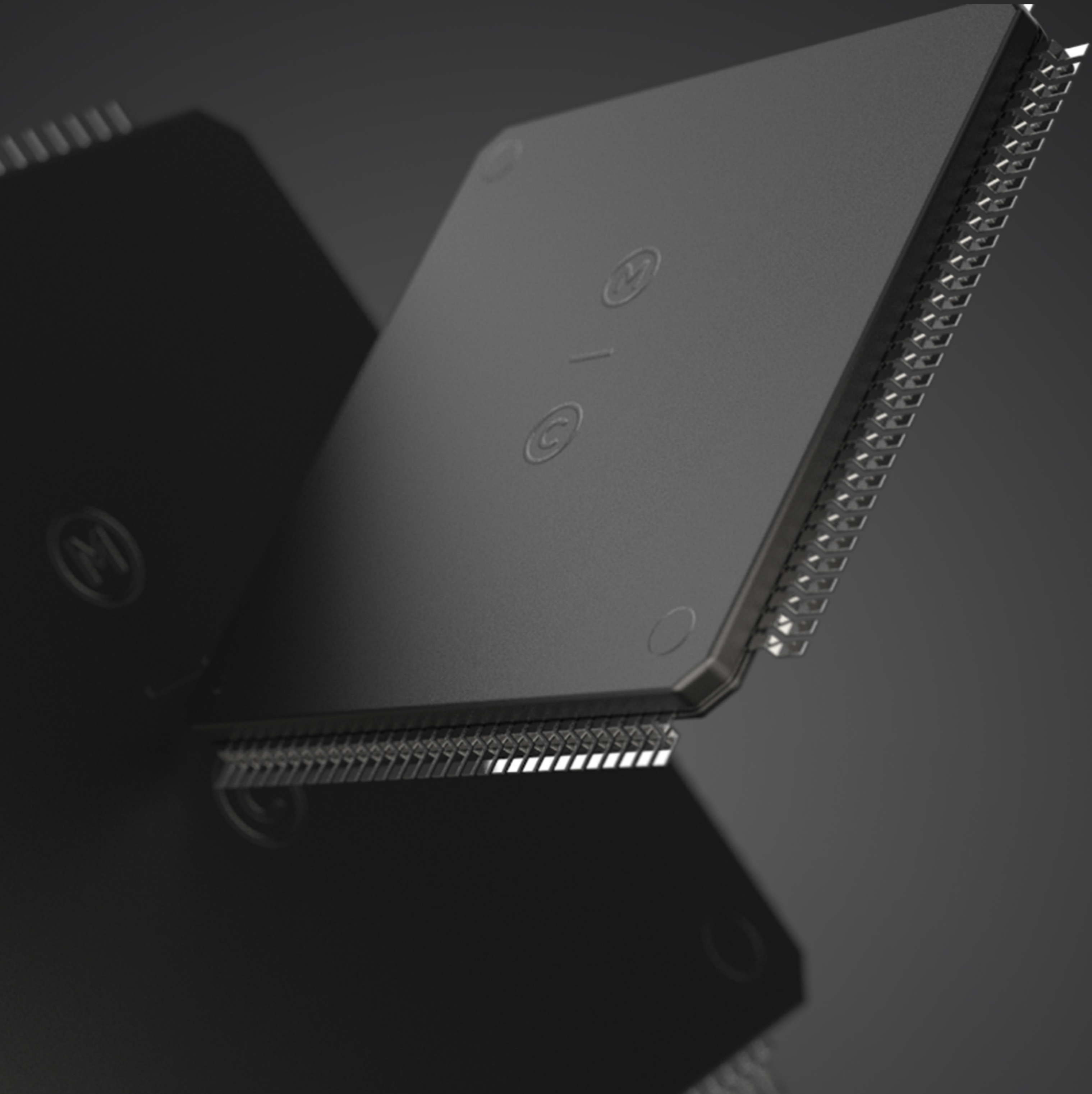
NEW COMPILER CHALLENGES:

Optimized for ADAS Applications



NEW COMPILER CHALLENGES: OPTIMIZED FOR ADAS APPLICATIONS

INTRODUCTION

New Compiler Challenges: Optimized for ADAS Applications

The compiler is a key tool for the cost-efficient design of ADAS applications. However, the tools currently available must be better adapted to this challenging task. This includes considering the code structures and specific safety requirements typical of ADAS applications.

