Using Parallel Programming Models for Automotive Workloads on Heterogeneous Systems - a Case Study

Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez and Andreas Koch
Embedded Systems and Applications Group, TU Darmstadt, Germany.
{sommer stock, solis, koch}@esa.tu-darmstadt.de

***Abstract*—Due to the ever-increasing computational demand of automotive applications, and in particular autonomous driving functionalities, the automotive industry and supply vendors are starting to adopt parallel and heterogeneous embedded platforms for their products.**

However, `C` and `C++`, the currently dominating programming languages in this industry, do not provide sufficient mechanisms to target such platforms. Established parallel programming models such as OpenMP and OpenCL on the other hand are tailored towards HPC systems.

In this case study, we investigate the applicability of established parallel programming models to automotive workloads on heterogeneous platforms. We pursue a practical approach by re-enacting a typical development process for typical embedded platforms and representative benchmarks.

***Index Terms*—embedded, automotive, parallel programming, heterogeneous, OpenMP, OpenCL, CUDA**

I. Introduction

In recent years, the computational demands of automotive applications have been steeply increasing, in particular with the introduction of advanced driver-assistance (ADAS) and, at least partially, autonomous driving (AD) functionalities.

As these functionalities require the processing of complex, compute-intensive algorithms with high performance, the automotive industry faces challenges similar to those encountered by the high-performance computing (HPC) community about a decade ago. The computational power provided by single-core embedded processors is not sufficient anymore to meet latency and/or throughput requirements.

In reaction to these challenges, the automotive industry is starting to turn towards parallel and heterogeneous platforms [1], e.g., combining multi-core CPUs and GPUs. These multi-core processors as well as accelerators typically require programming language mechanisms to express parallelism and leverage their computational power. However, `C` and `C++`, the currently dominating programming languages in the automotive field [2], do not provide sufficient mechanisms. As a consequence, the automotive industry needs to adopt parallel and heterogeneous programming models.

While there is a number of well-established standards for parallel and heterogeneous programming in the HPC community, the embedded target platforms in the automotive industry differ significantly from the HPC systems these programming models were tailored towards. The thermal and power budget, the computational power and the coupling between host CPU and accelerator of embedded, heterogeneous platforms deployed in automotive vehicles differs significantly from HPC systems.

So although the HPC programming models can serve as a solid base, they are not usable "out-of-the-box", and will require adaption for use in automotive usage scenarios.

The aim of this case study is to investigate programming models established in the HPC field with regard to their applicability in embedded automotive applications. The intent of this case study is to provide insights into how well established programming models are suited for use in automotive applications, and how they could be improved and extended for automotive use and target platforms.

Note that we do not focus on the raw performance only, but also consider other important aspects for a *practical* usage of this parallel programming models in industry, such as programmer productivity and maintainability.

To this end, we have developed a practical approach described in Section III. In our evaluation, we present a detailed analysis of important figures, e.g., programmer productivity and effort to reach certain levels of speedup over a serial implementation.

II. Related Work

Prior studies that investigated the usability and maintainability of parallel programming models, such as [3] or [4], focused on HPC applications and algorithms, whereas our case study is focused on automotive, embedded applications. Most of these studies were also conducted as *classroom studies*, with novice programmers as developers. In this work, we explicitly do not consider the time required to learn a parallel programming model, and have experienced developers implement the kernels.

In other work, such as [5] or [6], the authors developed static and dynamic analyses to predict the performance of parallel implementations of algorithms on different embedded and also heterogeneous platforms. While they consider the underlying parallel characteristics of the algorithms and how well they map to the platforms, we investigate how well the parallelism in an algorithm can be *expressed* with the different parallel programming models, and how much programming effort is required to do so.

III. Approach/Methodology

The central aim of this case study is to investigate the use of existing (often HPC-centric) programming models for the implementation of automotive computation tasks on parallel, heterogeneous platforms, and to identify potential areas for improvement of the existing standards or tool implementations.

To this end, and in contrast to previous investigations (e.g., [2]), we take a practical and quantitive approach, based on real implementations of representative computational problems. The basic idea of our approach is to re-enact the typical development process of migrating an existing, serial implementation of an algorithm to a parallel, heterogeneous platform. With the continued integration of such platforms into automotive vehicles, many OEMs and component suppliers will be confronted with this task.

Through this re-enactment, we can investigate all relevant usability aspects of the programming models in detail, as well as the ecosystem of supporting tools.

