

Richard A. Lethin
Directing Engineer
Reservoir Labs, Inc.

Position (5/4/07)

The problems of parallelization to multiple cores, achieving high efficiencies, entails automatic optimization problems that are at the edge of what is tractable with today's compiler technology. Therefore, the question of how that optimization is partitioned over static and dynamic portions of the tool chain is germane.

Working from the perspective of the experience of developing automatic optimization technologies based on novel high level program representations, provides some insights. This perspective emerges from our experience in the Morphware forum contributing to the development of Morphware virtual machine APIs, particularly the streaming virtual machine. Also, we developed a High-Level Compiler, R-Stream, that can take C programs as input and optimize/partition them among control processors and streaming engines as in the Cell chip, managing distributed local memories, emitting DMAs, and so forth. The resulting program interfaces to the hardware through the Streaming Virtual Machine interface, which is an API for expressing streaming task invocation, communication operations, memory management, and synchronization. We can present some significant lessons learned (sometimes painfully) in developing this flow.

As a follow-on idea, while we can make great strides in representing the output of the mapper as traditional operations of tasks, loops, DMA, and synchronizations, we may be able to do more interesting things. In particular, new parallel optimization technologies are amenable to expressing execution models that are parametric at a very high level. We can and indicate computation as geometric subsets of iteration spaces over geometric subsets of data items, communication expressed as geometric strides and gathers, and synchronization similarly – all parametrically. Because communication costs dominate everything, expressing these parametric geometric spaces explicitly to be instantiated at runtime offers some interesting advantages, in compactness of expression of the mapping. In other words, the nature of our optimization algorithm capabilities would shape hardware architectures and these execution model abstractions toward geometric parametric specifications of operations. Furthermore, it is not just simple parametric strided operations that we want to express – certain edge cases in the way that lattices bounded by polyhedra go through affine transformations suggest hardware solutions that can radically simplify certain challenges that would otherwise burden the compiler. We are much more interested in these features than we are things like shared memory or transactional memory.

This interaction between hardware and compiler recapitulates the traditional need to co-design compilers and architectures that dates back at least 30 years, but in the current era is based upon new capabilities in high-level optimizers shaping high level architectures and programming models. The interaction between the programming language and optimizer capabilities are similar. What parallel programming language features are actually useful, vs. just a pain for the compiler? We take the position that the program specification should be as high level, logical, and divorced from the physical issues as possible. Parallel programming languages that emphasize letting the user control the knobs (e.g., specifying data layouts) are in the end doomed. If a programmer actually manages to get something working well in these languages of conflated logical and physical specification, their result will be nonportable and noncompileable as they will have overspecified their program to the degree that future compilers will not be able to extract the original intent, in order to carry the programs

forward to new architectures and generations of machines. Using C as a surface syntax for static control programs that follow some simple rules to make the programs easily abstractable is a much more useful step than any new parallel programming language.

Beside these lofty issues, there is an immediate critical need for an open source common API for specifying the mapped program on multi-core processors. This need is illustrated by the fact that for some of the widely available commercial multi-core systems (e.g., Cell), the number of different mapped program APIs is very large, with multiple commercial and academic solutions to what is essentially the same problem. In some cases, this reflects significant differences in execution model that make sense for the intended applications. In other cases, the differences are gratuitous, just instances in the age old battle to insinuate proprietary platforms and build in switching costs. We can discuss this issue and provide a list of the computation, communication, synchronization and storage mechanisms that we believe should be reflected in the common API. We could put forward the SVM 1.0 specification as a place for people who want to work together on an open spec to start.

Another important issue in multi-core tools and middleware is machine modeling, that is, building a language for the standard formalization of a description of an architecture suitable for use by systems tools. We can give an amusing (and somewhat depressing) account of our experiences in this area (mistakes painfully made) and point to some areas where research is desperately needed, particularly in the case of Morphable architectures, where the Machine model should become a first class writeable entity within our runtime system.

Biography

Richard Lethin is Directing Engineer at Reservoir Labs, and an Adjunct Professor at Yale University. Reservoir Labs provides leading-edge consulting and contract R&D to the computer industry, government, and business end-users, employing state of the art compiler and verification technologies, and also bring proprietary tools developed on a wide variety of architectures. Reservoir Labs uses its technologies and skills to help clients achieve substantial improvements in the capabilities, performance, power savings, cost, and reliability of their computer systems and computer products. Richard received his B.S. in Electrical Engineering from Yale University in 1985 and M.S. and Ph.D. degrees in Electrical Engineering and Computer Science from MIT in 1992 and 1997. Richard's graduate school education was sponsored by a fellowship from the John and Fannie Hertz Foundation, an organization committed to producing leaders in technology for national security. Currently, Richard is working as PI on a number of research projects in compilation and high level optimization.