By Dr. Alexander Herz

When planning a compiler technology roadmap, you will inevitably touch the issue of advanced driver assistance systems (ADAS), which all major OEMs and software suppliers of the automotive industry are committed to. A closer look, though, will raise some questions: What requirements are placed on compilers and toolsets by ADAS applications? Are these things related at all? What are the differences between traditional automotive applications and ADAS applications?

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 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, the environmental data acquired for ADAS applications are subject to noise (Fig. 2) and measurement errors. Therefore, they 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.

Quite frequently, data from different sources are consolidated (sensor fusion) for reduced error susceptibility. In order to automatically make decisions on behalf of the driver, a tremendous amount of data must be processed in real time. For instance, camera images (approx. 340 kbps) or radar data (approx. 1.5 Mbps) are often provided as floating-point numbers (floats/doubles). Traditionally, just isolated sensor data (involving only some integer or fixed-point numbers with 1 - 5 kbps) needed to be processed. Therefore, ADAS applications obviously require a lot more processing power than traditional applications.

Currently, it is very hard to predict which high-performance hardware architectures will prevail for these kinds of applications. However, it is clear that compiler support will be required for all architectures because ADAS applications must be produced in a reusable and cost-efficient manner. This 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.

As ADAS applications intervene with the driving process, these applications and the hardware used to execute them must adhere to relevant safety standards (ASIL-B or higher; ISO 26262).

FINDING A SUITABLE HARDWARE ARCHITECTURE

There is a risk for companies developing ADAS applications associated with the fact that no specific hardware architecture has prevailed until now. 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. 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 to select an architecture that fully supports safety-critical applications but 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 to create very efficient, target specific code, to meet all safety goals and to 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.

This brings us back to the initial question concerning which new capabilities an embedded compiler must provide in order to meet the just mentioned requirements. A 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.

EFFICIENCY OF CODE STRUCTURE

What does an ADAS-typical code structure look like in detail? The speed-relevant part of ADAS code often consists of arrays (vectors and matrices) of floats and doubles, which are subjected to linear algebraic operations (matrix multiplication, inversion, SVD, etc.). Using these operations, sensor data is combined 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).

