Optimizing compilers for ADAS applications

Alexander Herz - March 06, 2017

All major OEMs and software suppliers of the automotive industry are committed to advanced driver assistance systems (ADAS). A close look, though, raises questions on the demands ADAS applications place on compilers and toolsets. There are differences between traditional automotive applications and ADAS, and current compilers need some adaptations to better address ADAS needs.

ADAS applications as a challenge

To better support the task of driving autonomously, vehicles need to be much more aware of their surroundings. Several new sensors (Radar, Lidar, cameras, etc.) can be used to detect road markings, other vehicles, obstacles, and other relevant environmental data with high resolution (Fig. 1). In the past, it was common practice for automotive systems to process only individual measurements from specific actuators (steering angle, pedal positions, various engine sensors, etc.) in real time.

As is common with physical measurements, however, the environmental data acquired for ADAS applications are subject to noise (Fig. 2) and measurement errors. Therefore, the data require electronic post-processing by hardware and software before they can be used for their ultimate purpose, i.e., to automatically offload decisions from the driver. This post-processing is not always
dealing with individual measurements like before, however. Quite frequently, data from different sources get consolidated (sensor fusion) for reduced error susceptibility.

In order for the ADAS to automatically make decisions on the driver’s behalf, it must process a tremendous amount of data in real time. Further, that data is complex. Traditionally, just isolated sensor data involving only some integer or fixed-point numbers with 1 - 5 kbps data rates needed to be processed. Today, data are often provided as floating-point numbers (floats/doubles) at high rates. Camera images, for instance, provide approximately 340 kbps and radar data, around 1.5 Mbps.

![Figure 2: The noisy signal of an environment sensor and its filtered result](image)

Obviously, ADAS applications require a lot more processing power than traditional automotive applications. But currently it is very hard to predict which high-performance hardware architectures will prevail for these kinds of applications. So, because ADAS applications must be produced in a reusable and cost-efficient manner, it is clear that compiler support will be required for all architectures. This requirement mandates the use of abstract, portable design methodologies (e.g. C++11/14), model-based design and additional technologies including parallel programming (e.g. OpenCL, Pthreads). Furthermore, highly optimized, certified libraries will be required to implement standard operations efficiently, safely, and with maximum hardware independence. Because ADAS applications intervene with the driving process, these applications and the hardware used to execute them must also adhere to relevant safety standards (ASIL-B or higher; ISO 26262).

**Finding a suitable hardware architecture**

For companies developing ADAS applications, the fact that no specific hardware architecture has prevailed until now creates a risk. In general, major hardware accelerators -- including the Nvidia GPU derivatives (Drive PX) -- provide adequate computational power in the Teraflops range for the data-parallel parts of ADAS applications. However, apart from lacking sufficient safety features,
these devices are rather cost-intensive regarding their power consumption and purchasing price. On the other hand, typical architectures for safety-critical applications up to ASIL-D (incl. AURIX or RH850) have not yet utilized some hardware based opportunities to achieve higher data rates because these will be hard to certify according to ASIL-D.

OEMs or large suppliers of ADAS systems therefore are in danger of selecting an architecture that may fail in the market because it is too large, too expensive, or cannot meet the safety requirements. On the other hand, there is a risk in selecting an architecture that fully supports safety-critical applications that it is too small for the more demanding computations. During the development process, it might turn out that the envisioned application cannot be implemented for efficiency reasons.

Thus, the requirements of ADAS projects are quite complex. On the one hand, it is mandatory that developers create very efficient, target specific code, meet all safety goals, and minimize the risks outlined above. On the other hand, portable and high-level design methods are necessary to enable cost-effective application development. These high level design requirements mandate modifications of the embedded compilers that were originally designed for traditional embedded applications.