In the following sections we will describe the individual steps of our approach in more detail.

A. Identification of relevant programming models

In a first step, we need to identify the candidate programming models, which we will use for implementation in our case study.

For parallel programming and the integration of dedicated acceleration, a number of programming models and standards already exist, mostly originating from the high-performance computing domain. As `C` and `C++` are the dominant programming languages in the automotive domain at this point, having almost 50% share [2], we will focus on programming models that are based upon at least one of these languages. Beyond that, we further tighten that focus to well-established programming languages with an active community, to make sure sufficient training resources and experts are available.

After reviewing the parallel programming models currently enjoying the most prominence, we selected OpenMP [7], OpenCL [8] and CUDA [9] as candidate models. The three models cover a broad spectrum, ranging from the rather high-level abstractions of OpenMP to the very explicit parallelization and offloading of OpenCL.

B. Benchmark Selection

The beginning of the re-enacted migration process of an existing application to a parallel, heterogeneous platform usually is the serial implementation of an algorithm. As the central aim of this project is to investigate the applicability of the programming models to *automotive* software, we chose to use algorithms from the automotive domain and their corresponding serial implementations as starting points for our implementation.

After review, we selected the open-source DAPHNE benchmark suite [10] as the source for the serial implementations. The DAPHNE suite contains three automotive kernels, called points2image, euclidean_clustering and ndt_mapping, that were extracted from the Autoware autonomous driving framework [11]. In addition, the benchmark also provides datasets with input- and reference data captured during an actual drive, that we can use to ensure the correctness of our parallel implementations.

C. Selection and Bring-Up of Evaluation Platforms

For testing and performance evaluation of the benchmark implementations, suitable evaluation platforms are required. In the selection process of these platforms, our central aim was to cover a broad range of current embedded, parallel and heterogeneous platforms. After a review of available technologies as step three of our approach, three different platforms were acquired:

- Nvidia Jetson TX2
- Nvidia Jetson AGX Xavier
- Renesas R-Car V3M

All three selected platforms combine a multi-core CPU with a GPU (called *image recognition engine* in case of the V3M) and are designed for automotive usage scenarios. As such, they exhibit the particular characteristics regarding computational power and energy budget typically found on automotive platforms.

D. Benchmark implementation and porting.

The fourth step of our approach is the actual implementation process of the benchmarks that lies at the heart of our practical, quantitative approach.

In contrast to many other surveys (e.g., [3]), we explicitly do not consider the time required to *learn* a parallel programming model here. In our implementation case study, the developers performing the implementations are already experts with multi-year experience with the respective programming models. This is similar to a real-world scenario, where companies are likely to hire developers that are familiar with programming models and have prior experience in their use.

During the implementation, the original serial code is parallelized using the means provided by the respective programming model. Additionally, the compute-intensive parts of the application are offloaded onto the parallel accelerators, i.e., the GPU, if the programming model allows to do so.

This implementation flow replicates the typical process of migrating an existing, serial code base to a new parallel, heterogeneous platform. Beyond that, many of our insights should also be applicable to the development process of new software from scratch, i.e., without a pre-existing serial implementation.

Once an application has been parallelized, it should be deployable to multiple *different* heterogeneous compute platforms, therefore *portability* plays a major role for the applicability of a programming model for the automotive domain.

To assess the portability of the selected programming models and the resulting development effort, we also re-enact the typical process of porting an application to a different platform. To this end, the resulting parallel implementation of a benchmark targeting an initial platform is also evaluated and optimized at least on a second one.

While our practical approach allows us to investigate the programming models in a real-world scenario, it does have a number of limitations that might make it less suitable for other purposes.

- The kernels were selected to study the parallelization effort for different parallel paradigms/platforms. They represent some automotive workloads, but not *all* automotive workloads.
- With Autoware's roots in fundamental academic research, their implementation is not necessarily performance optimized. Similarly, the highly modular ROS-based structure does carry a performance overhead, as it is very difficult (or not even possible at all) to optimize data transfers between host and accelerator memories across ROS node boundaries.

IV. Evaluation

The raw performance of the parallel implementations is *not* the key aspect of the programming models we want to investigate in this case study. But even so, achievable performance plays a crucial role when judging a parallel programming model, and can therefore not be completely neglected in this case study.