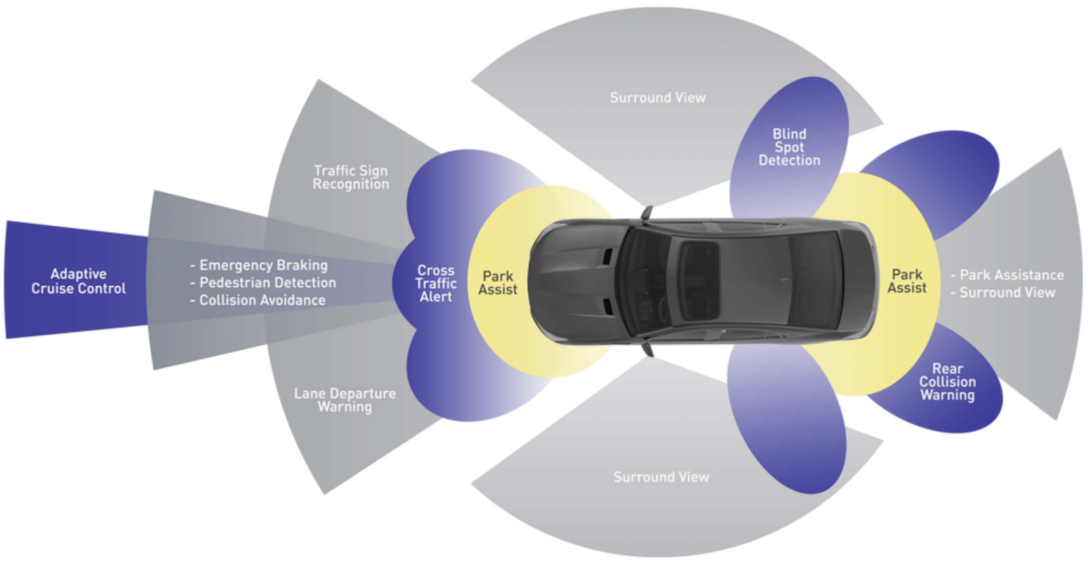


Figure 1: Sensor-based monitoring areas of different ADAS applications

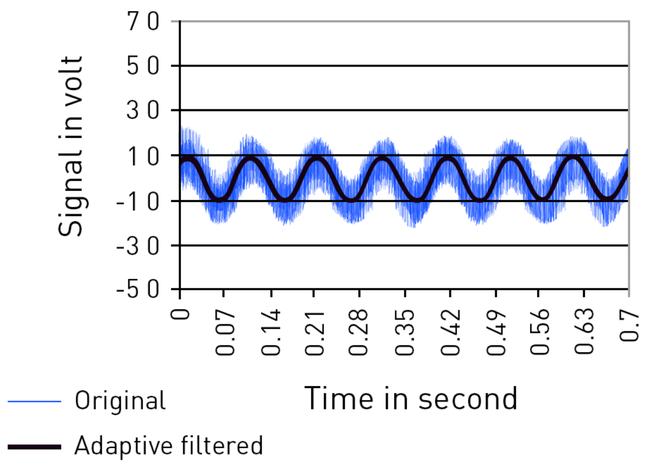


Figure 2: The noisy signal of an environment sensor and its filtered result

The code used for sensor fusion and analyzing sensor data is commonly generated using model-based tools like BASELABS™. The data structures used and the operations they are subjected to in an ADAS application differ fundamentally from those found in classical applications. Entirely new optimizations are required for compilers in order to provide efficient results for this kind of applications.

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 must be well optimized by the compiler in order to prevent these computations from becoming the bottleneck.

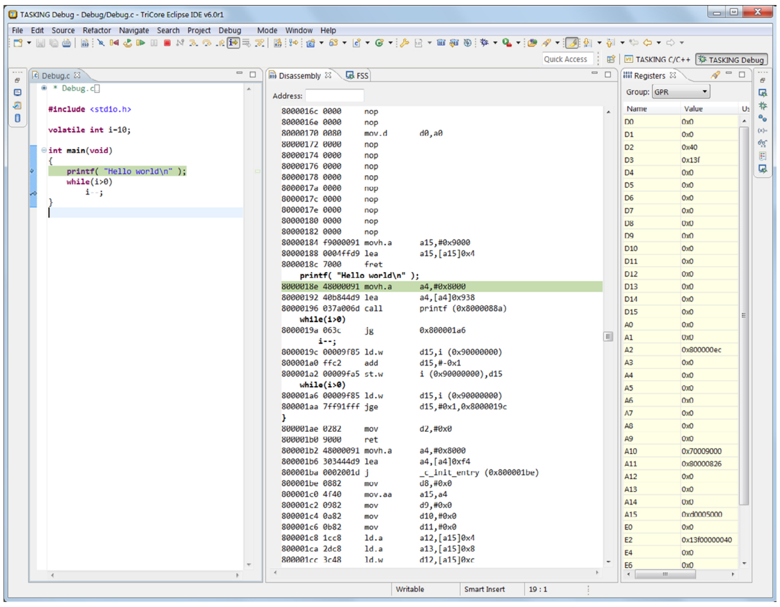
HIGHLY OPTIMIZED LIBRARIES

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 like LAPACK is used for these standard operations. Libraries supporting this 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 or Intel™ 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 LAPACK interface-based application can immediately achieve excellent efficiency even on new target platforms without the need for designers to optimize and test the underlying, performance-critical computations in the target specific library themselves. Note that not all libraries are adequately certified for safety-critical applications or suitable for embedded systems. An additional new capability required from embedded compilers is the support of parallel algorithms through current languages like C++11/C++14. The computational overhead of many ADAS applications exceeds the capabilities of sequential implementations on a single core when considering all response-time requirements.

HOW TO WRITE PORTABLE, PARALLEL PROGRAMS

C++11 and later variants offer significant advantages over the older C99 language standard. C++ classes and inheritance are time-tested methods to write code on a higher, more abstract level. The goal is to improve the code's reusability and to achieve more with less lines of code without giving up the closeness to the hardware required for efficiency reasons. Furthermore, C++11 (and C11) finally provide the opportunity to write portable, parallel programs.

As older standards like C99 do not acknowledge parallelism, programmers must exercise their excellent hardware and compiler knowledge with the intention to write a parallel program which is hopefully correct. In order to ensure that no data updates are lost or incorrectly read as a result of parallel accesses, specific data ranges and code sections must be excluded from parallel accesses. Barriers (mutexes) are inserted into the code to keep critical sections from being executed by more than one core at a time. However, this only works if the compiler is aware of these barriers to prevent code sections from being moved out of the protected parts by compiler or hardware optimizations.


