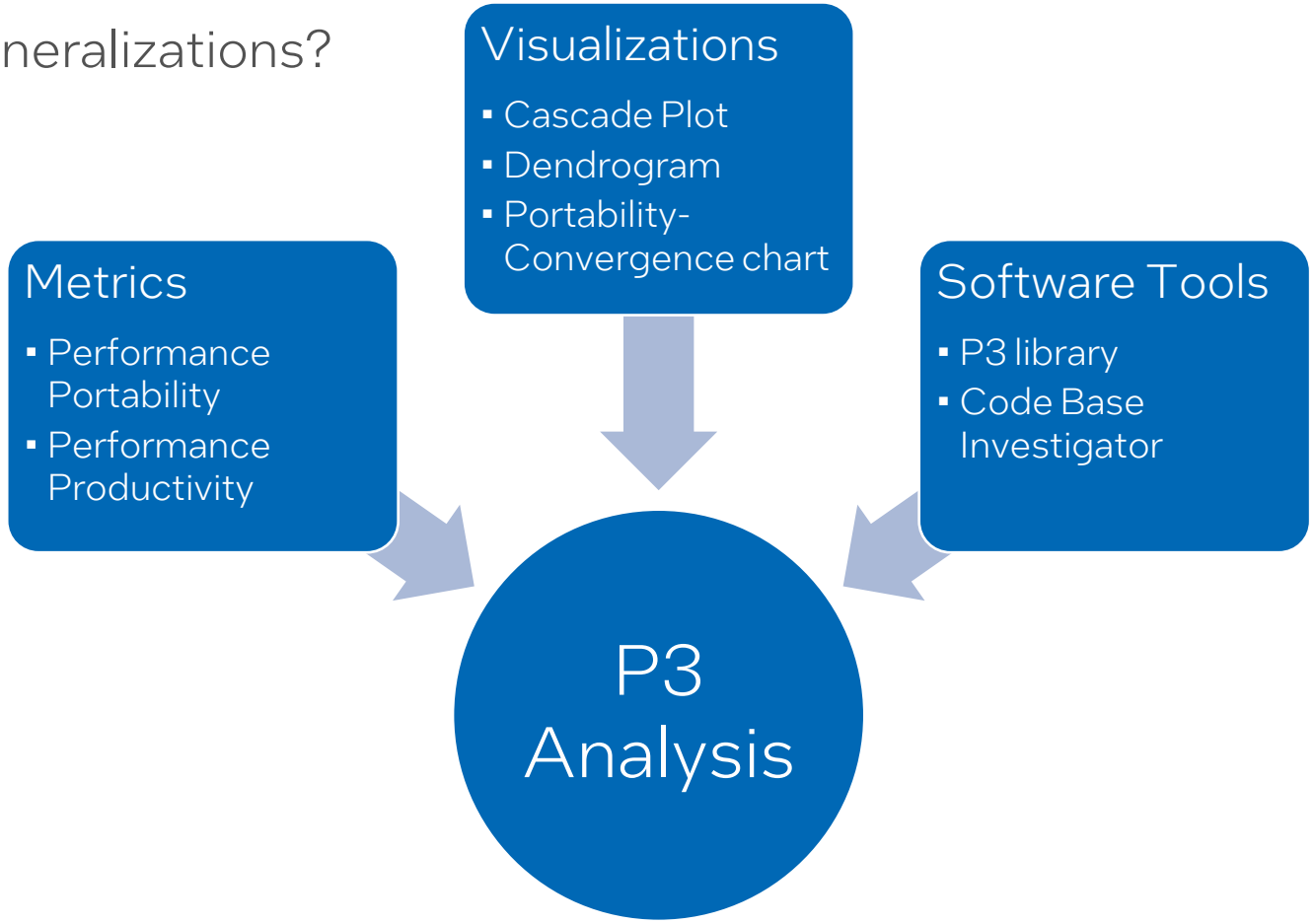
- Within a given language or framework, we also make optimization choices for different targets, with similar trade-offs
  - AoS vs SoA data layout
  - Loop nesting
  - cache blocking
  - etc

Performance Portability & Productivity

- It is important to support *and perform well* on multiple platforms
- Most would prefer to *not* write a custom version for each platform
- But having *the option* to optimize for a specific platform can be valuable
- Very important to avoid creating code maintenance problems

Performance Portability & Productivity

- Can we move past vague generalizations?



