cadence

Breakfast Bytes Blogs

Silexica: Mastering Multicore

By Paul McLellan (1 Dec 2017)

Since the invention of the microprocessor, it was a dream that it would be possible to build a really powerful computer by taking a lot of cheap simple computers and putting them together. This was especially a dream of hardware designers, who could see their way to addressing the hardware problems, and then the rest was "just" software. That software turned out to be difficult to create. About a decade ago, it became clear that microprocessor clock frequencies could no longer be increased and companies like Intel switched to multi-core processors. However, that software was still not easy to write, and it was hard to make use of large numbers of cores for individual jobs.

There were some tasks that work well on this sort of fabric. Some programs are "embarrassingly parallel" with almost limitless opportunities. The standard example is graphics where each pixel can sometimes be processed individually, with reference only to a few nearby pixels. In fact, it is this that allows GPUs to function, since even when dealing with polygons and shading, it is still the case that one part of the image is largely independent of parts that are not in the immediate neighborhood.

Another important area where finding parallelism is fairly simple is a web server. If you have a thousand cores and thousands of simultaneous users, then just put a few on each core. All of the interactions of one user is independent of the interactions of the other (except where they access the shared database that typically lurks in the shadows behind—making that database highly parallel is a different challenge that has its own set of techniques, such as sharding).

The EDA world has its own tools that spit into easy and hard to parallelize. Physical verification is relatively easy since a chip can be divided up into a large number of pieces that can (largely) be verified separately. Individual rules can often be verified in parallel too. Simulation is hard since there is a common time and so individual parts of the simulation can't get too far ahead of each other without potentially violating causality. Doing just that is the secret sauce in Xcelium.

At the recent Linley Processor Conference, I ran into Jim Brogan, who I used to work with at VaST. He works for a company called Silexica, which is attacking this problem. Demanding embedded applications require multicore platforms, but they are difficult to program. Silexica is actually headquartered in Köln. aka Cologne—famous for its cathedral, "Eau de", and the best selling solo jazz album ever—with a US office near the San Jose Airport. I went over to find out more and to get a demo.

I met with Kumar Venkatramani, who does US business development, as well as with Jim. He explained that they are not focused only on multicore processors, but on multicore, multiprocessor systems, where the code might not even be written in a

single, homogenous code base. This is the market where Tensilica (mostly) plays. You can use a Tensilica processor as a control processor, and many users do, but the more typical use is to have an Arm control processor and use Tensilica processors for offloading audio and video processing, or for a deep-learning neural network inference engine.

Target Markets

Silexica considers parallel processing to be the fourth tectonic shift in computing, following the move from mainframes to minicomputers, then to PCs, and then to cell-phones and datacenters. There has always been a relentless pressure for more computing power, but now the performance of a single core is pretty much capped. You can have more cores, but not faster ones. I like to use the analogy with planes—they haven't gotten appreciably faster, but you can have lots of them. If you want to transport 10,000 people from London to New York, that works great. But getting a single very important person from London to New York in 10 minutes? Forget it.

The target markets for Silexica are embedded high-performance computing. These are markets that need higher performance, but can't just build a bigger datacenter. Within reason, they can add more cores and more specialized offload processors. The markets that are the beachhead—that need solutions now and where theirs is a business imperative—are automotive, aerospace and defense, wireless baseband, and embedded vision. The thing they have in common is highly demanding applications, requiring multicore platforms, that are hard to program efficiently.

This is a problem that is only getting started since the number of cores on an SoC is going up and will continue to go up. I call this Core's Law, that the number of cores is going to double every couple of years, but we just haven't really noticed that much yet since we are on the flat part of the exponential still.

Doing It by Hand

Today, this type of system is mostly programmed by hand, using manual analysis of the code, then manual partitioning, compiling each block of code for its processor/core, and adding all the communication code between the different processors to make it all work. Then, when the next version of the system comes along, with a different selection of cores, that work has to be redone all over again. Obviously, this is far from efficient, especially compared with what software engineers are used to, compile the code and go. Silexica provides an automated flow.