TASKING Debug - Debug/Debug.c - TriCore Eclipse IDE v6.0r1
File Edit Source Refactor Navigate Search Project Debug Mode Window Help
Quick Access
TASKING C/C++
TASKING Debug
Debug.c
Disassembly
FSS
Registers
Address:
Group: GPR
Writable
Smart Insert
19 : 1

```
* Debug.c

#include <stdio.h>

volatile int i=10;

int main(void)
{
    printf( "Hello world\n" );
    while(i>0)
        i--;
}
```

```
8000016c 0000     nop
8000016e 0000     nop
80000170 0080     mov.d      d0,a0
80000172 0000     nop
80000174 0000     nop
80000176 0000     nop
80000178 0000     nop
8000017a 0000     nop
8000017c 0000     nop
8000017e 0000     nop
80000180 0000     nop
80000182 0000     nop
80000184 f9000091 movh.a     a15,#0x9000
80000188 0004ffd9 lea        a15,[a15]0x4
8000018c 7000     fret
    printf( "Hello world\n" );
8000018e 48000091 movh.a     a4,#0x8000
80000192 40b844d9 lea        a4,[a4]0x938
80000196 037a006d call       printf (0x8000088a)
    while(i>0)
8000019a 063c     jg         0x800001a6
        i--;
8000019c 00009f85 ld.w       d15,i (0x90000000)
800001a0 ffc2     add        d15,#-0x1
800001a2 00009fa5 st.w       i (0x90000000),d15
    while(i>0)
800001a6 00009f85 ld.w       d15,i (0x90000000)
800001aa 7ff91fff jge        d15,#0x1,0x8000019c
}
800001ae 0282     mov        d2,#0x0
800001b0 9000     ret
800001b2 48000091 movh.a     a4,#0x8000
800001b6 303444d9 lea        a4,[a4]0xf4
800001ba 0002001d j          _c_init_entry (0x800001be)
800001be 0882     mov        d8,#0x0
800001c0 4f40     mov.aa     a15,a4
800001c2 0982     mov        d9,#0x0
800001c4 0a82     mov        d10,#0x0
800001c6 0b82     mov        d11,#0x0
800001c8 1cc8     ld.a       a12,[a15]0x4
800001ca 2dc8     ld.a       a13,[a15]0x8
800001cc 3c48     ld.w       d12,[a15]0xc
```

Name	Value	Us
D0	0x0	
D1	0x0	
D2	0x40	
D3	0x13f	
D4	0x0	
D5	0x0	
D6	0x0	
D7	0x0	
D8	0x0	
D9	0x0	
D10	0x0	
D11	0x0	
D12	0x0	
D13	0x0	
D14	0x0	
D15	0x0	
A0	0x0	
A1	0x0	
A2	0x800000ec	
A3	0x0	
A4	0x0	
A5	0x0	
A6	0x0	
A7	0x0	
A8	0x0	
A9	0x0	
A10	0x70009000	
A11	0x80000826	
A12	0x0	
A13	0x0	
A14	0x0	
A15	0xd0005000	
E0	0x0	
E2	0x13f00000040	
E4	0x0	
E6	0x0	

Figure 3: A compiler translates the C source code (see Debug.c on the left) into machine instructions (see Disassembly in the middle). On the right hand side, the values of the machine registers are shown. The assembly instructions (nop, mov, …) read, modify and write registers as well as memory.

As these barriers are now part of the standard, a C++11 compiler is aware of all barriers. It can therefore prevent optimizations where necessary without incurring any unnecessary speed penalties. Before the advent of C11/C++11, there was no uniform way to notify the compiler of any barriers. Important optimizations therefore had to be disabled altogether, which can lead to significant efficiency degradations. Alternatively, attributes like 'volatile' were misused to restrict compiler optimizations. In the meantime, it is generally accepted that the 'volatile' attribute is not sufficient to write correct and portable parallel code. Instead, it is necessary to use the 'atomic' attribute introduced by C11/C++11. With 'atomic', the compiler generates suitable code to address the hardware in such a way that the code exposes the behavior expected from the standard and so that a minimum performance overhead is generated. Programmers thus can focus their efforts on their main task, i.e. the code's functionality instead of trying to generate specific code patterns with unsuitable means like 'volatile' and optimization inhibition.

Interestingly, programs that do not use 'atomic' (or other barriers based on 'atomic') to protect variables from potential parallel accesses are meaningless according to the C11/C++11 standards. Thus, compilers could generate any code here, although an error message would be the most meaningful output. Unfortunately, it is generally not possible to detect all program sections and data which have incorrect protection from parallel access. Sometimes, programs with incorrect protection will not generate any compiler error and thus appear to operate correctly while they spontaneously produce false results, depending on subtle timing issues. These errors generally occur only after very long testing times and cannot be reproduced because they depend on relative execution times and time-related disturbances within the system which are very hard to reproduce.