Preliminary: Performance Efficiency

- How well does a specific app perform on a specific platform?
- In the context of the following metrics, two suggested choices, depending on our goal:
  - Architectural Efficiency: Achieved performance relative to peak *theoretical* limit of the given hardware
    - Classic example: Roofline plots
  - Application Efficiency: Achieved performance relative to the *best observed* for the given application on the given platform
    - Good when it's not clear which HW limits practically apply
- See "Implications of a metric for performance portability" for more info

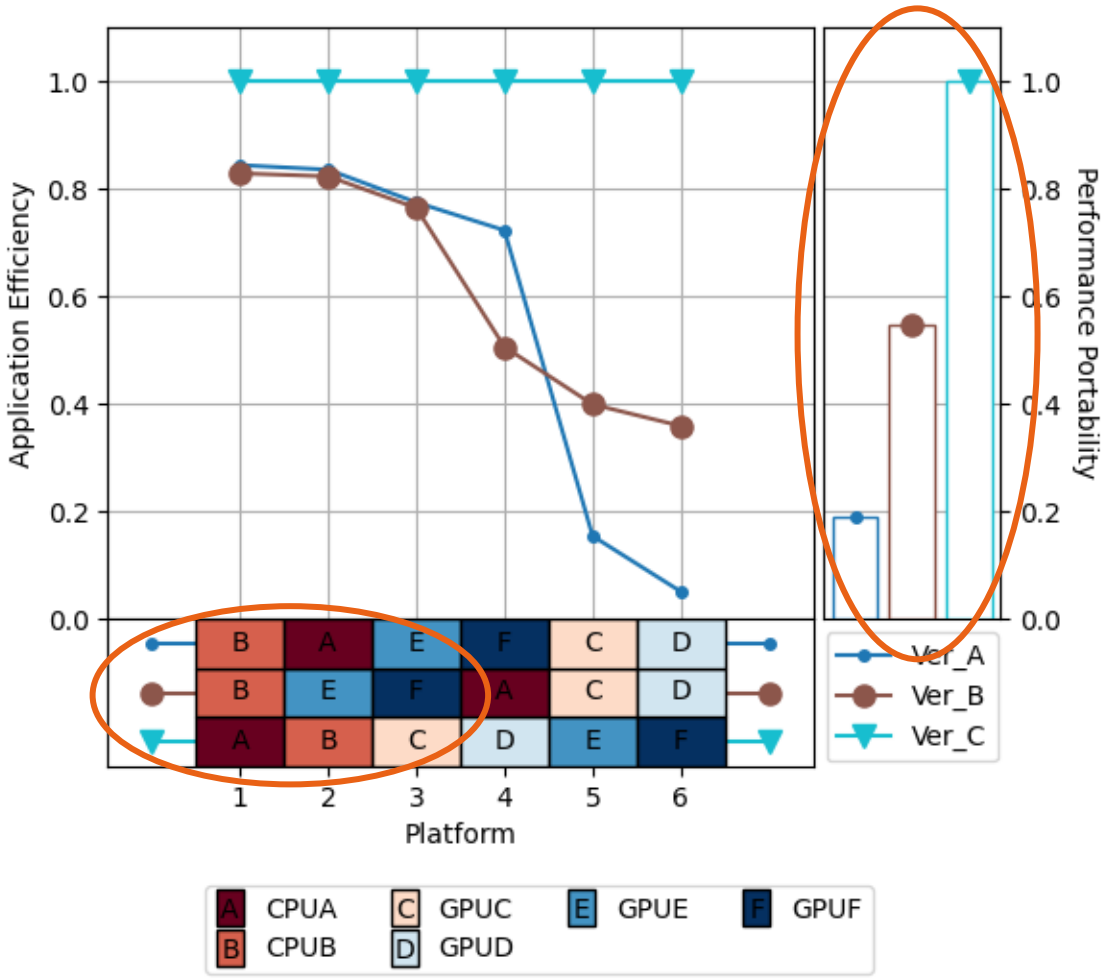
Performance Portability

$$\Phi(a, p, H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

From "Implications of a metric for performance portability"