Some code transformations can be done automatically, but others require the code to be analyzed and the programmer has to make a decision. For example, if you are writing some code to process lots of buffers of data, then in sequential programming you would typically allocate a buffer; then have a loop that fills the buffer, does the processing, exports it; then it repeats. When you are done, you free up the buffer. But this loop cannot be unrolled and parallelized since all the instantiations of the loops are competing for the same buffer. Instead, the loop needs to be changed so that nothing happens to start with, then each run through the loop, a buffer is allocated, filled, processed, exported and deallocated. Now the loop can be unrolled as many times as appropriate and several iterations of the loop can be processed in parallel on different cores.

SLX, Silexica's suite of tools, does three things: it analyzes your software to find parallelism and opportunities for further parallelism; then it does the automatic distributions of the various parallel tasks to use the hardware efficiently; and, finally, it organizes the code generation.

Describing the System

The platform needs to be described so that the SLX system knows what the processing elements are, but also the memory hierarchy, the communication fabric, and down to

details like power/frequency domains. Luckily, they don't have to re-invent the wheel since they are riding on the work underway with the Multicore Association, who have a standard in development called SHIM 2. SHIM is a somewhat ungainly acronym for Software-Hardware Interface for Multi-many-core. Of course, a "shim" is also something used to fill a small gap. If you have ever rebuilt the gearbox on your car (which doesn't seem to be part of being a teenager anymore) then you will have put in some shims, for sure.

SHIM provides a way (in XML) for hardware vendors (or whoever put the system together) to describe the system in a way that tools like SLX can consume. It is worth emphasizing that this is not a complete description of the hardware, it is just the things that the software cares about. It contains some measure of performance since that is important assessing the quality of any partitioning and also giving an estimate, not 100% accurate, of the overall system performance.

Demo

Jim gave me a demo of the system. The target system was a "toy" system that was close enough to real-world applications that you could see what was going on. The system took a stream of video frames from a USB camera and then found black-on-white squares in the feed. (It isn't really necessary to understand the details of the algorithm that finds edges and then sees if they are squares and so on.) What is most important is that it is computationally demanding; otherwise, it wouldn't benefit from more than one core.

By analyzing the call graph and tracing execution, SLX searches for opportunities for parallelization, both the low-hanging fruit that can simply be run on multiple cores and opportunities where the code requires re-factoring to break possibly unnecessary dependencies (like in the buffer example I talked about earlier). On this example, the expected speedup is 2.4X (if all the refactoring is done, SLX estimates an idea improvement as high as 3X).

On a single thread, the recognition algorithm runs at seven frames per second (by definition, a speedup of 1.00). With four workers, that goes up to nearly 11fps. With some refactoring and three threads, that speed goes to nearly 16fps. With three worker processes, that goes to 19fps, a speedup of 2.68X.

Legacy Code

The challenge getting a good speedup on a multicore system is two-fold. One problem is that many problems don't have a lot of things that can be done in parallel, no matter how smart the algorithms or the programmers. It's like trying to mow your lawn with an infinite number of lawnmowers. If your "lawn" is 100 acres, you can probably use a lot of mowers, but for a standard suburban lawn you can't use many before coordinating them all would take more effort than just getting on with the mowing.

The second problem is that most code has not been written to be run on a large number of threads. Most code is legacy code, either a code-base written over years, or libraries and open-source stacks from third parties. For example, Cadence tools have been around a long time and contain tens of millions of lines of code. Even if the engineering investment can be justified, there is a limit to how much of a complex tool can be re-written in the few years between process nodes. The same parallelization issue rears its head here, in that you can't put a hundred engineers on the project to accelerate it—programming doesn't work that way.

Putting these constraints together means that most code remains unchanged; some code can be added or altered to drive all the parallelization, but in many cases, there are inherent interactions in the code that limit

how much can be done. In the end, you run into some version of Amdahl's law, that if 10% of the code cannot be parallelized, then the maximum theoretical speedup is 10X (once everything else goes so fast it takes no time at all).

More Information

More information on Silexica and SLX are on their website (also in Chinese, Japanese and Korean ...but not German). Start here for information on Cadence's Tensilica multicore configurable processors.