DOI:10.1145/3150215

To view the accompanying paper, visit doi.acm.org/10.1145/3150211

Technical Perspective Can High Performance Be Portable?

By Manuel Chakravarty

THE DEVELOPMENT OF high-performance software has always suffered from a tension between achieving high performance on the one hand and portability and simplicity on the other hand. By specializing an algorithm for optimal performance, considering the memory hierarchy and other architectural particulars, we introduce architecture-specific detail. This obscures algorithmic structure and conflates the general with the specific, compromising simplicity and clarity. It also hurts portability to all but very similar architectures—simple changes, such as different cache sizes, can have substantial performance implications. Moreover, distinctly different architectures, such as CPUs versus GPUs versus DSPs, often require fundamentally different optimization strategies. As a result, high-performance code is difficult to write, debug, maintain, and port.

Numerous research efforts were aimed at addressing this issue by applying automatic code transformations and other forms of compiler optimizations. Ultimately, we would prefer the software developer simply code the algorithm and leave it to the machine to specialize that algorithm to any particular architecture for efficient execution. In this ideal world, portability is a matter of retargeting a compiler's optimization engine. Unfortunately, architectural complexity and the lack of architectural models that are simultaneously sufficiently detailed and tractable have prevented us from realizing this vision.

The following work by Ragan-Kelley et al. on the image processing language Halide explores a substantially different approach to architecture-specific code optimization. By shifting our perspective on how to express architectural constraints and how to generate high-performance code, it achieves the impressive feat of simplifying high-performance code, while at the same time improving both portability and performance beyond that of traditional complex and non-portable approaches. This threefold success is indicative of a qualitative breakthrough, a definitive step forward in the state of the art.

The following work on the image processing language Halide explores a substantially different approach to architecture-specific code optimization.

Key to the authors' approach is the strict separation of the algorithmic code from an explicit specification of how to optimize that code for a given architecture. This specification, which they call the execution schedule, determines evaluation order, the amount of inlining, storage of intermediate data structures, and the choice between caching versus recomputation. With all the details of execution separated out, the remaining algorithmic code is purely functional.

This idea of separating the algorithmic code from the details of how to specialize that code for a specific architecture has been put forward before—for example, in the work on algorithmic skeletons. However, previous work lacks the clarity and simplicity of Halide and has failed to provide practical benefits at the scale of Halide. Its extraordinary success is due to the choice of the architectural specifics included in the schedule together with the specific optimization and code-generation technology informed by the schedule. Ultimately, the work on Halide combines conceptual insight with the engineering prowess required to turn this insight into a distinct improvement for realistic applications. In this context, it is important to recognize that Halide started with a tight focus on a specific application area, namely image processing. While the concepts underlying Halide are more general, the tight domain focus has led to convincing applications—for example, Halide is used in Google's Pixel phone, Google Photos, and YouTube.

Looking ahead, the core question is to what extent Halide's approach can be generalized to applications outside of image processing and, more broadly, how Halide's programming model can be generalized. At its core, image processing is a subdomain of array programming. This provides a natural progression for Halide's approach to grow into other domains. First steps in this direction have been undertaken by successfully applying Halide, as is, to algorithms from linear algebra and machine learning. More challenging will be to extend the expressiveness of Halide to cover a broader range of computational forms than currently supported by its algorithmic language, while retaining the clear separation of algorithmic code from the execution schedule.

In addition to generalizing the application domain, a second question is the complexity of developing execution schedules. The authors note that, even in the current context, complex schedules require expert knowledge. While this is hardly surprising, as conventional high-performance optimization requires experts as well, machine support is a tantalizing option. The authors have begun to study this, but many questions remain.

Manuel Chakravarty is a functional programming evangelist at Tweag I/O, Paris, France.