- Can we measure how *performance portable* a given application is across a chosen set of platforms?
- The **Harmonic Mean** of the performance efficiencies of app ***a*** solving problem ***p*** across target platform set *H*.
  - your choice of performance efficiency metric
- Useful properties:
  - If any platform is unsupported, metric is 0
  - Proportional to sum of efficiencies; increases when any of them increase
  - Fixing ***worst*** platform is the fastest way to improve

PP Visualization: Cascade Chart



- Visualize **performance portability** of multiple apps relative to a set of target architectures, and how well it each does across the target architectures
- In this specific case, three versions of a SYCL kernel across 6 HW platforms (2 CPU, 4 GPU), using *Application Efficiency*
- Generated using the **P3** Analysis Library from performance measurements of all combinations
- See "Interpreting and visualizing performance portability metrics" for more info

Code Divergence

$$\mathrm{CD}(a,p,H) = \binom{|H|}{2}^{-1} \sum_{\{i,j\} \in H \times H} d_{i,j}(a,p)$$

$$d_{i,j}(a,p) = 1 - \frac{|c_i(a,p) \cap c_j(a,p)|}{|c_i(a,p) \cup c_j(a,p)|}$$

From “Navigating Performance, Portability and Productivity”

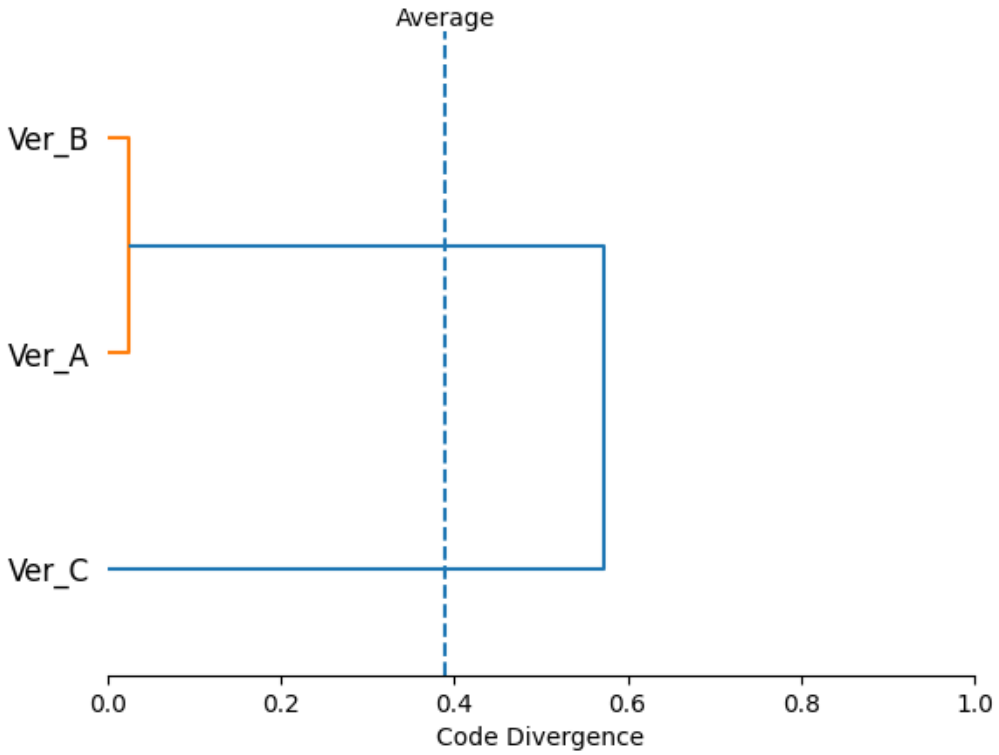
- Assume we **maintain** multiple apps which solve the same problem across different platforms: How different are the sources from each other?
- **Code Divergence** is the average of the distances between each pair of codes. In this case using *Jaccard* distance
  - **Jaccard distance:** dissimilarity metric between the two sets of source lines, range [0,1]
- Useful to approximate future cost of maintenance & of adding one more platform
- *Different* from one-time development cost of given apps

