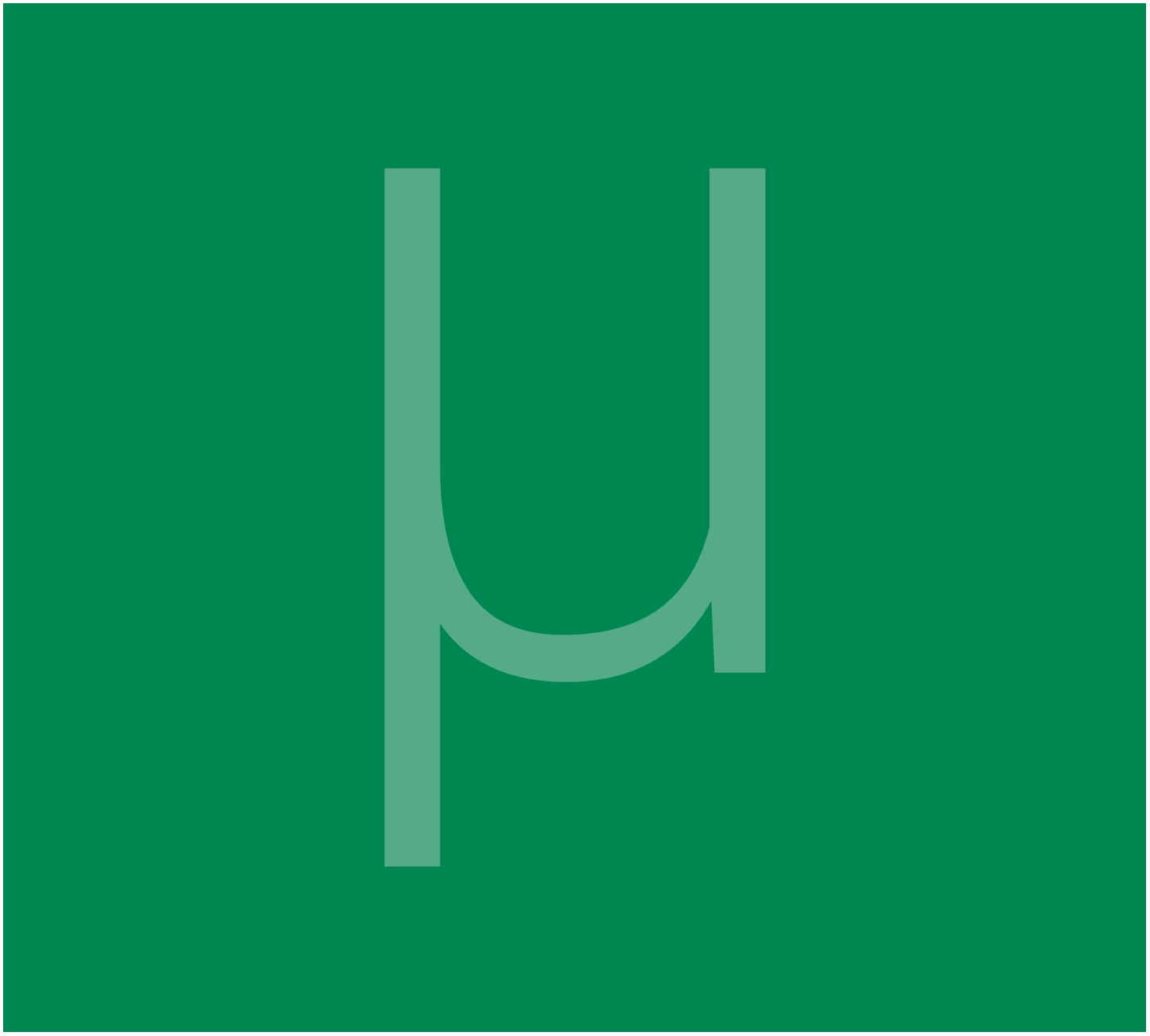
VDA | Verband der Automobilindustrie

FAT | Forschungsvereinigung Automobiltechnik

FAT-Schriftenreihe 317

EPHoS: Evaluation of Programming Models for Heterogeneous Systems



EPHoS: Evaluation of Programming Models for Heterogeneous Systems

Technische Universität Darmstadt
Embedded Systems and Applications Group (ESA)

Lukas Sommer
Florian Stock
Leonardo Solis-Vasquez
Andreas Koch

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e.V. (FAT) gefördert.

Abstract

In recent years, there has been a steep increase in the computational demands of automotive applications, in particular in the field of assisted and autonomous driving. With the slowing down of Moore's Law, these computational requirements can no longer be satisfied by classical, embedded single-core processors. Therefore, automotive manufacturers, suppliers, and original equipment manufacturers (OEM) turn towards parallel and heterogeneous platforms, combining multi-core CPUs and dedicated accelerators on a single board or SoC to satisfy their application's compute demands.
As the programming languages traditionally used in this industry, specifically `C` and `C++`, do not provide abstraction mechanisms to leverage the parallelism in such heterogeneous platforms, new, parallel programming models need to be considered for adoption in the industry.
In this work, we evaluate the applicability of parallel programming models established, e.g., in high-performance computing, for embedded use in the automotive domain, targeting parallel and heterogeneous embedded computing platforms. To this end, we conduct a practical implementation survey with benchmarks from the autonomous driving domain, targeting representative execution platforms, and evaluate important usability aspects of the investigated programming models. Our aim is to identify desirable characteristics and recommend possible candidate models for wider adoption in the industry.

Contents

1 Introduction and Motivation

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, e.g., due to soft or hard real-time constraints, the automotive industry faces challenges similar to those encountered by the high-performance computing (HPC) community about a decade ago. With the slowing down of Moore's Law, vendors can no longer rely on improved single-core performance through improvements in manufacturing technology for sufficient performance gains to keep up with application needs. Cooling issues imposed by physical limitations make it difficult and costly to further scale-up the operating frequency of single-core processors.

In reaction to these challenges, the HPC community turned towards parallel and heterogeneous platforms. By spreading out the solution of computational problems across multiple processing cores, significant performance gains could be achieved for algorithms with sufficient potential for parallelization. Even greater improvements, regarding performance as well as energy efficiency, can be achieved through the use of specialized accelerators. Even though the implementation of custom chips for every computational task is not feasible, programmable and configurable accelerators can provide significant gains in performance and energy consumption. The most prominent example of such accelerators, which are typically coupled with a host CPU in a heterogeneous system, are graphics processing units (GPU), but other accelerator technologies such as field-programmable gate arrays (FPGA) and digital signal processors (DSP), also receive increasing attention nowadays. The automotive industry has also started to adopt such technologies through collaborations with the respective vendors [22, 6, 37].

The increasing adoption of parallel, heterogeneous platforms presents new challenges to automotive manufacturers, in particular with regard to the *programming* of these platforms. C and C++, the currently dominating programming languages in the automotive field [21] do not provide sufficient mechanisms for the development on parallel and heterogeneous platforms. Both languages offer only library support for threads as the sole means to program multi-core processors, but programming with explicit handling of threads (e.g., at the pthreads level) requires enormous amounts of work and

is an error-prone task. Language support for offloading computations to specialized accelerators is completely lacking from both languages.

Therefore, the automotive industry needs to break new ground and adopt new programming models that facilitate the productive development for the parallel, heterogeneous platforms employed in future automotive vehicles. Of course, there is no need to reinvent the wheel here: many of the challenges linked to the programming of parallel and heterogeneous platforms have already been addressed by the HPC community, resulting in a number of established or even standardized programming models actively maintained by open-source communities and/or vendors. Although these programming models present solid fundamentals for the parallelization of applications and the offloading of compute-intensive parts to accelerators, they are not usable "out-of-the-box" for automotive platforms. Target platforms in the automotive domain differ significantly from HPC systems, e.g., with regard to the power and thermal budget. The programming models established in the HPC community can therefore serve as a basis, but might required adaptation for the automotive field.

Consequently, the aim of this project is to investigate programming models established in the HPC field with regard to their applicability in automotive applications. The intent of this survey is to provide insights about how well established programming models are suited for use in automotive applications and how they could be improved and extended for automotive use.

The focus of our investigation is on the usability of the examined programming models. Programmer productivity is a key aspect of programming models. Good abstractions, making the composition of the targeted heterogeneous platform transparent for the programmer and hiding unnecessary hardware details from the programmer, are critical for a productive use of a programming model.

At the same time, the abstractions should not affect the efficiency with regard to performance. In addition to programmer productivity, maintainability and portability are critical aspects of programming models: throughout an application's lifetime, which can extend to years, or even decades in the automotive field, the software has to be maintained and should be portable on other heterogeneous platforms.

While we will investigate the performance of our implementations on target platforms representative for the high-performance embedded parallel computing scenarios in advanced automotive applications, that aspect is only peripherally relevant to this study. Due to our use of real, but completely generic (non-platform optimized) algorithms extracted from the open-source

Autoware autonomous driving framework (discussed in Section 4.1.1), as well as the highly diverse target hardware platforms, the raw performance numbers are *not* comparable. They are indicative *neither* of the performance of the platforms, *nor* of the programming tools and should not be interpreted as such.

In order to investigate the usability aspects of programming models and characteristics closely related to them, we pursue a practical, quantitative process: in our survey, we want to reproduce a realistic implementation scenario in the automotive field as closely as possible. We implement a set of typical benchmark algorithms on parallel, heterogeneous platforms employed in today's and future automotive vehicles, thereby putting the programming models to test for their intended use in the automotive field.

Note that, in contrast to more academic approaches, we have *not* cherry-picked our sample algorithms for maximum speed-up potential on the target platforms. Instead, the key criterion for inclusion in our benchmark suite was the total amount of compute time spent in a specific algorithm during real-world use. This actually results in a number of benchmarks that are difficult to execute efficiently with current heterogeneous platforms and their programming tools, e.g., due to having only very short per-invocation compute times. Our findings have already led to some programming tools being improved to better cope with these very relevant, but awkward to execute kernels.

More details on our approach can be found in Chapter 2. In Chapter 3 we introduce the programming models investigated in this survey and Chapter 4 gives details on the benchmark applications we used to evaluate them. The platforms targeted in our implementation survey are presented in Chapter 5. Insights that we gained about the different programming models are given in Chapter 6, and in Chapter 7 we present our quantitative evaluation. Chapter 8 concludes this report and gives an outlook to future developments in this field.

2 Design of the Implementation Study

As indicated in the previous chapter, we want to take a practical approach to evaluating the applicability of different candidate parallel programming models. In this chapter, we first describe the scope of this study, and then proceed to discuss our approach, the metrics employed for evaluation, and potential inaccuracies in our findings.

2.1 Aim and Scope

As pointed out in the previous chapter, the use of heterogeneous, parallel compute architectures will be inevitable for future automotive vehicles. As a consequence, there will be a need for suitable programming models that allow to program such architectures in a efficient, effective, and performant way. Therefore, the aim of this implementation survey is the investigation of existing and well-established programming models with regard to their suitability for heterogeneous, parallel platforms in the automotive sector.

Historically, most of the candidate programming models have been designed for the use in high-performance computing (HPC). The platforms used in HPC-scenarios differ significantly from the (embedded) compute architectures built into modern and future cars.

Typically, HPC computers have a much higher power and thermal budget available, and their per-unit costs are much higher than those for embedded platforms. The design of embedded processors (often also called *electronic control unit*, ECU) is almost solely based on system-on-chip architectures. These have the different components, and in the context of this survey, the different processing elements, all integrated into a single chip. Whereas in the HPC domain, accelerators (e.g., GPUs) are typically deployed as dedicated discrete accelerator cards, with a much looser connection to the main CPU (e.g., PCI Express ot NVLink). It can therefore be expected that none of the existing programming models, which were designed for HPC platforms, will be perfectly suitable for out-of-the-box usage in embedded scenarios, Instead, they will need adaptation for this usage scenario.

Still, we believe that we can gain many interesting and valuable insights from a practical investigation of existing programming models. Programming models are often build from a small number of fundamental abstractions. In consequence, many programming models will share at least a few (and in practice even many) common characteristics.