**Efficiency of code structure**

One necessary, new compiler feature is the need to support the typical code structures of ADAS applications in order to create highly-efficient code for this kind of application. The data structures used and the operations they are subjected to in an ADAS application differ fundamentally from those found in classical applications, and the code used for sensor fusion and analyzing sensor data is commonly generated using model-based tools like BASELABS. For instance, the speed-relevant part of ADAS code often involves arrays (vectors and matrices) of floats and doubles, which are subjected to linear algebraic operations such as matrix multiplication, inversion, SVD (singular value decomposition), and the like. Using these operations, the system combines the sensor data arrays to compute an abstract representation of the environment which is then used as the basis for decision making (e.g., for detecting objects in an image or for tracking and assigning the spatial position of an object).

**Highly optimized libraries**

This heavy use of arrays and floating-point data means that entirely new optimizations are required for compilers in order to provide efficient results. For instance, the most commonly used linear algebraic functions are typically provided by libraries that are highly optimized for the specific target architecture. All computations not included in the libraries, therefore, must be well optimized by the compiler in order to prevent these computations from becoming the bottleneck.

Many of the performance-critical computations in ADAS applications are based on a set of standard linear algebraic operations. The overhead resulting from porting such ADAS applications to different target architectures can be reduced dramatically if a standard interface is used for these standard operations. Libraries supporting a standard interface and which are highly optimized for the specific target architecture are available for the most relevant hardware platforms, including LAPACK from Tasking for embedded, cuBLAS from Nvidia, and Intel's Integrated Performance Primitives.
Quite often, these libraries are as much as an order of magnitude faster than open-source offerings or in-house implementations. Consequently, a standard interface-based application can immediately achieve excellent efficiency even on new target platforms without the need for designers themselves to optimize and test the underlying, performance-critical computations in the target specific library. Note, however, that not all libraries are adequately certified for safety-critical applications or suitable for embedded systems.

**Parallel programming**

An additional new capability required from embedded compilers is the support of current languages like C++11/C++14. The goal in ADAS design is to improve the code’s reusability and to achieve more with fewer lines of code, without giving up the efficiency provided by closeness to the hardware. C++ classes and inheritance are time-tested methods to write such code on a higher, more abstract level.

C++11 and later variants support these methods but offer significant advantages over the older C99 language standard. Furthermore, C++11 (and C11) finally provide the opportunity to write portable, parallel programs. The computational overhead, considering all response-time requirements, of many ADAS applications often exceeds the capabilities of sequential processing implemented on a single core. Parallel and multicore processing, therefore, is a common ADAS system requirement.

Older standards like C99 do not acknowledge parallelism, so programmers using those languages must have excellent hardware and compiler knowledge to correctly write a parallel program. Programmers must, for instance, exclude specific data ranges and code sections from parallel accesses in order to ensure that no data updates are lost or incorrectly read during access. Programmers must also insert barriers (mutexes) into the code to keep critical sections from being executed by more than one core at a time.

However, the barrier insertion technique only works if the compiler is aware of these barriers. Without such awareness, the compiler may move code sections out of the protected parts during compiler or hardware optimizations. Before the advent of C11/C++11, there was no uniform way to notify the compiler of such barriers. So, programmers had to disable important optimizations altogether, resulting in significant efficiency degradations, or they misused attributes like ‘volatile’ to restrict compiler optimizations. It has now become generally accepted that using the ‘volatile’ attribute is not sufficient for writing correct and portable parallel code.

Mutexes and the like are now part of the C++ standard, however, so a C++11 compiler is aware of all barriers. The compiler can therefore prevent the application of optimizations when necessary without incurring any unnecessary speed penalties. Instead of using optimizations, programmers use the ‘atomic’ attribute introduced by C11/C++11. With ‘atomic’, the compiler generates code that addresses the hardware so as to expose the expected behavior and generates a minimum performance overhead. Programmers thus can focus their efforts on their main task, i.e. the code’s functionality, instead of trying to generate specific code patterns via unsuitable means like ‘volatile’ and optimization inhibition.