Code Divergence

- As an example, we can compare OpenACC versions of a cloud microphysics code (ECMWF's Cloudsc) to the Fortran baseline
- Code Divergence as in previous slide, based on Jaccard distance, including both host / driver side and kernel side code
- Note: Sometimes two versions of a program have similar sources, but kept as separate files, for clarity. This may have consequences for divergence metric as currently defined
  - Though in a way it reflects practical issues as well

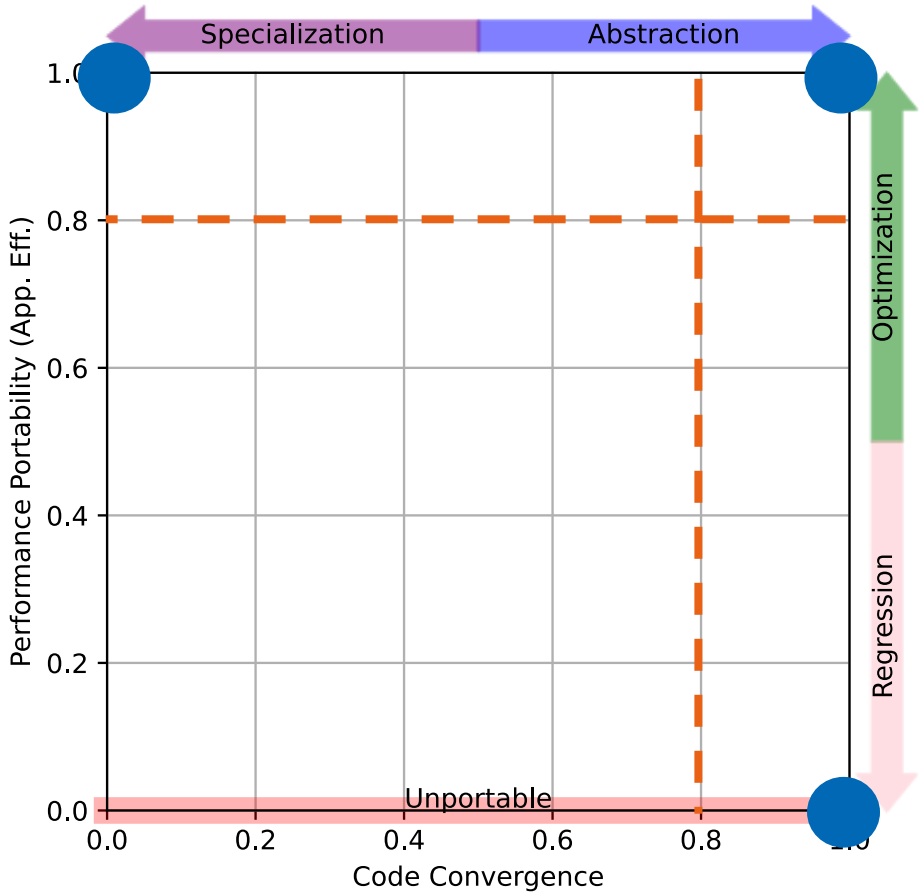
OpenACC implementation	Code Divergence vs baseline
Basic	0.014 (1.4%)
"hoist" version	0.61 (61%)
"scc" version	0.065 (6.5%)