Even if we are not able to find a universal "silver bullet" to cope with all problems arising in heterogeneous, parallel programming, we can still identify a set of desirable characteristics that one would want to see in a parallel, heterogeneous programming model.

As such, the insights gained about certain programming features, exposed by a given programming model, can be leveraged in other models. Moreover, this approach also allows to assess *future* programming models, which might note even have been invented at this point.

Although the assessment of parallel programming models often boils down to the evaluation of their respective performance, raw performance is *not* the central focus of this survey. Programming models naturally have an influence on the achievable performance of applications implemented by them. This is due to many factors, most commonly the cost of executing certain programming model abstractions on concrete hardware architectures.

A programming model that fails to sufficiently capture the parallel semantics of a program will not be able to deliver the desired performance. Therefore we cannot fully neglect performance in our survey. However, a performant, but barely usable programming model (e.g., raw pthreads) is only of limited use. For this reason, the focus of this survey is on the *usability* and *applicability* of the programming models throughout the whole *lifecycle* of automotive software, from initial development to maintenance during product life-time.

2.2 Approach

The central aim of this survey 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. [21]), 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, heterogenous platform. With the continued integration of such platforms into automotive vehicles, many OEMs and component suppliers will be confronted with this task.

Our approach does not only cover the applicability of the programming models themselves, but implicitly also the availability of corresponding tools and ecosystems supporting them. For this survey we have chosen an approach comprising the following five steps:

1. **Identification of relevant programming models.** For parallel programming and the integration of dedicated acceleration, a number of programming models and standards already exists, mostly originating from the high-performance computing domain. As a comprehensive investigation of *all* existent programming models is not feasible, we need to identify a small number of promising and representative candidate models in a first step.

   The survey conducted by Molotnikov et al. [21] showed that with almost a 50% share, C and C++ are the dominant programming languages in the automotive domain at this point. Thus 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.

   To identify the most promising candidates, we reviewed the parallel programming models currently enjoying the most prominence according to the criteria. As a result, we identified three representative candidates to use in this study. More details on the selection process and criteria can be found in Chapter 3.

2. **Benchmark selection and extraction.** The beginning of the re-enacted migration process of an existing application to a parallel, heterogeneous platform 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. To this end, we reviewed multiple open-source projects from the automotive domain, in particular from the AD/ADAS fields. We found Autoware [16] to be the most promising source of reference implementations, and identified three different benchmark kernels in it. For ease of development, testing, and debugging, we extracted each of the kernels to a standalone benchmark and developed a validation suite for each one. More details on the kernel identification, extraction process, as well as each of the benchmark kernels can be found in Chapter 4.

3. **Selection and bring-up of evaluation platforms.** For testing and performance evaluation of the benchmark implementations, created with the different programming models in **Step 2** of our approach, 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. The platforms should come from different vendors, resulting in a large cross-section of technologies, software stacks, and tool ecosystems. After a review of available technologies, four different platforms were acquired:

- Nvidia Jetson TX2
- Nvidia Jetson AGX Xavier
- Renesas R-Car V3M
- Xilinx Zynq Ultrascale+ MPSoC ZCU102

While the first three of these platforms are "classical" heterogeneous platforms in the sense that they combine a multi-core CPU with a GPU, the last platform combines a multi-core CPU with a field-programmable gate array (FPGA), a technology that is relatively new to the automotive domain, but attracts increasing attention[6, 37]. The programming of FPGA-based accelerators is very different from that for common software-based architectures, therefore we expect to gain some interesting insights on the applicability of FPGA-based accelerators in the automotive domain.

More details on all evaluation platforms and the bring-up process can be found in Chapter 5.

4. **Benchmark implementation and porting.** With the evaluation platforms up and running, along with the relevant kernels extracted into standalone benchmarks, we can start the implementation process that lies at the heart of our practical, quantitative approach.

   In contrast to many other surveys (e.g., [12]) which investigated parallel programming models, we explicitly do not consider the time required to *learn* a parallel programming model here. In our implementation survey, 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.

   The implementation for each of the programming models always starts from the original serial code, which was extracted into a standalone benchmark in **Step 2**. During the implementation, the algorithm 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 (e.g., GPU, FPGA) available in the heterogeneous system, 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. The insights about each of the programming models are transferable to real-world migration scenarios. 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.

Typically, an automotive manufacturer or supplier will employ different compute platforms in his product line. Thus, once an application has been parallelized, it should be deployable to multiple *different* heterogeneous compute platforms, potentially from different vendors and with a varying composition of processing units and accelerators. 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. We expect to gain important insights about the portability of the programming models from this process. Details on the insights gained from our re-enacted development process, as well as lessons learned for each programming model, can be found in Chapter 6.

5. **Evaluation.** Although many features of programming models, such as the portability or ease-of-use, cannot be measured directly, they can be quantified *indirectly*, e.g., by measuring the required development time for a benchmark implementation. Therefore, we have designed a set of metrics that allow us to substantiate the insights gained during the re-enacted implementation process with measurements. These metrics will be the subject of the next section.

2.3 Evaluation and Metrics

As already indicated before, the mere performance of the parallel implementations is not the key aspect of the programming models we are interested in. But even so, achievable performance plays a crucial role when judging a

parallel programming model, and can therefore not be neglected completely in this survey.
However, for the business decision on which programming model to use for the implementation on heterogeneous platforms, other aspects of programming models are at least as important. A company will always be interested in spending the expensive resource "developer time" as efficiently as possible, and the choice of the most appropriate programming model for the task at hand has a major impact on the efficiency of the developers. Therefore, the following three aspects of programming models need to be considered:

- **Programmer productivity.** The productivity a developer achieves using a given programming model gives insights into the ease-of-use of the model. Assuming that the developer is already familiar with the programming model and has some prior experience in using it, we can begin assess the applicability of a model. We do this by considering the effort required to parallelize an application and offload critical sections of the algorithm with the aim to achieve a certain level of application performance. The ease-of-use and thereby the programmer productivity is directly influenced by the different characteristics and features of a programming model. For example, the ability of a programming model to support step-by-step (incremental) parallelization, as is possible in OpenMP, probably improves the ease-of-use.

- **Maintainability.** Application software in the automotive field typically has a rather long life-time, potentially extending over more than a decade, and partially conditioned by legal requirements. Thus, good maintainability is indispensable. Besides the mere volume of the code, the code complexity is an important factor for maintainability. Often, the complexity of code for an application is directly influenced by the choice of programming model for the implementation, as parallel heterogeneous programming models often require additional runtime function calls or compiler directives. In some cases, even modifications of the algorithm might be required to make it fit the programming model.

- **Portability.** Once a code base has been parallelized and partially offloaded to dedicated accelerators, it should be usable for multiple different heterogeneous platforms. These might have varying compositions of processing units and accelerators. While the effort required to make an existing parallel implementation usable on another platform should

ideally be small, there might be cases where porting requires large code re-arrangements. 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.

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 detail in the following paragraphs.

Programmer productivity. Fundamentally, programmer productivity can be measured by recording the number of working hours spent by a developer on an implementation. This metric is a much simplified, but already useful indicator for the programmer productivity enabled by a given programming model.

However, for the comparison of programming models conducted in this survey, we extended the simple tracking of working hours with also recording the performance achieved at the corresponding time intervals.

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 deliver the best peak performance, is preferable. It enables developers to meet the requirements faster and therefore allows for a higher programmer productivity.

Our simultaneous tracking of working hours vs. achieved performance allows to determine which programming model yields the required performance with the least development effort. In our survey, the developers measure performance each time they have spent roughly sixty minutes on the implementation with one of the programming models, resulting in graphs like the one that can be found in Figure 2.1.

The horizontal green bar indicates the required performance, as dictated, e.g., by real-time requirements. For each of the programming models, a line depicts the progress of the implementation process, with the performance being shown as speedup relative to the original serial implementation.

In our hypothetical scenario, the implementation using OpenMP first causes a slight decrease in performance, e.g., due to algorithmic changes. After that,

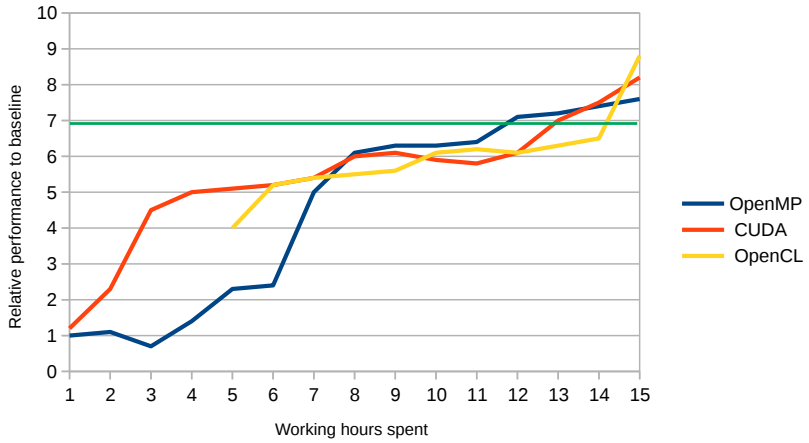


Figure 2.1.: Sample graph showing the simultaneous tracking of working hours vs performance for a hypothetical application

the incremental parallelization improves performance step by step, until the required performance target is met after twelve hours of development.

For CUDA, we initially observe a steep increase in performance. After that, performances increases steadily at a lower rate than at the beginning. The required performance target is finally met after thirteen hours of development.

The line for OpenCL only starts after five hours of development, indicating that the corresponding OpenCL implementation could not be compiled, or yielded incorrect results before that time. Afterwards, performance increases step by step and eventually meets the required performance target after fourteen hours of development effort.

Even though OpenCL finally yields the *best performance* in this hypothetical example, the OpenMP programmer was the *first* to meet the performance requirements. Therefore, in this case OpenMP would be preferred over CUDA and OpenCL, as it allows to meet the required performance with the least effort. We interpret this to indicate that OpenMP is enabling a higher programmer productivity for the hypothetical application.

Maintainability. The maintainability of code written using the various programming models can be quantified using a combination of three different metrics. 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 get familiar with the code base in order to make the desired changes. Therefore, all three metrics we use to assess the maintainability of programming models try to give an estimate of the effort required to understand a parallelized implementation.

- The first metric we employ to assess maintainability is the traditional lines-of-code (LoC) metric, which provides information about the raw size of the code. Although this metric is of course strongly dependent on the complexity of the implemented algorithm itself, the comparison of the LoC metric values between different programming models can be used as a first indicator on their respective maintainability. This comparison is similar to the *code expansion factor* used in prior work [12].

- However, the raw code volume alone is not a sufficient indicator for maintainability. Imagine a scenario where one model would add a small number of compiler directives to the serial code, leading to an increase in code size. In that scenario, another programming model could lead to the overall code size decreasing, but replacing large fractions of the original serial code with complicated, specialized code. In this the scenario, the code of the first programming model would generally considered to be more maintainable, as the model-specific modifications to the original serial code are much more limited. To reflect this aspect of maintainability, we use the extent of the changes compared to the original serial implementation as a second indicator for maintainability. We compute the volume of changes by using `git diff`, usually between the code in a given parallel heterogeneous programming model and the original code of the serial baseline.

- While the two metrics defined for maintainability so far provide information about the complexity added in size by the programming models, they do not serve as an indicator for the complexity of the models themselves. Traditional metrics for code complexity, such as those proposed by Halstead [9] or McCabe [20], also do not serve our purpose, because these metrics are tailored towards control-flow heavy, "classical" business software, and do not reflect the complexity of our programming models.

  Therefore we decided to define a new metric to assess the complexity added by the use of a parallel programming model on top of the original serial code. Almost all of the programming models reviewed during the selection process (cf. Section 2.2, Chapter 3) extend a base language, such as C or C++, in one or multiple of the following ways.

  - Runtime function calls used to control the parallel execution or use of dedicated accelerators in heterogeneous systems.

- New keywords to make parallel or heterogeneous features available in the base language or add new types.
- Compiler directives guiding or instructing the compiler in its effort to parallelize the code and offload critical sections to dedicated accelerators.