Unfortunately, it is generally not possible to detect all the program sections and data that have incorrect protection from parallel access. Sometimes, programs with incorrect protection will not generate any compiler errors and thus appear to operate correctly. Yet these programs can spontaneously produce false results as a result of subtle timing issues. These errors generally
appear only after very long testing times. Further, they are difficult to reproduce because they depend on relative execution times and time-related disturbances within the system.

Thus, it is not quite trivial to write correct parallel code even when using C11/C++11. Self-written parallel code also bears the risk of being correct but only marginally faster (or even slower) than the easier-to-maintain functionally equivalent sequential code. Fortunately, libraries like EMB² and LAPACK can be used with relatively little risk, as they were written by experts in this field. As an additional advantage, these libraries ensure a relatively large speed increase due to their parallelism and optimization.

**Hardware accelerator support**

New compiler capabilities are also required to address increasingly heterogeneous hardware architectures featuring widely differing cores (ARM, AURIX, RH850, Nvidia, etc.) and supplemental accelerators (e.g., FFT in AURIX, SIMD in ARM and Nvidia). Several approaches are possible. One such approach is support for intrinsics, the most straightforward method to support hardware accelerators. These constructs can be used to address special hardware instructions from C/C++.

At the next level compilers could support special high-level languages (most of which are similar to C) that accelerator suppliers provide to enable designers to address their hardware efficiently. Examples include OpenCL for Nvidia (compiler from Nvidia), C for GTM (compiler from Tasking), and extended C for EVE from TI. Although these high-level languages require compilation and optimization, their closeness to the underlying hardware simplifies the whole process.

As a final option, compilers could automatically detect code areas that can be executed efficiently by an accelerator in order to automatically generate the appropriate code, as Intel’s icc does with SIMD architectures. However, this fully automatic discovery is restricted in most cases because standard C/C++ code is not explicitly written for the specific accelerator. Still, with minor code modifications the compiler’s automatic discovery can yield excellent results, although a suitable tuning tool is indispensable in order to find and implement the necessary changes with reasonable overhead.

Unfortunately, many heterogeneous hardware architectures mandate that each programmable unit be addressed with its own compiler. In order to avoid dealing with an excessive number of incompatible tools, which would generate new safety risks, it is advisable to use tool environments that can address all programmable units and ensure mutual compatibility between the tools. The Infineon AURIX/TriCore tool environment from Tasking, for example, can be used to program and debug all units of the architecture from a single IDE. Interactions between units can be controlled and monitored more safely because symbol information is compatible between the different units.

**Safety requirements for ADAS applications**

In order to meet the specific safety requirements of ADAS applications, all tools – including modeling tools, compilers, and analysis tools -- and software components (OSes, libraries, etc.) relevant for these applications must be developed and qualified according to ISO 26262.

Some of the newer safety requirements can be understood using neural networks (Fig. 3). Neural networks are software components that are often used for detecting and processing sensor data in
ADAS applications. Although there are interesting prototypes based on neural networks, it is still not clear how a correct behavior of these networks can be ensured in extreme situations.

At the moment, no known procedure can guarantee that neural networks always behave correctly without any risks for road users. Therefore, one cannot let neural networks make safety-critical decisions without a suitable supervisory entity. In addition to the hardware for the neural networks themselves, which will issue a hard-to-predict decision proposal based on the input data (e.g., accelerate to 100 km/h, pass on the left side, actuate an emergency stop, etc.), there must be a supervisory entity running on hardware featuring the highest safety certification (ASIL-D). The latter will operate using predictable algorithms to check if the proposal made by the neural networks can be executed safely or if a safer alternative should be chosen (Fig. 4). For instance, the supervisory entity would check if the passing maneuver proposed by the neural networks can be executed without any risk. For this purpose, it will use its own, predictable calculations to check if there are no obstacles etc. Predictable algorithms are still being researched in some areas (e.g., data fusion) in order to create an effective supervisory entity.
Many of the predictable algorithms for ADAS applications are based on linear algebraic calculations supported by LAPACK and others. Optimized solutions can be used to implement these algorithms efficiently and safely on various target platforms.