Therefore, it is not quite trivial to write correct parallel code using C11/C++11, but 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. Self-written parallel code, in turn, bears the risk to yield parallel code that is correct but only marginally faster or even slower than functionally equivalent sequential code, which would even be easier to maintain. It is still an expert task to write correct, efficient parallel code especially for fine-grained data structures.

New parallel data structures are the subject of new scientific publications. 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 und NVIDIA).

HARDWARE ACCELERATOR SUPPORT

Intrinsics are 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, special high-level languages, most of which are similar to C, are supplied enabling designers to address their hardware efficiently. Although these high-level languages require compilation and optimization, their closeness to the underlying hardware simplifies the whole process (OpenCL for NVIDIA, compiler from NVIDIA; C for GTM, compiler from TASKING; extended C for EVE from TI). As a final option, compilers can automatically detect code areas that can be executed efficiently by an accelerator in order to automatically generate the appropriate code (SIMD, icc). However, this fully automatic discovery is restricted in most cases because standard C/C++ code is not explicitly written for the specific accelerator. Quite often, minor code modifications yield excellent results, although a suitable tuning tool is indispensable in order to find and implement the necessary changes with reasonable overhead.

Furthermore, 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. For instance, the TASKING tool environment for AURIX 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, tools (modeling tools, compilers, analysis tools) and software components (OS's, 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. 4). Neural networks are software components that are often used for detecting and processing sensor data in ADAS applications.

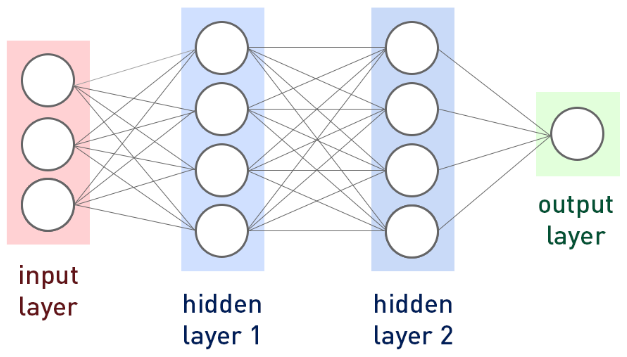


Figure 4: Schematic diagram of a neural network

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 100km/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).

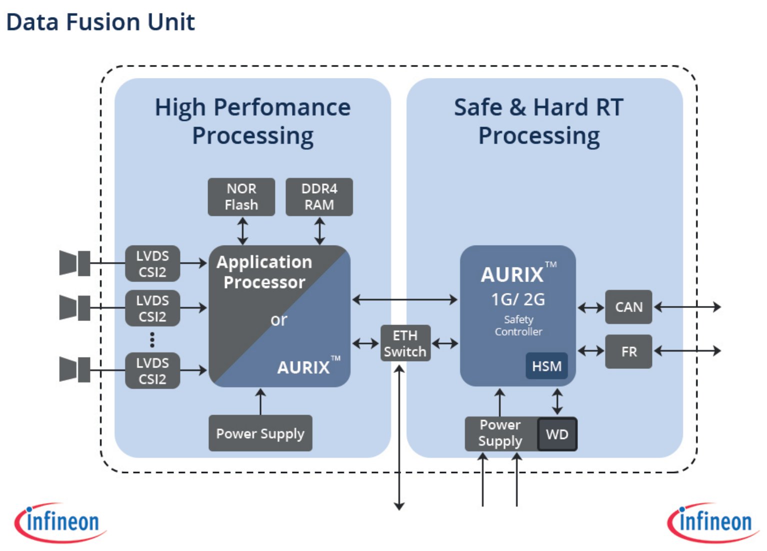


Figure 5: High-performance processing for neural networks and image processing combined with safe processing for the supervisory entity

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. 5). 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 including the LAPACK Performance Libraries from TASKING 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 TASKING compiler with its associated safety checker tool support. The compiler's integrated safety checker extension 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 the TASKING safety checker tool (which can also be obtained for third party compilers, if desired) 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.

SAFETY REQUIREMENTS FOR ADAS APPLICATIONS

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, this 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.

Although the TASKING compiler already supports many of the requirements mentioned above, 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. Happy to accept this challenge, TASKING is planning new products, including certified LAPACK libraries for AURIX. These will be introduced in the near future, supported by collaborations with tool partners and customers.