The complexity introduced into a code base by a programming model is determined by counting the instances of *language extensions* used for parallelization and integration of accelerators.

Beyond that, additional complexity introduced by the *parameters* to the language extensions. For example, in case of a runtime function called from the code, a developer trying to get familiar with that code needs to understand the purpose of each individual parameter, as well as the reason for which a specific value of the parameter was chosen. Therefore, we define our third metric as the count of both distinct language extensions as well as values/variables appearing in combination with these extensions. Listing 2 and Listing 1 demonstrate how this metric is calculated.

```
1 dim3 threaddim(THREADS);
2 dim3 blockdim((vector_size+THREADS-1)/THREADS);
3 cudaMallocManaged(&a, vector_size * sizeof(float));
4 vector_add<<<blockdim, threaddim>>>(a, b, c);
5 cudaDeviceSynchronize();
```

Listing 1: CUDA example snippet (simplified).

Listing 1 shows a short CUDA code excerpt. In this example, we count two new keywords/types (`dim3`, `<< ... >>`), two runtime functions (`cudaMallocManaged`, `cudaDeviceSynchronize`) and eight parameters (`threaddim`, `THREADS`, `blockdim`, `vector_size`, `a`, `b`, `c`, `vector_add`), resulting in a *complexity count* of twelve.

```
1   #pragma omp parallel for default(none) shared(a) reduction(+:sum)
2   for (i=0;i<n;i++)
3   sum=sum+a[i];
```

Listing 2: OpenMP example snippet.

For the short OpenMP example in Listing 2 we count seven compiler directives and clauses (`omp`, `parallel`, `for`, `default`, `none`, `shared`, `reduction`), as well as three parameters (`a`, `+`, `sum`), adding up to a complexity count of ten.

Portability. When discussing portability, we need to distinguish between *functional* porting and *performance* porting. While the former cares only about making a existing implementation compile and compute correctly on a new platform, the latter also considers the potentially necessary changes to the existing parallel implementation required to achieve good performance on the new platform.

A first metric to assess the effort for porting an existing code base when going from one platform to another is the extent of the changes required. Similarly to what we have defined in the previous paragraph, we can use the difference between two implementations on different platforms, computed by a tool such as `git diff`, as a first indication for the porting effort.

Performance porting typically involves a process of incremental improvements to optimize performance on a new platform. The length 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 existent implementation vs. the performance on the new platform, similar to the one we employed to measure the programmer productivity. In Chapter 7, we use the metrics defined in this section to evaluate our implementation survey.

2.4 Limitations and Potential Threats to Validity

In this section we discuss the limitations of our survey, as well as potential caveats to our findings that might restrict their general applicability.

Time and budget constraints within the project force us to limit our practical implementation survey to a rather small number of benchmarks (cf. Chapter 4). This small number of benchmarks yields a small data base for our evaluation. In consequence, not all findings from our evaluation might hold true for *all* automotive applications, in particular as all of our benchmarks originate from the rather narrow subfield of AD/ADAS.

Another point to which we must pay close attention is the baseline performance of our benchmarks. The process of extracting the kernels from Autoware also involves the replacement of external library functionality with custom code, enabling stand-alone execution. Thus, we need to make sure that our initial, serial benchmark implementations are not artificially slowed. In order to demonstrate that our baseline numbers match the real-world

performance of the original kernels within the Autoware framework, we compare the performance of our extracted benchmarks to the original code in Chapter 4.

The last source of inaccuracy is the "human factor". All of the developers in our practical implementation survey are experts in their respective programming model. Although all of them have undergone a similar professional education, we cannot fully eliminate small proficiency differences. Furthermore, the developer in charge of the CUDA implementation was also responsible for the extraction of the benchmark kernels from their original code base. His prior knowledge of the algorithms might have given him a small advantage over the other developers. We have sought to reduce this disparity in developer knowledge by holding regular meetings, in which all developers are briefed about the benchmark algorithms' function and structure.

3 Landscape of Parallel Heterogeneous Programming Models

Almost all parallel programming models available today have their origins in the high-performance computing (HPC) field, where both parallel programming and the use of dedicated accelerators have been well established for more than a decade.
In the HPC field, programming models are used mainly for two purposes:

- Distributing a computational task across the nodes in a cluster to simultaneously compute on multiple nodes. A representative model for this usage is MPI.
- Parallelizing the computation on a single node, e.g., by making use of multiple cores in the central processing unit(s). A well-established tool often used for this purpose is OpenMP.

Virtually all established programming models support only one kind of parallelization. If both kinds are to be used for a computational taks, typically a combination of two or more programming models is employed. In the HPC domain, that is typically the *hybrid* parallelization using MPI between nodes and OpenMP within each node.
Programming models for distributed computing such as MPI do not play a role in this survey. The HPC-style distribution of computation across multiple nodes is still uncommon for automotive embedded computing, something we do not expect to change in the near future.
Programming models for parallel processing and the use of accelerators are always a more or less unique mix of model-specific features combined with more fundamental ones that typically span models. Common to most of the parallel programming models is the use of an established serial language as a base. In many cases, this is C or C++. This base language is then extended by additional syntax elements and keywords, enriched by annotations and compiler directives, and complemented with a set of callable runtime functions.
Despite all commonalities, there are significant differences in the ways different parallel programming models express parallelism and heterogeneity. This ranges from relatively low-level models, such as OpenCL, where parallelism

has to be stated very explicitly, to models such as OpenMP or OpenACC that describe parallelization opportunities at a high level of abstraction.

For the less abstract models, the programmer is expected to be aware of low-level details of the execution platforms and needs to fully describe the parallelization strategy and structure. Often, the memory hierarchy is exposed to the developer, allowing (and sometimes requiring) to explicitly place data in the most appropriate location. The disadvantage of such models is the high up-front effort required for a parallelization, often requiring the rewrite of large fractions of the original code. On the other hand, the programmer has very detailed and immediate control over all parameters, which can be advantageous when optimizing an application.

In contrast, high-level programming models try to instruct or guide the compiler in its effort to parallelize and offload the application. This automated approach is based on providing parallel semantics and algorithmic information through compiler directives or annotations. These models place the burden of parallelization and offloading mainly on the compiler. Based on the hints provided by the programmer, the compiler is often able to make good parallelization choices. However, a disadvantage of these models is that missing or inappropriate hints may lead to poor parallelization quality. On the positive side, these high-level models enable parallelization with a low initial development effort, and also supporting incremental improvements without major code changes.

From the set of available parallel programming models, we want to select a set of three models that ideally should cover the full range of abstraction levels, and be representative for a larger set of programming models that share similar characteristics. As C and C++ are the most common programming languages in the automotive industry as of now [21], we restrict our search to models based on at least one of these languages. We further constrain our research to well-established programming models and disregard still exotic or not-yet-mature models such as Chapel or Legion. Widespread use of a model implies an active community, the availability of training resources, tool ecosystems, as well as a sufficient number of experts as potential employees.

After reviewing the available programming models, we selected OpenCL, OpenMP and CUDA for our practical implementation survey. In combination, these three models cover a wide range of abstraction levels, from a low-level, explicit notation (OpenCL) to a high-level, prescriptive/descriptive approach (OpenMP), with CUDA somewhere in between. Each of the selected models is representative for a set of other programming models with similar

characteristics, and together they cover many of the programming model features that can be found in parallel programming models today.
The following sections will each give a short overview of and introduction to the three selected programming models. We will look beyond these already established models in Chapter 8 and discuss some advances, including Domain-Specific Languages (DSLs) that have been proposed recently and which might play major roles in the future.

3.1 OpenMP

OpenMP (Open Multi-Processing) is a free and open standard [25] for shared-memory parallel programming, available for the programming languages C, C++ and Fortran. The OpenMP standard is maintained by the OpenMP Architecture Review Board (ARB), which recruits its member directly from the community and major vendors.
OpenMP started in 1997, driven by the need for a open parallel programming standard, portable across machines from different vendors. At that time, each vendor provided proprietary extensions for parallel programming that were often incompatible. Since then, OpenMP has become the most common model for shared-memory parallel programming in the HPC community. It is supported by virtually all major compilers and implementations exist from nearly all vendors as well as in open-source form. OpenMP is made up of three main components:

- **Compiler directives** to be inserted directly into the source code guide the compiler to parallelize the affected code. They provide parallel semantics not present in the base language (e.g. C++) and hints on how to distribute work across threads. Compiler directives are an easy-to-use mechanism to parallelize code without requiring invasive changes. They also allow to parallelize code incrementally, i.e. in a step-by-step approach.

- **Runtime library routines** allow an OpenMP user to directly interact with the underlying runtime for parallel execution, providing access to more detailed information and switches to influence the execution at a lower level than with directives.

- **Runtime implementations** actually realize the parallel execution as specified by the developer. They manage threads and data and control the parallel execution.

Historically, OpenMP only supported the parallelization of loops and code sections. Since that time, the OpenMP standard has been constantly evolving under the governance of the OpenMP ARB, driven by user demand and the need to support new architectures and systems. Today, the OpenMP compiler directives can be used to describe the worksharing loops, parallel tasks, and device offloading as the main mechanisms for parallelization.
We give an short introduction to each of these mechanisms in the following paragraphs. Next to these main constructs, OpenMP defines a number of other directives for different purposes, e.g., the `critical`-directive for thread synchronization and mutually exclusive processing.

Worksharing Loops. By adding the directive `omp for` inside a parallel region, or by using the combined construct `omp parallel for`, a sequential (for-)loop can be turned into a worksharing loop. If a thread encounters a worksharing loop, it creates a new team (if not already present) of subthreads and becomes the master of this team. The iterations of the loop are then distributed across the threads in the team. If required, the developer can also give hints on which scheduling algorithm to use. E.g., some workloads might run better with static scheduling, others could profit from dynamic scheduling. Additionally, the developer needs to make sure that all data is correctly classified as either being shared among threads, or being private to each thread, by using the corresponding data-sharing clauses.

```
1 int *a = ...
2 int *b = ...
3 int *c = ...
4
5 #pragma omp parallel for default(none) shared(a,b,c)
6 for(int i=0; i < SIZE; ++i){
7         c[i] = a[i] * b[i];
8 }
```

Listing 3: OpenMP worksharing loop example (simplified).

The snippet in Listing 3 presents an example for a worksharing loop. The arrays `a`, `b` and `c` are shared by all threads, the loop counter `i` is private for each thread by default. The iterations of the loop are distributed across the threads in the team, so each thread will compute a subset of the elements of `c`. Next to the clauses used in this example, the OpenMP standard defines a number of other clauses, e.g., `schedule` to set the schedule for the loop and `reduction` for reduction of values across threads (e.g., sum, min, max etc.). More details can be found in the OpenMP standard specification [25].

```
#pragma omp parallel
{
        ...
        #pragma omp single
        {
                for(int i = 0; i < CHUNKS; ++i){
                        #pragma omp task\
                                depend(out: read[i])
                        {
                                (void) read_input(...);
                        }

                        #pragma omp task\
                                depend(in: read[i]) \
                                depend(out: processing[i])
                        {
                                compute_result(...);
                        }

                        #pragma omp task\
                                depend(in: processing[i])
                        {
                                write_result(...);
                        }
                }
        }
}
```

Listing 4: OpenMP task example (simplified, adapted from [26]).

Parallel Tasks. Not all algorithms can be expressed in terms of loops that comply with the OpenMP definition of a *canonical* loop, and therefore cannot be parallelized using worksharing loops. However, many workloads could be parallelized by executing parts of the code as *concurrent tasks*. In order to support parallelization of such workloads in OpenMP, the `task` construct was introduced.

Tasks are single-entry, single-exit regions of code. Given a team of threads, typically the code is structured in a way (e.g., by using the `single`-directive) that a main thread prepares new subtasks. In general, the execution of prepared tasks is deferred, so the freshly prepared tasks are just put into a queue for later execution. While the main thread prepares tasks, the other threads in the team remove tasks from the queue and actually execute them. Each time a thread finishes execution, it removes another prepared task from the queue, until no more tasks are left for execution. This way, tasks are executed concurrently by the threads in team. If a certain order of task execution among the threads is required, the `depend`-clause can be used to explicitly specify dependencies among tasks.