The remaining parts of ADAS applications can be certified using several tools and processes that are helpful to meet various ISO 26262 requirements. Simple programming errors (including non-initialized data) can be detected efficiently using static analysis (Polyspace, Klocwork, etc.). For detection of safety relevant access violations (software components with different ASILs accessing each other and creating protection faults in the Memory Protection Unit – MPU), it is beneficial to use the compiler’s integrated safety checker extension, which can be used to define different safety categories (e.g., ASIL A to D), to assign data and functions used in the project to different safety categories, and to manage the access privileges between these categories.

This information can then be used for two purposes. First, the compiler is unable to perform, certain optimizations (reverse inlining, code compaction) because these optimizations could result in safety access violations if they are performed without taking the access privileges into account. Second, the same information can be used with a safety checker tool to identify undetected access violations that would generate MPU exceptions without any additional testing overhead and with high code coverage.

In order to qualify the tools (modeling tools, compilers, static analysis, safety checker, etc.) according to ISO 26262, most manufacturers provide an ISO Kit greatly simplifying the necessary process. In this context, it is helpful to work with tools and manufacturers with a long track record in the automotive area. ISO 26262-8 uses the term proven in use for this purpose: It is assumed that a tool that was frequently used over a long time with few problems will probably be less fault-prone than a new one.
Apart from addressing the safety risks outlined above, the compiler and its associated tools can help to mitigate the design risks associated with ADAS applications.

**Controlling the design risks**

Selecting an inappropriate combination of hardware, libraries and development tools represents a significant design risk in the ADAS space. At the moment, it is virtually impossible to predict the efficiency of a specific combination of hardware and software for a specific ADAS application.

For instance, there is a risk that the hardware used is too large and energy-hungry, or that it is too small. Unfortunately, no continuous spectrum is available today: hardware turning out as too small cannot be replaced straightforwardly by compatible hardware which is slightly bigger in size. You can select between a very large configuration featuring a lower safety certification level, or a significantly smaller configuration with a higher certification level. Changing between both options currently entails excessive overhead because of the significant differences between the architectures and the code structures required for their optimal usage.

At the moment, this problem cannot be solved by the compiler alone. However, the compiler can make the risk manageable if it is paired with a suitable software environment. If a good modeling solution based on hardware-specific, highly efficient linear algebra libraries is combined with a compiler environment meeting the typical requirements of ADAS applications, the resulting comprehensive solution will be more than just the sum of its parts.

ADAS applications can be largely implemented in a hardware-independent manner by using modeling solutions. The underlying compiler and the libraries enable the generic, model-based implementation to be ported to widely differing hardware platforms with minimum overhead and high efficiency. Minor inefficiencies resulting from this can be identified and eliminated quickly by profiling.

For generic code, a combined solution of this kind would provide a short, defined path towards optimum efficiency on various target platforms. With an additional set of benchmarks documenting the efficiency of the combination of all tools depending on the target hardware and the resolution of the input data, these data can be used to predict the expected efficiency of a new ADAS application on new target hardware with relatively good accuracy. This greatly reduces the risk of selecting target hardware that is inappropriate for an envisioned ADAS application.

None of today's compilers meets the full spectrum of requirements. Some of the tools and libraries required for the full solution are not available anywhere today. Thus, the roadmap for the compiler is clear: The remaining requirements must be addressed without losing sight of the entire solution. For example, Tasking is planning new products, including certified LAPACK libraries for AURIX, supported by collaborations with tool partners and customers.

—Alex Herz received his diploma in theoretical physics at the Technical University of Munich in 2009. He obtained his Dr. rer. nat. degree in 2015, and is now working as a compiler specialist and head of product development at TASKING.
Also see:

- The mythical software engineer
- Using MISRA C and C++ for security and reliability
- Attaining functional safety: Standards, certification, and the development process
- Safety & security architecture for automotive ICs