However, for the business decision on which programming model to use for the implementation on heterogeneous platforms, the following three non-functional aspects of programming models need to be considered as well:

- Programmer Productivity
- Maintainability
- Portability

All three aspects of programming models listed above are "soft" characteristics, i.e. they cannot be measured directly. Rather, one needs to quantitatively assess them indirectly through a combination of multiple metrics. To this end, we have assembled a set of measurements and metrics described in the following. After the definition of our metrics, we will investigate each of the listed aspects in Sections IV-A to IV-C.

a) Programmer Productivity: The productivity a developer achieves using a given programming model gives insights into the ease-of-use of the model.

We use a simultaneous tracking of working hours vs. achieved performance to determine which programming model yields the required performance with the least development effort. For many applications, a performance *lower* than the maximum achievable performance on a given platform is absolutely sufficient, e.g., to meet real-time requirements. In such a case, a programming model that achieves the *required* performance faster than the other models, even though this model may not be able to deliver the best peak performance, is preferable.

In our case study, the developers measure performance roughly every sixty minutes, resulting in graphs similar to the ones shown in Fig. 1.

b) Maintainability: Application software in the automotive field typically has a relatively long lifetime, potentially extending to over more than a decade. Thus, good maintainability is indispensable. The effort required for the maintenance of a piece of code is dominated by the time that a developer, who is not the original author of the code, needs to become familiar with the code base in order to make the desired changes.

The maintenance effort is influenced by the code volume and the complexity of the code. To assess the impact of parallel programming models on the code volume, we measure the number of changed lines compared to the original serial version of the code. In contrast to prior work [3], we also consider *in-place* changes, because many parallel programming models also require to restructure the original code of the application.

Assessing the complexity added to the code base due to the use of a parallel programming model is more complicated. Classical software complexity metrics such as the ones proposed by McCabe [12] or Halstead [13] are tailored towards control-flow heavy business software, and are not suitable for this purpose. We therefore developed a new metric, the *Complexity Count*. The reasoning behind the complexity count, is that complexity introduced by parallel programming models stems from the inclusion of new keywords, new datatypes, runtime function calls and compiler directives defined by the programming model into the code of an application. To calculate the complexity count, we simply count the number of these additional programming model constructs in the code base, and also the number of parameters passed to these constructs, e.g., to runtime functions. For example, the use of

```
cudaMallocManaged(&a, vector_size * sizeof(float));
```

in the code would yield a complexity count of three.

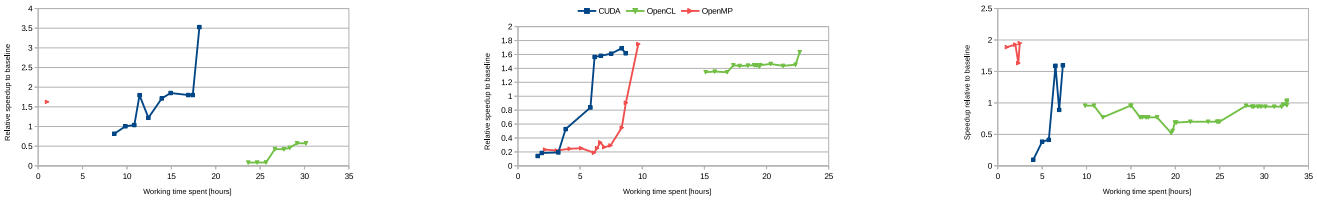
c) Portability: Once a code base has been parallelized and partially offloaded to dedicated accelerators, it should ideally be usable for multiple *different* heterogeneous platforms. The characteristics of a programming model (e.g., high-level abstractions, compiler directives, etc.) can directly influence the portability. It is thus important to assess the porting effort required for each of the selected models.

Besides making the existing code compile, and compute correctly on the new platform, porting typically involves a process of incremental improvements to optimize performance on a new platform. The duration of this process indirectly provides information about the portability characteristic of a programming model. We use a simultaneous tracking of working hours spent on porting an existing implementation vs. the performance on the new platform, similar to the one we employed to measure the programmer productivity.

A. Programmer Productivity

While the three programming techniques employed in the study have different effort vs. performance curves in Fig. 1, a trend is clear across the benchmark kernels.

a) OpenMP Implementation: Across all three benchmarks investigated in our implementation case study, OpenMP typically requires the least effort to parallelize an application.



(a) Benchmark points2image. (b) Benchmark euclidean_clustering. (c) Benchmark ndt_mapping.

Fig. 1: Result of simultaneous tracking of working hours vs. speedup over serial baseline to assess programmer productivity.