The excerpt in Listing 4 showcases the usage of OpenMP tasks to overlap reading, processing and writing of work items. In each iteration of the loop, three tasks are created: one task for reading the element, one for processing and one for writing back the result. Dependency-clauses make sure that the three tasks are executed in the correct order. But as there is no dependency among different elements the processing of *Element 1* can take place in parallel to the reading of *Element 2* (*pipeline parallelism*).

Just as with the worksharing loops, the OpenMP standard defines additional clauses and constructs for management and fine-tuning of tasks in the official specification [25].

Device Offloading. The ever-increasing usage of dedicated accelerators such as (GP)GPUs in the HPC field sparked a desire to support using such accelerators in OpenMP. As a reaction, device offloading support was introduced into the OpenMP standard, starting with version 4.0.

With the `target`, `teams`, and `distribute` directives, a user can offload computations to a device, create multiple thread teams, and distribute workloads among them. Additionally, the worksharing constructs introduced earlier can also be used to further partition the work.

In many computing platforms, one has to assume that the accelerator and the host CPU do not share memory. Thus, the OpenMP standard defines a number of data-mapping clauses that enable the user to specify which and how data is mapped to/from the device.

```
int *a = ...
int *b = ...
int *c = ...

#pragma omp target teams distribute parallel for\
        map(to: a[0:], b[0:]) map(from: c[0:])
for(int i=0; i < SIZE; ++i){
        c[i] = a[i] * b[i];
}
```

Listing 5: OpenMP device offloading example (simplified).

Listing 5 presents an example for OpenMP device offloading. The arrays `a` and `b` are mapped (in the simplest case: *copied*) to the device. A number of thread teams is created (determining their number is left to the runtime system) and the iterations are distributed across the teams, so each team only computes a subset of the iterations of the loop. Inside each team, the work is further distributed up between the threads in each team (again, leaving it to the runtime system to pick a reasonable number of threads), so each device thread executes only a fraction of the iterations. After the execution on the device has completed, only the results in array `c` are mapped back to host memory.

OpenMP defines many more directives and clauses for device offloading, e.g., to instruct the compiler to create a device version of a function. More details on these directives can be found in the official specification [25] and in textbooks such as [26].

In contrast to tasks and worksharing constructs, which are fully supported in virtually all major compiler frameworks, device offloading is a relatively new feature and therefore not fully, or only experimentally, supported in many compilers at this time.

3.2 OpenCL

OpenCL (Open Computing Language) provides an open royalty-free standard for writing parallel programs to execute on heterogeneous platforms consisting of CPUs, GPUs, FPGAs, and other hardware accelerators. It defines a single programming model and a set of system-level abstractions that are supported by all standard-conformant platforms.

OpenCL is maintained by the Khronos Group, a member-funded consortium of technology companies dedicated to the creation and advance of open royalty-free standards in several markets (graphics, parallel computing, neural networks, and vision processing) on a wide variety of platforms and

devices, ranging from server and desktop to embedded and safety critical devices [17].

The standard was initially developed by Apple, and refined afterwards into an initial proposal made in collaboration with AMD, IBM, Qualcomm, Intel, and Nvidia. The proposal was submitted to the Khronos Group, who released its initial OpenCL 1.0 specification in 2008. The extended specifications OpenCL 1.1 and 1.2 were released in 2010 and 2011, respectively. New features introduced in these releases include *additions* in image formats & OpenCL C built-ins, *enhancements* in event-driven execution & OpenGL interoperability, as well as the introduction of device partitioning & separation of compilation and linking. A major leap in OpenCL development was achieved in 2013 with the release of OpenCL 2.0, which introduced features such as shared virtual memory, on-device dispatch, generic address spaces, and pipes. As of December 2018, the latest OpenCL version 2.2 brings the OpenCL C++ kernel language into the core specification, aiming to further increase programming productivity. Please refer to the official specification in [24] for a more detailed description.

The OpenCL specification describes its core ideas by using a hierarchy of four *models*:

Platform model. An OpenCL platform is a vendor-specific implementation of the standard, and defines the logical representation of the hardware capable of executing an OpenCL application. It consists of a *host* connected to one or more OpenCL *devices* that execute OpenCL *kernels*. Additionally, it defines an abstract hardware model by which devices are divided into one or more *compute units* (CUs), the latter being further divided into one or more *processing elements* (PEs).

Memory model. The memory is divided into *host* and *device* memory. For the latter, an *abstract* hierarchy composed of four disjoint regions (*global*, *constant*, *local*, *private*) is used by the kernels, regardless of the *concrete* underlying memory architecture. For handling global memory objects, predefined object types such as *buffers*, *images*, and *pipes* are provided. Additionally, the standard defines requirements for memory ordering using a consistency memory model based on that of ISO C11. In addition, an optional shared virtual memory (SVM) mechanism is specified, through which the global memory region could be extended into the *host* memory.

Execution model. Two distinct units of execution are defined: a *host program* that executes on the host, and a *kernel* described as a function over *NDRange*, an N-dimensional index space. A single element thus identified

is known as a *work-item* (i.e., a processing thread), while the total number of elements in the index space can be divided into independent subspaces, so-called *work-groups*. Kernel execution always occurs within a *context*, i.e., an environment that includes devices, kernels, memory, as well as *program* objects. Furthermore, the host-device interaction is managed by *command-queues* via *kernel-enqueue*, *memory*, and *synchronization* commands.

Programming model. From a syntactic standpoint, a kernel is similar to a standard C function. Beyond some additional keywords, the main difference is the concurrency model that the kernels implement. Two programming models define how concurrency is mapped to physical hardware. In the *data-parallel* or *Single-Program Multiple-Data* (SPMD) model, multiple instances of a single kernel are executed concurrently on multiple PEs, each with its own data and its own program counter. Alternatively, in the *task-parallel* model, computations are expressed as multiple concurrent tasks, with each task being able to run a different kernel.

The specification describes the feature requirements for HPC platforms, as well as for handheld and embedded platforms. OpenCL *profiles* define the criteria a device must meet to be considered OpenCL-compliant. The requirements for handheld and embedded devices (`EMBEDDED_PROFILE`) are a subset of the numerical and image processing capabilities that apply to HPC platforms (`FULL_PROFILE`). As our project focus is on embedded devices, we take into account the profiles supported by our evaluation platforms, as well as the the OpenCL platform & device versions. These details are discussed in Chapter 6.

Additionally, the Khronos Group has defined an OpenCL C++ wrapper API [35] to be used in conjuction with the OpenCL specification. This is built on top of the OpenCL C API, and is *not* intended as a replacement. Calls to the C++ wrapper often map very closely to the underlying C functions (e.g. `cl::Memory` is mapped to `cl_mem`), and introduce no additional execution overhead. As summarized in [31] [15], the C++ wrapper offers the benefits of a higher-level language, such as classes and exception handling. In practice, we see two main advantages of using this C++ wrapper in coding the host-part of an OpenCL application. First, function calls are simpler because C++ functions usually require fewer arguments than their C counterparts. Second, as allocation of OpenCL objects is handled automatically, there is no need to manually handle reference counts (e.g., using `clRetain*()`, `clRelease*()` in the C API).

3.3 CUDA

In the 2000s, programmers first started the trend to utilize the available processing power in GPUs. Initially, this was done by "abusing" graphics commands of the OpenGL or Direct3D API. General computations were mapped to the available graphics operations. While cumbersome, for certain problems this approach provided acceleration even then.

Due to growing computational demands on the graphics processing pipelines (e.g., more complex anti-aliasing, support of higher resolutions, ...) GPU capabilities continued to improve. In 2007, NVIDIA introduced CUDA (then meaning Compute Unified Device Architecture) as an official interface for performing general-purpose computations on GPU hardware (GPGPU).

By generalizing the architecture from a dedicated graphics-specific one towards a generic SIMD-based many-core GPGPU architecture, NVIDIA was able to expose the raw computing power of the GPU to a larger audience of software programmers. Core parts of CUDA are a compiler for the CUDA language (nvcc) and the CUDA libraries. The compiler accepts C source code, in which the CUDA programming constructs are embedded. It then extracts the CUDA code constructs and compiles these parts with a specialized tool chain. The remainder of the code is compiled with an conventional compiler of the host system (e.g. gcc or icc). Later versions of CUDA extended nvcc to accept not only C, but also C++-code.

The CUDA libraries offer support on two different levels: A higher level API, called Runtime API (the functions are usually prefixed with `cuda`), and a lower level API, called Device API (the functions are usually prefixed with `cu`). While offering similar functionality, the lower level API is often more flexible and allows run time loading and starting of kernels.

With later versions of the CUDA environment (the most recent version is 10), more special libraries were and continue to be added. These provide GPU-acceleration on a more abstract level and are easier to use by less experienced software developers than manually coding CUDA kernels. The libraries include functionality like BLAS, FFT, solver, sparse matrix operations, graph, neural network, and many more (not to mention other 3rd parties libraries available).

Also part of CUDA are a (growing) number of developer tools. These include profilers and debuggers that help the programmer to write and optimize code on for the GPU.

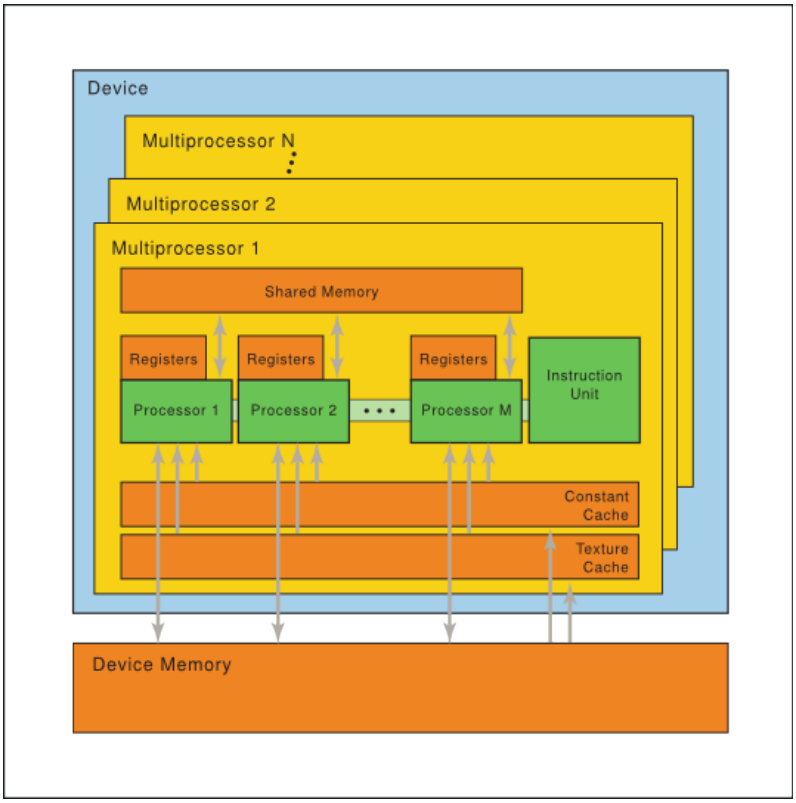


Figure 3.1.: Fundamental NVIDIA GPU model ([7]).

3.3.1 Architecture

Newer generations of GPUs have support for a broader range of functionality (e.g., more data types, more atomic operations, more dynamic parallelism, new memory features, ...), so a capability mechanism was built into the CUDA libraries right from the start. This allows the compiler to selectively target different GPU generations/types. The API also allows querying the supported features at run-time, which allows to create a capability independent implementation that can activate optimized code paths only when available.