Code Divergence Visualization: Dendrogram



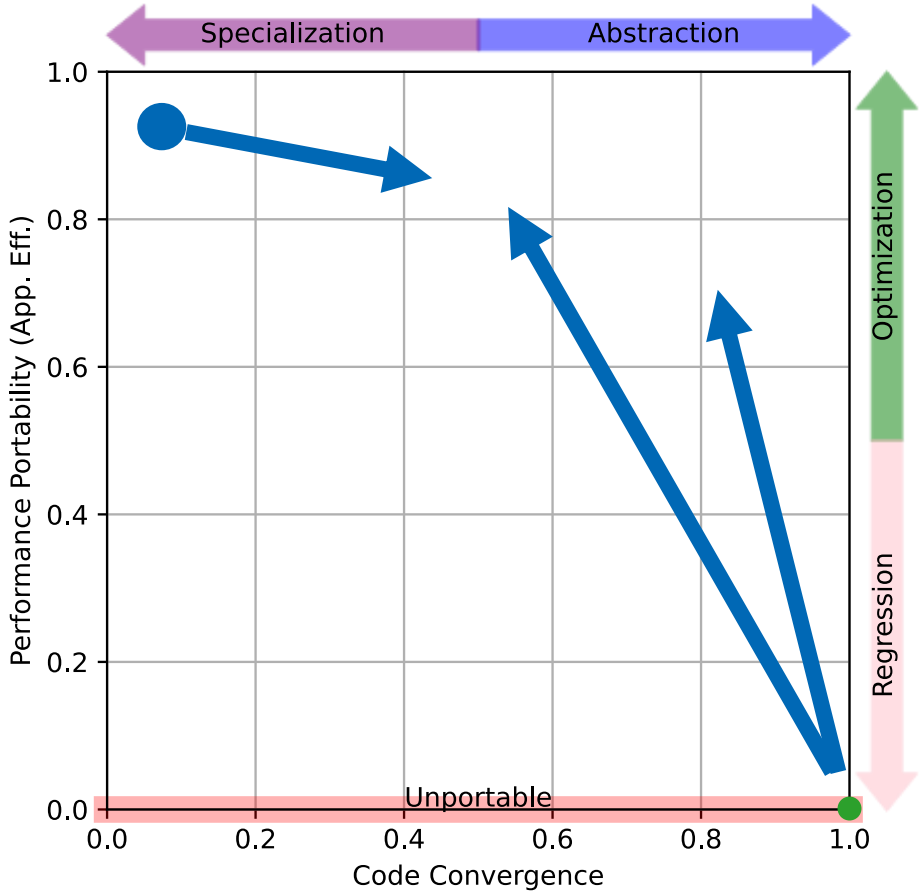
- A *dendrogram* illustrates Code Divergence relationships among a set of applications
- Apps connected more closely (left) are more similar, less divergent
- The **Code Base Investigator** tool (on Github) was used for the creation of this chart, as well as computing the Code Divergence metric.
  - Counts SLOC per version, guided by user info on source files and compile time variables

Putting it Together



- This chart combines:
  - **Code Convergence** on X-axis
  - **Performance Portability** on Y-axis
- Graphical representation of the P3 status of a given codebase targeting a given set of platforms
- Our position can give us clues about what to do next
- Bottom Right: one version, not portable
- Target-specific optimizations aim high, but incur large code divergence (top left)
- Ultimate goal: top right, single source, peak performance across the board
- It may make sense to aim for practical thresholds, e.g., 80%

General Patterns



- Every change to the codebase can be seen as a movement in this space, e.g.,
- DSLs and high-level abstractions can enable good performance on multiple platforms with relatively little code divergence; but it may be harder to optimize each platform to the limit
- Lower-level abstractions like SYCL incur more code divergence than a DSL, but retain high optimization potential
- If we start from a highly specialized codebase, we can improve code divergence by introducing common abstractions, though we may struggle to maintain the same performance as we do so

Further thoughts

- Abstractions are good for Perf. Portability
  - Can exist at several levels
  - Especially good if work on implementation layers is shared/re-used
  - Also good if possibilities remain open for further optimization
- Fortran makes things a bit tricky
  - Fewer DSL & Framework choices vs C/C++ or Python; significant incentive to migrate
  - AI integration
  - Interfaces & Translation layers can help, though complexity suffers
- Automated translation
  - One-time translation (E.g. Intel's SYCLomatic CUDA → SYCL tool) may help with porting cost, But doesn't really address Code Divergence
  - Effect of compile-time translation may depend on specifics: how much it constrains design decisions, how difficult to get a well-optimized result
- Integration: A weather prediction suite has many pieces with different characteristics
  - Complexity of making the right choice for an integrated result
  - "same approach across all pieces" vs. "the right approach for each piece"?
  - All at once, or incrementally?

References

- Harrell, Kitson, et al. “Effective Performance Portability”
- Pennycook, Sewall, et al. “Navigating Performance, Portability, & Productivity”
- Sewall, Pennycook, et al. “Interpreting and Visualizing Performance Portability Metrics”
- Pennycook, Sewall, Lee, “Implications of a metric for Performance Portability”

- Code Base Investigator: https://github.com/intel/code-base-investigator
- P3 Analysis Library: https://github.com/intel/p3-analysis-library