For example, the parallelization of the points2image benchmark in Fig. 1a takes only a single hour of development effort. In general, OpenMP, mainly based on compiler directives, benefits from the fact that it typically requires less invasive restructuring for parallelization than other programming models. This also implies that the performance of the application can be assessed throughout the development cycle, which can also be a big plus for development productivity.

b) CUDA Implementation: CUDA typically also allows for fast parallelization. Once a parallelization approach is determined, it can often be realized in just a few hours for kernels with the complexity of our benchmarks (e.g. Fig. 1b and Fig. 1c). Performance-wise, OpenMP and CUDA are mostly comparable, the lower number of threads available on the CPU (note that we focus on the CPU-based features of OpenMP here!) and the overhead of offloading computation and data to the more powerful GPU often cancel each other out.

c) OpenCL Implementation: For the majority of the benchmarks, OpenCL requires much up-front work to restructure and partition the application, resulting in a phase where performance cannot be assessed to determine the prospects of success for the chosen parallelization strategy. In Fig. 1a and Fig. 1b, this is indicated by the late start of the green curve for OpenCL. The relatively complex host code for OpenCL, and the invasive changes to the serial implementation, also cause OpenCL to often require the most effort for parallel and heterogeneous implementation.

B. Maintainability

The ranking with regard to the required development effort also correlates with the results that we get from our metrics for *maintainability*. The evaluation of the total number of line changes (added or deleted) in relation to the LoC of the original, serial implementation is given in Fig. 2.

Because OpenMP allows to reuse the serial implementation almost without changes in most cases, and only requires to add the description of parallel semantics through compiler directives, the number of changes is relatively small (3%-17%).

In contrast, CUDA requires kernel functionality to be extracted to dedicated device functions and the inclusion of additional API calls into the host code, resulting in a significantly higher number of changes (24% to 80%).

For OpenCL, the extraction of device code to separate files and the inclusion of even more boilerplate code into the host

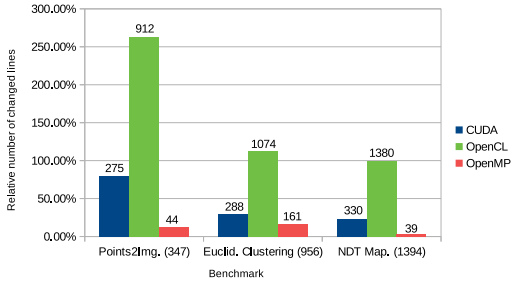


Fig. 2: Number of changed lines relative to serial baseline. Numbers in parentheses give LoC of serial implementation.

leads to sweeping changes in the code base, ranging from 99% to 263%.

To assess the additional complexity introduced by the use of parallel programming models into an application's code, we use the *Complexity Count*. The counts for all benchmarks and models are given in Table I.

Benchmark	CUDA	OpenCL	OpenMP
points2image	70	329	12
euclidean_clustering	17	120	15
ndt_mapping	64	113	28

TABLE I: Complexity count.

Because the OpenMP compiler directives add parallel semantics in a descriptive/prescriptive manner and operate on a high level of abstraction, the complexity added by new keywords and directives is very low.

For CUDA, the added complexity has two main sources: New data-types and keywords are added on top of the `C++` programming language, mainly to partition the application between host and device. Additionally, a number of API functions has to be called in order to transfer data and execution to the device. Nevertheless, the complexity added is still moderate.

In OpenCL, the sources of complexity are similar to CUDA, namely new data-types and keywords for the device section and API calls in the host code. However, OpenCL requires much more host code than CUDA, resulting in significantly higher complexity counts.

For OpenCL, there is also a considerable difference between the use of the traditional `C`-API and the `C++` wrapper API: While the points2image benchmark was implemented with the

C-API, the other two benchmarks use the C++ wrapper API for the host code. Using the latter, some steps of the host-side setup process are abstracted, resulting in a notably smaller complexity count.

C. Portability

Similar to the discussion of development effort vs. performance, we can also see a trend of using the three different programming methods in terms of their portability.

The high-level of abstraction supported by OpenMP also benefits *portability*. Moving an existing, parallel OpenMP implementation to another platform typically boils down to a simple re-compilation on the new platform, taking less than 20 minutes to complete for each of our benchmarks.

For CUDA, the situation is similar. When moving from one platform to another, the code usually does not need to be changed, thanks to the standardization of the CUDA language by Nvidia for all its devices, and only a small number of parameters needs to be tuned.