While the concrete GPU architectures differ in certain details, they share the same fundamentals. Figure 3.1 shows the hardware from the programmers perspective. Each GPU has multiple Streaming Multiprocessors (SM), and each of the SMs has multiple cores. The cores in one SM execute the same thread in SIMD fashion. Group of threads are arranged in a grid, with each group called a block. A subset of the threads in a block, the so called warp, are simultaneously executed by the hardware. The warp size is commonly 32. In order to hide memory access latencies, multiple warps are scheduled on the same hardware: Once a warp stalls due to a memory access missed in cache, it is suspended. Then, the next ready warp in scheduled onto the

hardware. In earlier GPU generations, all of the warps had to come from the same kernel. Current generations can extend the scheduling of warps across kernel boundaries, potentially yielding higher throughput.
Each thread has access to different kinds of memory:

Registers These are only thread-local, and are the fastest kind of memory available.

Shared Memory Shared Memory is SM-local memory, slower than registers, but much faster than device memory. It has a higher memory capacity than the Registers and is used as a spill target in case the number of Registers required in a kernel exceeds the hardware limit.

Device Memory Usually GDDRx / HBM memory onboard the GPGPU card. This is the slowest, but in general largest memory directly available to the GPU. Often, an on-chip cache hierarchy tries to reduce the number of accesses to device memory. Specialized caches exist for specific use-cases. E.g., Constant and Texture Cache. The bandwidth to Device Memory is very high (up to 900 GB/s for current HBM-based cards). But the memory system can achieve that throughput only when a number of restrictions on memory access patterns across threads are obeyed.

Host Memory Newer CUDA versions support a unified memory model, where the GPU can access the host system memory via a bus such as PCI Express or NVLink. These accesses are even slower than those to Device Memory.

An exception to this are certain embedded system-on-chips (e.g., Tegra-class, Xavier SoC) that have the CPU and GPU *sharing* the same physical memory. While this memory is not as fast as the high-performance GDDR or HBM memories on desktop/server-class GPUs, its shared use can often avoid the costly data copies required between physically separate CPU and GPU memories.

3.3.2 Compile Flows

Regular Flow

The regular CUDA compile flow uses the CUDA `nvcc` compiler driver to partition the original program (having an extension `.cu` instead of `.cpp/.c`, to distinguish it from pure C/C++-code) into two parts: code that gets executed on the GPGPU device, and the remaining code that gets executed on the host system.

The host part is compiled and linked with the standard host compiler. The extracted code is compiled with a proprietary NVIDIA compiler (initially based on the Open64 compiler, now moved on to LLVM) into PTX. PTX stands for Parallel Thread eXecution and is NVIDIAs portable assembler code for its GPUs. It supports multiple types and generations of GPU (using so-called capabilities), and is later translated into binary code and linked together with the original host object files into a heterogeneous executable.
When using the low-level Device API, it is also possible to dynamically load PTX or translated binary code manually at runtime into the GPU.

Alternative Flows

A number of academic alternatives to the proprietary NVIDIA flow exist. Many try to compile from a different input language (e.g., OpenCL) instead of CUDA. Two of the more robust approaches were used in this evaluation to demonstrate the possibility of programming NVIDIA hardware with open languages.

PoCL Translates from OpenCL source code to PTX portable assembler [14].

SnuCL Can Translate from OpenCL source code to CUDA source code and vice-versa [18].

3.3.3 Programming CUDA

Due to the many processing cores (thousands in desktop/server-class devices) on a GPU the usual approach is to process highly parallel workloads to the GPU and implement them with thousands of threads.
Typically, this heterogeneous implementation first copies the required data from the host memory to the GPU device memory, then starts the kernel execution, and afterwards copies the results back to the host system. Due to the large overhead for these, only kernels which have a high computational intensity and start thousands of threads result in a net speedup.
Since this study focused on embedded computing, the Jetson TX2 and AGX Xavier NVIDIA platforms employed for testing both have physically shared memory between the GPU and CPU(s). They can thus avoid data copying and enable GPU acceleration even for smaller problem sizes. Note that very small problems will not be beneficial even on these platforms: Moving execution between CPU to GPU (suspending/creating threads) always has an overhead even by itself.

CUDA Language

NVIDIA has extended the C/C++ language with different constructs to help the programmer.

New Keywords: These are attributes to functions and variables. For functions, they designate which should become a kernel callable from host code (`__global__`), and which are functions that only get executed locally on the GPU device (`__device__`).

Similarly, the memory attributes can describe whether certain variables should become shared, global, or are in unified memory.

New Operators: Usually, a kernel is called from the host in a manner similar to a function call, but with additional invocation parameters enclosed in <<< and >>>. These give the number of threads that should be generated (and how these threads should be arranged in a 3D-grid). Note that kernels can also be launched using the lower-level Device API.

CUDA Libraries: These not only include functions that replace regular C/C++-functions, e.g., the math functions sin, cos, sqrt, but also include functions that give the developer access to GPU-specific operations, namely synchronization functions (fences), allocating and transferring to/from memory on the device, as well as atomic functions.

The libraries also include many functions for managing the device itself, e.g., querying capabilities, or selecting a device in case of multiple GPUs being present in a system.

Of course the libraries contain not only these functions, but also the definition of new data types required for them.

Finally there are some pseudo-variables available in the device code, that identify each thread with its id and its coordinates in the thread grid (`blockDim, blockIdx, threadIdx, ...`).

Listing 6 gives a small example CUDA program containing all of the language additions described above.

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
        C[i] = A[i] + B[i];
}

int
main(void)
{
  [...]
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    [...]
    float *d_A = NULL;
    float *d_B = NULL;
    float *d_C = NULL;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    [...]
    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d threads\n",
           blocksPerGrid, threadsPerBlock);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    [...]
    return 0;
}
```

Listing 6: CUDA example (shortened, without error checking). Showing a vector addition of vectors with 50000 elements, performed by 50000 threads, each thread handling one component.

3.3.4 Limitations

The CUDA technology does have a number of limitations that may hinder its practical use:

- No serial execution of CUDA programs. In earlier versions, the compiler was capable of generating serialized executable code for the host system from the designated GPU kernels. This allowed for easier functional debugging, even on system not having a GPU. This feature was later removed and is no longer available.
- Vendor-lock in. As an NVIDIA product CUDA supports only NVIDIA devices for execution. There are some academic projects ([18, 19]) trying to translate CUDA to OpenCL, but they are far from perfect and have several limitations. Recently, AMD has begun to offer a toolflow (ROCm HIP) for translating CUDA to portable C++ code.
- Only limited C++ support. Several C++ constructs are not supported in CUDA kernel code and only limited support for the STL is available. Furthermore, allocation of memory via `new` is not possible, RTTI is not available, and exception handling is unsupported.

Benchmarks 4

4.1 Selection Criteria

Since our study is based on the implementation of real application, we had to choose suitable candidates. We employed the following three criteria to guide this process.

Representative The kernels for the benchmark should be representative for real computations in the automotive field, ideally from the AD/ADAS domain.

Parallelizable As the study evaluates parallel programming models, the selected code must offer opportunities for actual parallelization.

Impact While one can parallelize almost anything, the most important reason for parallelization is speed-up. So the selected code must not only offer opportunities for parallelization, but these should also result in a speed-up of the ported code, at least on some execution platforms.

The kernels should also execute in an environment typical for their actual use. We thus do not only consider pure kernel run-times when judging speed-ups. Instead, we also include the data type conversion and copying, if required, that would have to be performed in a real application-level use of the kernel.

4.1.1 Autoware AD/ADAS Suite

As an open software suite that contains everything necessary for autonomous driving (AI-algorithms, signal processing, location, mapping, ...), Autoware [16] was chosen as starting point for the benchmark selection. Started as an academic project at Nagoya University (Japan), and now continued by the Autoware Foundation initiated by Tier IV et al., this software suite contains a plethora of relevant algorithms, and has the advantage that it is licensed under the New BSD License.

Autoware uses ROS (Robot Operating System [30]) as base framework for message passing between its components. Each Autoware task is realized as a ROS module (see Figure 4.1). Since ROS provides the capability to record message traffic as well as inject it, the data of complete runs (e.g., based on actual test drives) can be recorded and played back later. This mechanism was used to obtain both input data for the kernels as well as “golden” output data.

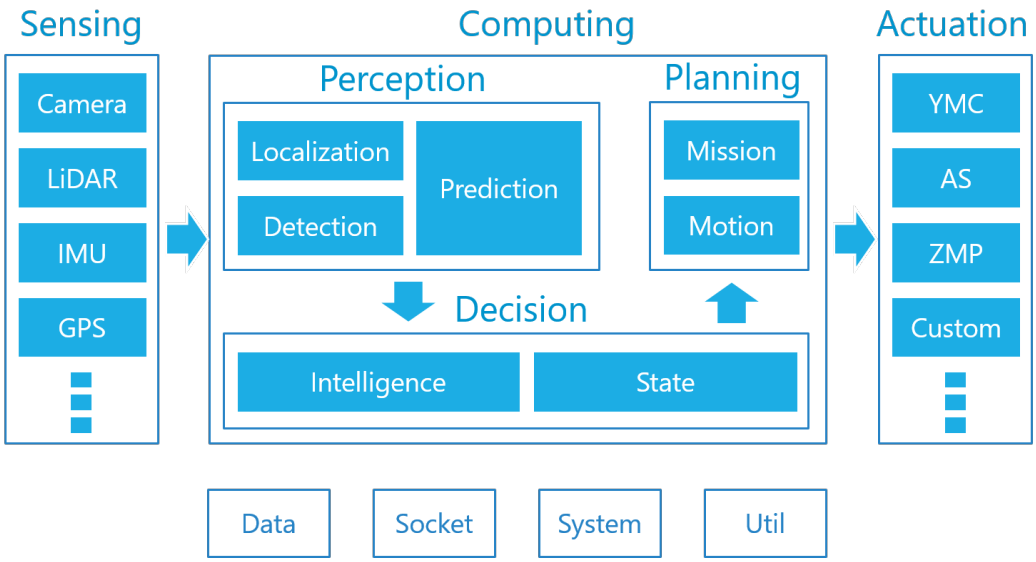


Figure 4.1.: Autoware overview [2].

Specifically, we extracted our kernels form Autoware 1.6 with ROS Indigo and used the `sample_moriyama` data set for inputs. It contains the recordings of driving along an eight minute long course, captured, e.g., by a VELODYNE HDL32-E LIDAR and a JVAD GPS Delta 3 GNSS.

We began by profiling the execution behavior of the Autoware suite with the data set on a server with four AMD Opteron 6134 CPUs, having a total of 32 Cores and 128GB of memory. With this profile, several compute intensive modules were identified. From these, the most promising candidates for parallelization were hand-picked and extracted.

Extraction means that we created full functional versions that could execute outside of the ROS environment. To this end, the inputs and outputs to/from the functions were recorded (via manual code instrumentation), the code for the functions themselves extracted, and placed into a generic benchmark skeleton. Finally, all references to third party libraries were either removed (if they were not relevant for the operation of the function) or replaced by out own implementations. The latter was done to keep the source of the benchmarks small and to be independent of the implementations in the library. The next section describes the selected kernels in detail. Note that we verified that, even with our own implementations of essential library functions, we achieve the same execution times as the original.

As the focus of our study lies on the portability of the different parallel implementations, but not on the quality of the algorithms, the algorithms were used in their original precision. E.g., if the Autoware implementation used double-precision computations, we did not attempt to implement a

Kernel	Input Size [MB]	Output Size [MB]	# of Testcases
points2image	19 000.000	4 300.000	2500
euclidean_clustering	45.000	54.000	350
ndt_mapping	4 200.000	0.008	115

Table 4.1.: Kernel statistics

possibly faster version in single-precision arithmetic. An exception to this rule were the implementations on the Renesas platform, as its OpenCL software stack only supports single precision computations. For this case, separate single-precion implementations of both the original sequential as well as the parallelized versions were created. This included recomputing the "golden" reference output data in single-precision by the sequential version.

4.2 Kernels

This section discusses the three kernels we picked for the study.

4.2.1 points2image Kernel

Original Autoware Package: points2image

Input: A point cloud (from LIDAR) with intensity and range information

Output: LIDAR point clouds projected onto a given 2D view. When multiple LIDAR points are mapped to the same 2D view point, the closest LIDAR point is selected for projection. Note: In the original code, the case where multiple LIDAR points with the *same* range were projected onto a single 2D point led to unspecified behavior. For deterministic execution, we decided to consistently project the LIDAR point having the *highest* intensity.

Changes: We resolved the unspecified behavior and fixed an obvious error in the code. The latter was a unit mismatch: When computing the projection, a conversion from meters to centimeters is performed, but in the comparison of distances, the old value using meters as unit is used (see Listing 7).

Comment: With thousands of points getting projected at the same time, this seemed like a very good candidate for parallelization. However, in practice, multiple 3D LIDAR points may end up being projected to the same 2D point. This leads to a race condition in fully parallel execution, and can result in incorrect results when not handled with care (e.g., using atomic operations).

```
1          int pid = py * w + px;
2          if(msg.distance[pid] == 0 ||
3             msg.distance[pid] > (point.data[2]))
4              {
5                  msg.distance[pid] = float(point.data[2] * 100);
```

Listing 7: Error in the original points2image-code. The comparison ignores the factor 100 that was applied during the assignment. Either `msg_distance` must be also multiplied by 100, or the `point_data` must be divided by 100 for the comparison.

4.2.2 euclidean_clustering Kernel

Original Autoware Package/Node: lidar_tracker/euclidean_cluster

Input: A point cloud.

Output: Each point cloud is segmented into five clusters (based on distance from the center), then points within a cluster are clustered again based on euclidean distance.

Changes: The original implementation uses a kd-tree to compute the distances between points, realized as a function call to `radiusSearch` in the Point Cloud Library (PCL). To make our implementation library free for easier parallelization, the call to PCL was replaced by a precomputed look-up table of all distances. To verify that this standalone kernel has a similar performance to the original one, we benchmarked it against the original Autoware version. Figure 4.2 shows that the performance of the two implementations is quite close. We also fixed an error in the original code: A function returned the $\arctan$ of a value, and its caller applied $\arctan$ to result yet again. This was also fixed in later versions of Autoware itself.

Comment: Our replacement of `radiusSearch` with a lookup-table is not optimal from a performance standpoint. The LUT uses a *triangular* matrix to save memory, which is essential, e.g., for the small Renesas V3M platform. On the larger platforms having more memory, it is much faster to use a *square* matrix holding all point-to-point distances. We have created a second version of the code using this faster data organization, which we also backported to the sequential code, speeding it up as well. Thus, when giving speed-ups for euclidean_clustering, we will list them both relative to the original and to the sped-up sequential baselines.

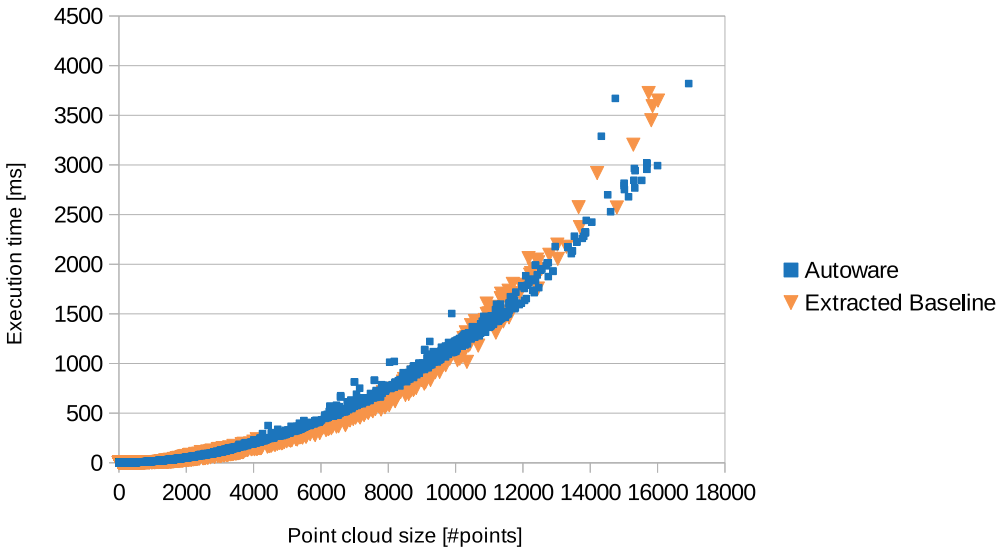


Figure 4.2.: Comparison of the original Autoware algorithm for `radiusSearch` in the euclidean_clustering and its replacement.

4.2.3 ndt_mapping Kernel

Original Autoware Package/Node: ndt_localizer/ndt_mapping

Input: A point cloud from LIDAR and a point cloud representing the map.

Output: A transformation (rotation and translation). The transformation is the best transformation that fits the given LIDAR point cloud to a given map point cloud using normal distribution transformations (ndt).

Changes: To make the kernel code library free, we replaced the complex SVD algorithm, used to solve a 6x6 equation system, with a much simpler Gauss solver. For production code, the impact of that change would have to be investigated more closely, as the SVD solver deals better with singular matrices and has a shorter worst-case execution time. We checked to make sure that the degenerate case for the Gauss solver is not triggered by our input data. Similarly, we replaced a kd-tree for computing the minimum distance of a point outside of tree to any points in the tree with a linear search. As shown in Figure 4.3, neither of these changes slows down the sequential baseline. On the contrary, they actually improve the runtime of the sequential baseline, and thus do not artificially inflate the speedups we achieve by parallelization.

Comment: For SLAM (simultaneous location and mapping) algorithms in general, the newly discovered points are added to the existing map. This leads to the map point clouds *growing* during execution. To combat

this ballooning of size, the original Autoware code updates the map only after the car has driven a certain distance. For our benchmarks, the behavior leads to smaller testcases having shorter runtimes (fewer points to start with, fewer points added), and larger testcases have disproportionately longer runtimes (more points to start with, even more points iteratively added during execution).

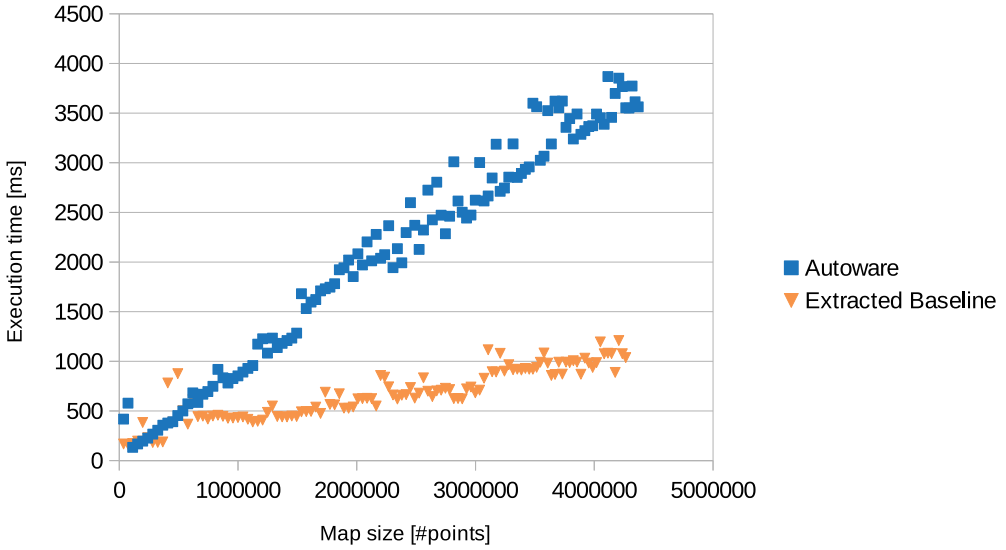


Figure 4.3.: Performance comparison of the original Autoware algorithm for ndt_mapping and our library-free version, which is even faster.

4.3 Limitations of Benchmarking Approach

While our approach yielded suitable kernels for the rest of the study, 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.
- The kernels should not be used to compare the different platforms in terms of their performance, as all platforms examined in this study are very different. Not only in their compute capabilities, but also regarding their space and power requirements, and of course in cost. Some platforms have additional compute abilities that have not been used in the evaluation, but which could accelerate certain workloads

even further. E.g., the NVIDIA AGX Xavier platform has an additional VLIW coprocessor intended for computer vision applications, which was not used at all in our study.

Also, the single-precision versions of the code we created specifically for the Renesas platform would need to be ported to the other platforms as well for valid performance comparisons.

- Autoware has proven to be extremely useful for our work. However, with its roots in academic fundamental research, it 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 copies between host and accelerator memories across ROS node boundaries.

- The extracted kernels are not suitable for production use. For example, precision issues were only addressed as required for the recorded test cases, and error checking was reduced to a minimum. Security issues were not considered at all. All of these aspects would have to be addressed before real-world use of the kernels would be advisable.

Evaluation Platforms

5

In this chapter, we give an overview of the hardware platforms we used in our study. As mentioned previously, they are located at very different points in the performance, energy requirement, and cost-space. Thus, they serve here only to illustrate the wide variety of heterogeneous embedded computing systems, and to provide context for discussing application portability. Our results are not usable as direct performance comparisons between the platforms.

Figure 5.1.: Picture of all four used platforms (left to right, top to bottom): NVIDIA AGX Xavier, Xilinx Zynq UltraScale+ MPSoC ZCU102, NVIDIA Jetson TX2, Renesas R-Car V3M. The bottom right system is an embedded x86-64 PC platform for size comparison.

5.1 NVIDIA Jetson TX2

The NVIDIA Jetson TX2 is an embedded heterogeneous computing platform. The main component is an "Parker" SoC from the NVIDIA Tegra line. It contains two kinds of ARM ISA CPUs: a large dual-core ARM Denver2 block, and a smaller quad-core Cortex-A57 block. As GPU, it contains a 256-core Pascal-class unit. Unusually compared to desktop/server-class GPUs, the

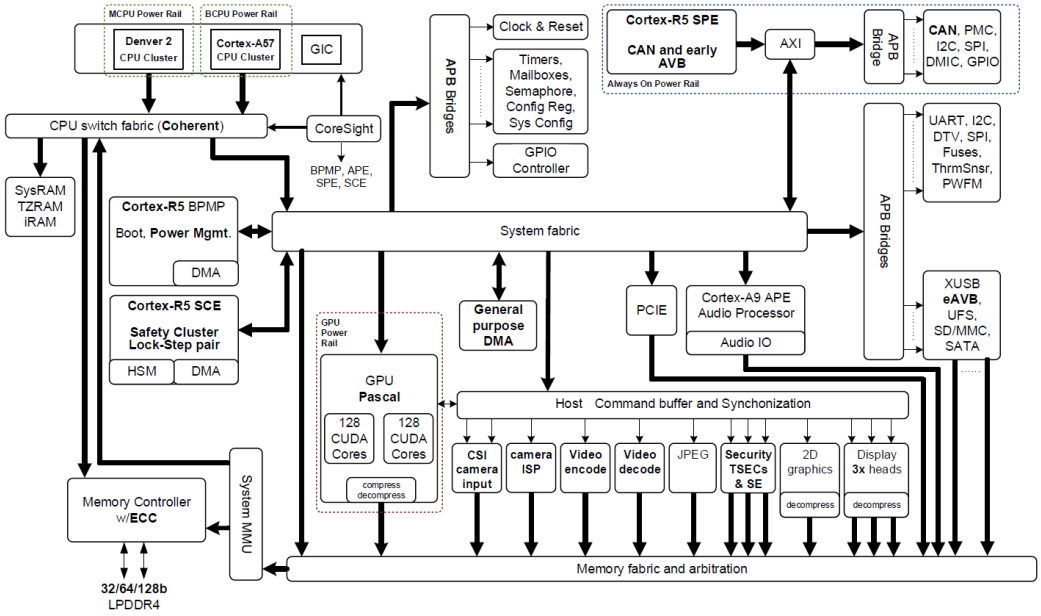


Figure 5.2.: Block diagram of the NVIDIA Jetson TX2 [8].

Pascal GPU does not have dedicated device memory, but is connected to the same memory controller as the ARM cores (8 GB LPDDR4-SDRAM, attached with a 128-bit wide interface). While this is slower than the memory used in desktop/server-class GPUs (often faster GDDR or even HBM memory), the physically shared nature of the memory on the Parker SoC can avoid the data copies that are required between host and GPU memories in the desktop/server use-case.

The Jetson TX2 is available in different form factors, including an industrial module in a rugged design. For the evaluation, we used the mini-ITX board from the NVIDIA TX2 developer kit. It has a multitude of peripherals and ports for I/O, including a dual CAN bus controller, network, video encoder/decoder, and many more. As these are not relevant for this study, we will not describe them further.

The CPU and GPU support different power profiles, with a typical power draw being 7.5 Watts. Peak power requirements at full clock speeds can reach 18 Watts.

Figure 5.2 shows the block diagram of the Jetson TX2, and Table 5.1 lists the details of key hardware components. To give an impression of the single-core CPU performance, we employed the Himeno [10] benchmark, running it on a single core, for both the Denver2 and the A57 cores.

The Pascal-class GPU acting as accelerator to the CPUs consists of two streaming multiprocessors, offering a total of 256 cores with SIMD processing in

	NVIDIA Jetson TX2	NVIDIA AGX Xavier
GPU	256-core Pascal@1.3GHz	512-core Volta@1.37 GHz
CPUs	1 dual-core Denver2@2.0 GHz 1 quad-core Cortex-A57@2.0 GHz	4 dual-core Carmel@2.26 GHz
Cache	2x 2 MB L2	4 MB L3 + 2 MB L2/ 2 Cores
RAM Memory bus width/bandwidth	8 GB LPDDR4@1.866 GHz 128 bit/59.7 GB/s	16 GB LPPDR4x@2.133 GHz 256 bit/137 GB/s
Power (typical/max)	7.5/18 Watt	20/47 Watt
Himeno-Benchmark (Single CPU Core)	741 MFLOPS (A57) 2155 MFLOPS (Denver2)	2979 MFLOPS

Table 5.1.: Specifications of key components in the NVIDIA Jetson TX2 and AGX Xavier boards.

each. The GPU is programmed using NVIDIA CUDA (see Section 3.3) and has a CUDA feature capability level of 6.2.

5.2 NVIDIA AGX Xavier

As successor to the NVIDIA TX2 described in 5.1, the NVIDIA AGX Xavier platform has far more capabilities. In addition to CPUs and GPU, it also has an additional vision accelerator (7-way VLIW) and two tensor accelerators for inference in neural networks. Both were not used during this evaluation, as they were not programmable in CUDA, and would not be applicable to the selected benchmarks.

The “Xavier” SoC at the heart of the AGX system, specified in Table 5.1, has eight Carmel CPU cores, organized as four dual-core clusters. Each of these cores is faster than the Denver2 core, which was the fastest CPU core on the “Parker” SoC of the TX2 platform.