With OpenCL, things are different. Basic features, such as support for double-precision floating-point arithmetic are only *optional* features, and different vendors typically support different versions of the OpenCL specification. To these they might add extensions that only work on platforms manufactured by this vendor.

For two of our three benchmarks, this required manual changes that often took hours. For example, adapting the points2image benchmark for the Renesas V3M platform required code changes to use only single precision floating point computations, instead of the double precision of the original code. This required almost 12 hours of development time.

V. Conclusion

In this study, we have taken a very practical approach to evaluate the applicability of today's parallel programming models in the automotive domain. We considered both the nature of typical automotive compute kernels, which are often very short compared to HPC kernels, and the constraints of actual embedded hardware platforms.

Based on our insights, we cannot declare a single "winning" programming model here. However, our experiences should serve as a first indicator for the applicability of different programming models, and show a way forward to future development and research.

The high-level abstractions defined by the **OpenMP** standard allowed for a very good programmer productivity. For the actual parallelization, OpenMP relies on the compiler, which yielded competitive performance for our benchmarks. However, we were yet not able to use the device offloading features recently added to the standard due to insufficient compiler support on the target platforms. Future research should investigate the possibility to extend the compiler support for OpenMP to such target platforms and workloads in more depth (e.g., use OpenMP to target FPGAs [14], [15]).

CUDA strikes a balance between high-level abstractions and explicit parallelization. In combination, this allows reasonable programmer productivity and good performance. However, beyond the official, proprietary compilers and runtimes from Nvidia, no competitive open implementations for CUDA exist. Thus, the use of CUDA carries the risk of vendor lock-in. Alternatives for moving CUDA outside the Nvidia ecosystem (e.g., AMD HIP/ROCm [16]) are only slowly appearing and need further evaluation in future research.

In contrast to CUDA, implementations with **OpenCL** require much more host code, and far more invasive restructuring of the application. The partitioning into multiple files for host and device code causes a large up-front effort for implementation, before parallelization and optimization can even be started.

With **SYCL** as a spiritual successor to OpenCL, the Khronos Group provides a modern, open standard designed to overcome these limitations of OpenCL. As soons as SYCL becomes available on more embedded platforms, future research should investigate the use of SYCL for the implementation of automotive workloads on the corresponding target platforms.

More details on the evaluation and an investigation of FPGAs as accelerators is available in the technical report [17].

References

[1] Nvidia Inc., "Nvidia Drive platform," https://developer.nvidia.com/drive.

[2] Z. Molotnikov, K. Schorp, V. Aravantinos, and B. Schätz, "FAT-Schriftenreihe 287 - Future Programming Paradigms in the Automotive Industry," Tech. Rep., 2016.

[3] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov. 2005.

[4] L. Hochstein and V. R. Basili, "An empirical study to compare two parallel programming models," in *ACM SPAA*, 2006.

[5] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate gpu performance estimation through source-level analysis and trace-based simulation," in *HPCA*, Feb 2019.

[6] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud, "Optimal performance prediction of adas algorithms on embedded parallel architectures," in *2015 Intl. Conf. on High Performance Computing and Communications*, Aug 2015, pp. 213–218.

[7] OpenMP Architecture Review Board., "OpenMP Application Programming Interface - OpenMP Standard 5.0," Nov. 2018.

[8] "OpenCL Specification," https://www.khronos.org/opencl, 2019.

[9] "CUDA C programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2018, accessed: 2019-02-05.

[10] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, "Wip: Automotive benchmark suite for parallel programming models on embedded heterogeneous platforms," in *Proceedings of the International Conference on Embedded Software*, ser. EMSOFT '19, 2019.

[11] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, Nov 2015.

[12] T. J. McCabe, "A complexity measure," *Transactions on Software Engineering*, 1976.

[13] M. H. Halstead *et al.*, *Elements of software science*, 1977.

[14] L. Sommer, J. Korinth, and A. Koch, "Openmp Device Offloading to FPGA Accelerators," in *2017 IEEE 28th Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[15] L. Sommer, J. Oppermann, J. Hofmann, and A. Koch, "Synthesis of Interleaved Multithreaded Accelerators from openmp Loops," in *2017 Intl. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*.

[16] Advanced Microdevices, Inc., "HIP," https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/, 2019.

[17] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, "FAT-Schriftenreihe 317 - EPHoS: Evaluation of Programming-Models for Heterogeneous Systems," Tech. Rep., 2019. [Online]. Available: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-317.html