The AGX platform als has a far faster memory system, and a GPU that is larger (512 cores) and of the newer Volta-generation compared to the Pascal GPU in the TX2. The additional features supported by the Volta-class GPU are reflected in its updated feature capability level of 7.2.

As with the TX2 system, CUDA is used to program the GPU component of the SoC.

5.3 Renesas R-Car V3M

The Renesas R-Car V3M is an automotive SoC primarily designed for front-camera applications, surround view systems, and LIDARs. It aims to provide

high performance for computer vision (CV), while supporting very low-power consumption and a high level of functional safety [29].
The core of the **hardware platform** is the V3M SoC composed of several computing subsystems ranging from general- and specific-purpose CPUs, image & video processors, memory interfaces, and connectivity blocks. From the board documentation [28], the main component blocks are:

- Dual ARM Cortex-A53 MPCore cores clocked at 800 MHz (for application programming).
- Dual lockstep ARM Cortex-R7 core clocked at 800 MHz (for running AUTOSAR).
- An IMP-X5+ image recognition engine, featuring two CV engines (each with 32 threads), with local 896 kB SRAM. This accelerator is the target of the OpenCL stack from Coldplay. However, the underlying hardware has a number of restrictions:
  - Unsigned integers are not supported.
  - Supports only single precision floating point operations.
  - Division (integer and floating point) is not supported at all.
- A video encoding processor for inter-device video transfer.
- 2 GB of DDR3-SDRAM (DDR3L-1600, 32-bit wide), with a bandwidth of 6.4 GB/s.
- Needs less than 15 Watt of power.

The **software development** tools for the Renesas R-Car V3M are ComputeAorta and ComputeCpp, both developed by Codeplay. ComputeAorta and ComputeCpp allow developers to run OpenCL and SYCL (respectively) applications on the CV engines in the IMP processor.
For application development, it is important to keep in mind the value of `CL_DEVICE_MAX_MEM_ALLOC_SIZE`, i.e., the maximum size of memory object allocation in bytes. For the V3M, this value is 110 MiB and prevents allocating `cl::Buffer` objects larger than that size. Further platform-specific optimization guidelines and notes for both OpenCL- and SYCL-based applications are provided in the R-Car guide [4].

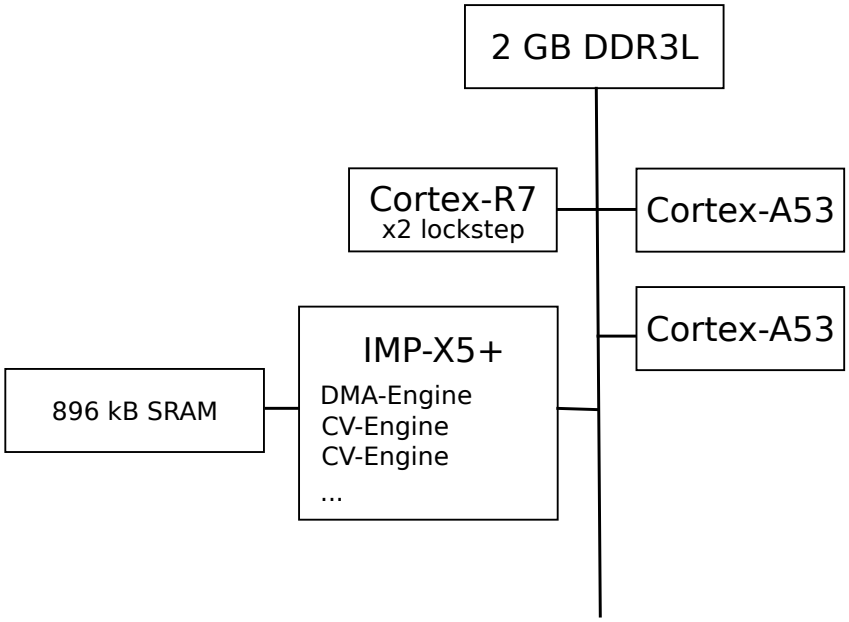


Figure 5.3.: Renesas R-Car V3M block diagram (simplified) [29].

5.4 Xilinx Zynq UltraScale+ MPSoC ZCU102

The ZCU102 is an evaluation board for rapid-prototyping based on the Zynq UltraScale+ MPSoC (multiprocessor system-on-chip) device family [42]. This section is organized as a *hardware platform* and a *software development* discussion.

The **hardware platform** uses a Xilinx UltraScale+ MPSoC [41] as its key component. The Multi-Processor System-on-Chip contains a *processing system* (PS), integrated with a FPGA-like block called *programmable logic* (PL), in a single device.

The PS is formed around the embedded processor(s) and access to main memory, making it suitable for running an operating system and software applications. The FPGA fabric in the PL is suitable to implement application-specific accelerators. In addition, the platform includes on-chip memory, multiport external memory interfaces, and other peripheral interfaces. Figure 5.4 gives an overview of the MPSoC device as documented in [40]. Key components are:

- Quad-core Cortex-A53 Application Processing Unit (APU): ARM v8 architecture (64-bit) @1.5GHz
- Dual-core Cortex-R5 Real-time Processing Unit (RPU): ARM v7 architecture (32-bit) @600MHz

- Mali-400 GPU: ARM @667MHz

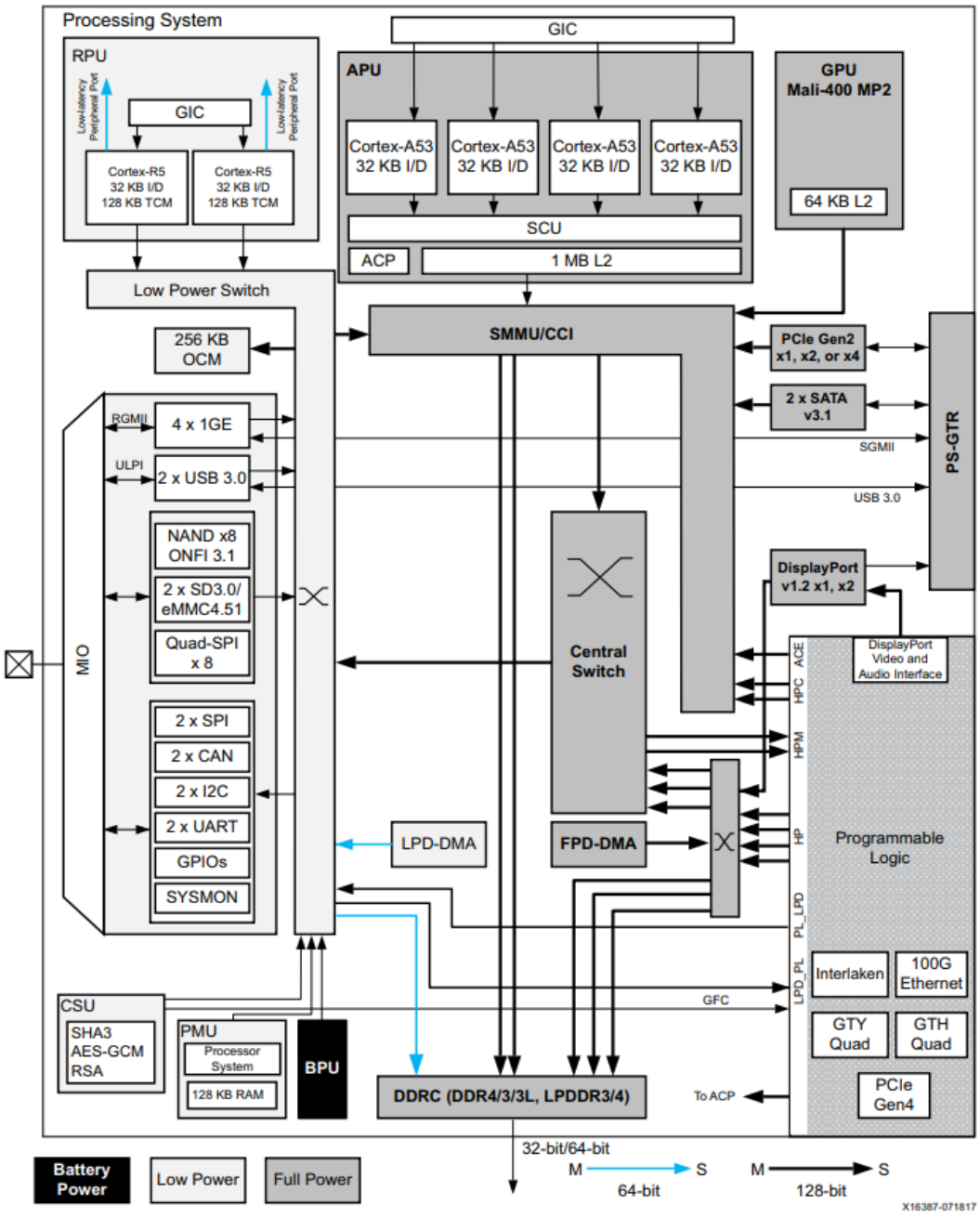


Figure 5.4.: Zynq UltraScale+ MPSoC block diagram [40].

The PL fabric on the ZU9EG chip used on the ZCU102 board holds:

- System Logic Cells: 600K
- Total block RAM: 32.1 Mb
- DSP slices: 2 520

The major **software development** tool for Xilinx embedded devices is SDSoC [32]. Its programming flow follows the standard *software-centric* steps of code development, compilation, linking for the platform/device, profiling the system for performance in simulation/emulation, and measuring the actual performance on the MPSoC itself.

Note that a hardware-centric flow (using e.g., Verilog and VHDL) is also supported, but it is out of the scope of this work. The software-centric flow focuses on the embedded processor application, as well as the acceleration of specific computationally-intensive functions (kernels). The latter are translated into hardware blocks capable of running on the FPGA fabric.

The front-end tools convert kernel functions expressed in C/C++/OpenCL into a hardware description language (HDL), which is then compiled for the PL using the back-end Vivado HLS tool. The accelerated function is expected to leverage the parallel architecture of the PL region by exploiting instruction-level and task-level parallelism [39]. The performance of the accelerator is typically measured in terms of the *initiation interval* (II), i.e., the number of clock cycles before the function can accept new input data, which is a measure of throughput. Ideally, the function achieves an II of one, meaning a new set of inputs is accepted every clock cycle. Often, this also implies that results are output at the same rate (one per clock cycle).

The application code, composed of the processor and accelerator parts, can be compiled for two modes: *emulation* and *hardware build*. As the compilation times for emulation are considerably shorter, this mode allows for quick design iterations (i.e., a GPU-like experience). Emulation can also be used to get an initial profile of the application, measuring communication to/from accelerator, and accelerator workloads.

Building for *hardware* generates the processor and accelerator binaries that will run on the actual system. Since this requires a full mapping, placement and routing step to the PL FPGA, it takes considerably longer than software compilation or building for emulation. We discuss the hardware build times in our evaluation criteria in Chapter 7.

While the FPGA-like flexibility of the PL would allow the creation of almost any processing microarchitecture, the use of the PL from OpenCL programs is subject to a number of limitations. A key one is that only OpenCL 1.2 is supported. While some vendor-specific extensions are available, many useful OpenCL 2.x constructs are missing. Also, OpenCL programs targeting FPGAs often have to be written in a style completely different from those targeting CPUs and GPUs. This is discussed in Section 7.5.

Source code examples showing the conventions in FPGA-specific OpenCL can be seen in a GitHub repository [38].

6 Benchmark Implementation and Lessons Learned

In this chapter, we share the experiences made while implementing the benchmarks in the parallel programming models.
We report details on all steps of the implementation processs, starting from initial setup, to programming, tool-support, and finally the achieved performance.
The experiences gained during actual, day-to-day use of the programming models gives a first indication on their usability and applicability, which is the central focus of this study.

6.1 OpenMP

As discussed in Section 3.1, OpenMP uses compiler directives to specify how to parallelize a piece of code. Thus, the development of code making use of OpenMP for parallelization, requires an OpenMP-capable compiler.
For each of the relevant platforms investigated in this survey, multiple OpenMP-capable compilers are available. Virtually all major compilers for Linux (the OS for all of our platforms), such as GCC or LLVM Clang, support OpenMP directives. However, depending on *when* a construct was added to the OpenMP standard, the level of support for it in the various compilers may differ.
Well-established OpenMP constructs target multi-threaded execution on the CPU. These were present in the initial versions of the standard, or were added many years back, and are thus being well supported in almost all compilers. In contrast , the compiler support for the recently introduced constructs, is often still limited and under development. In particular, this includes the the device offloading features, which would have been very interesting to use in our study of programming heterogeneous platforms.
Due to the lack of robust implementations for OpenMP device offloading, we concentrated on the traditional constructs for parallelizing execution on multithreaded CPUs. Once better support for offloading becomes available (using the OpenMP `target` directive), it would be very interesting to examine our OpenMP benchmarks on GPUs as well.
The installation and setup of OpenMP on the platforms described in Chapter 4 is very simple. Most of the C/C++ compilers available with the platforms'

Linux support OpenMP and a suitable OpenMP runtime implementation or can easily be installed from vendor/project repositories. Runtime implementations for OpenMP are typically based on standard threading mechanisms as offered by the underlying operating system, e.g., `pthreads` on Linux. Therefore, there is usually no need to install any additional drivers, large software-stacks, or to make modifications to the OS kernel. Note that we did not test OpenMP on the Renesas V3M board, as the power-saving dual-core ARM CPU on that platform does not have a very high performance for multi-threaded execution.

The invocations of the compiler to transform OpenMP-annotated code into a multi-threaded executable require only limited changes (e.g., compiler options), and integrate well with existing build systems, such as *GNU make* or *CMake*. Usually, it is sufficient to add a simple compiler flag to allow the translation of code with OpenMP pragmas for parallel execution.

As described in Section 3.1, a developer using OpenMP to parallelize parts of an application does so by interacting with the compiler via the compiler directives defined in the OpenMP standard. In C/C++, these directives are realized using the `pragma` mechanism. These pragmas can be added to existing code very easily, enabling the developer to reuse an existing, serial implementation without the need for restructuring (as in CUDA), or even having to partition the application into multiple files (as with OpenCL).

The parallelization through the inclusion of compiler directives into the code also allows for an incremental approach. I.e. different parts of the code can be parallelized independently from each other, and in a step-by-step fashion. The parallel sections can be expanded over time, later merging multiple parallel parts of the code into a single parallel region.

During the parallelization of a particular section of code, the rest of the application often remains unaffected. It thus is possible to execute the whole application to test, debug, and assess performance.

Since the introduction of compiler directives into existing code often takes only a short amounts of time (in the range of minutes), the effects of parallelization can be assessed very quickly and in short development cycles. This allows to determine early whether a parallelization approach will be profitable or not. For other programming models that require more substantial changes to the code up-front, this is generally not possible and leads to far longer development cycles.

The ability reuse of existing, serial implementations also reduces the risk of introducing a computational error during the process of parallelization. The already-tested core functionality of the algorithm remains untouched, as the

compiler directives only add parallel semantics that can be exploited by the compiler.

Looking at the tools eco-system for OpenMP, we found that, in contrast to the other two programming models investigated in this survey, the availability of sophisticated, free or open-source *profilers* is limited. Most tools for performance analysis of OpenMP code (e.g., Intels VTune Amplifier) target large-scale HPC workloads, and are not tailored for use in embedded scenarios. While standard profilers such as `gprof` etc. can be used to identify the parts of an application that could benefit from parallelization with OpenMP, they do not allow for a detailed analysis of individual OpenMP threads.

For performance, OpenMP benefits from the maturity of the runtime implementations. These provide good performance for shared-memory multi-core processing, while introducing only a small overhead for thread maintenance. However, when scaling-up from *multi-core* CPUs to *many-core* GPUs, the number of threads available for simultaneous execution on a GPU is typically much higher than the number of threads available to OpenMP on the CPU. In a scenario with strong data-parallelism, where hundreds or even thousands of elements can be processed independently, this could limit performance of OpenMP code in comparison to native GPU-based acceleration (e.g., using CUDA and OpenCL). However, in our implementation survey we only experienced this limitation once, and only by a small margin. In all other scenarios, OpenMP was competitive with the other programming models. More details on performance numbers can be found in Chapter 7.

Porting an OpenMP implementation from one platform to another is typically fairly simple. Because the high-level OpenMP parallelization mechanisms leave the determination of low-level, platform-specific performance parameters to the compiler, just re-compiling the OpenMP code on the new platform is usually sufficient for porting.

6.2 OpenCL

By being an open standard, OpenCL opens the possibility of having several implementations available for a given platform, either closed (e.g. Intel, AMD, Nvidia, Xilinx, etc), or open source (e.g. POCL [14]).

The available capabilities provided by such implementations vary, e.g., from OpenCL 1.2 supported by Nvidia and POCL, going through OpenCL 1.2 with *some* OpenCL 2.0 features as pipes supported by Intel and Xilinx FPGAs, up to OpenCL 2.x typically offered for high-end AMD and Intel devices.

Embedded accelerators pose additional challenges in some cases. The platforms selected for this study (Chapter 5) all provide full OpenCL 1.2 support,

which in general is sufficient to express parallel heterogeneous computations. Thus, we have used OpenCL 1.2 to be as portable as possible across platforms. Often, OpenCL 2.0 features would just improve programmability (e.g., by allowing more concise source code). However, the lack of certain features *not* available in OpenCL 1.2, but only in 2.x can actually have a significant performance impact, as will be discussed below.

On the other hand, OpenCL declares some capabilities to be optional, meaning they can be missing even in a fully compliant implementation. These optional features do limit the portability of code, even between OpenCL software stacks with the *same* version number.

Specifically, double precision computations are one such optional feature. This affects, for example, our porting of the points2image benchmark to the Renesas platform, which does not support double precision in its OpenCL 1.2 software stack. We were thus forced to develop a dedicated single precision version of the benchmark, including a new validation suite and new golden reference output data.

The installation and setup of OpenCL on the selected platforms was relatively simple. For the Renesas V3M, it was only required to copy the OpenCL drivers into the platform filesystem. Updated V3M drivers were released regularly by Codeplay, with 0.3.2 being the latest public release used in this study.

The case of the ZCU102 FPGA platform was different, since no actual driver installation was required. All runtime files were generated by Xilinx' SDSoC 2018.2 development tool, and copied to an SD card together with the rest of the run-time files. After booting the system from the SD card, the hardware-accelerated OpenCL applications could be executed.

In addition to the OpenCL compilers and drivers, another important part of the OpenCL tool ecosystem are the *auxiliary* tools that assist a developer in debugging and porting. For debugging, we have utilized a number of open-source tools. First, Oclgrind [23] features a virtual OpenCL emulator that is highly useful for OpenCL development. Particularly, its utilities are able to debug memory-access errors, detect data-races and barrier divergence. Such problems are hard to locate during development, and in some applications, they rarely manifest during normal execution. The only drawback we observed from using this tool was that its analyses require longer runtimes.

Second, CodeXL [5] is a tool suite that allows GPU debugging, profiling, and static OpenCL analysis among others. It works mainly on AMD devices, but also on others, including embedded ones such as the Renesas V3M used in this study. CodeXL is not just useful for debugging, but also for code optimizations. Particularly, its *application timeline traces* provides information

on kernel execution and data transactions, while its *profiling mode* can collect detailed information from GPU performance counters. As a drawback, some features only apply to AMD GPU/APU devices.

Not all platforms provide specialized OpenCL implementations. An example are the embedded processing platforms from NVIDIA. The Jetson TX2 and the AGX Xavier do not even provide OpenCL 1.2. We explored the use of the portable open-source POCL framework [14] to target this hardware, using the experimental LLVM NVPTX backend [27] for code generation and the CUDA API for execution control. An alternative would have been the use of OpenCL-to-CUDA source-to-source compilation using SnuCL [18]. Note that POCL supports only OpenCL 1.2, and has a number of limitations, e.g., lack of atomics, image types, samplers. None of these missing features were crucial for coding our benchmarks, though.

Regardless of the OpenCL software stack used, the OpenCL code is considerably more verbose than either OpenMP and CUDA code, as will be shown in Chapter 7. This is due to being OpenCL being both relatively low-level (compared to OpenMP), and considerably more portable than CUDA. The combination leads to long source code stanzas in the host code (outside of the kernels) dedicated to discovering the characteristics of the execution platform at run-time. An advantage of this approach that in most cases, the same host code can be used, regardless of the kernel running, e.g., on a GPU or on an FPGA.

For the same reason, most OpenCL implementations (with the exception of the FPGA ones) actually rely on that dynamically discovered information for their actual compilation of OpenCL source code, which happens Just-in-Time (JIT) during execution. The JIT approach allows tailoring of the generated binary code to the specific platform the code will execute on. However, despite these advantages, the JIT compilation does carry an execution time cost, which will especially penalize shorter kernels. They would have to amortize the longer startup time (due to JIT compilation) across a shorter execution time. Thus, even for OpenCL, so-called Ahead-of-Time (AOC) compilation is becoming an important topic for both open-source (POCL) as well as proprietary (Codeplay ComputeCpp) OpenCL stacks. While this mode of operation would have been very interesting to include in the study, none of the tools had reached a sufficient maturity yet. In the future, however, possible benefits of the AOC usage model of OpenCL should be closely investigated! Note that OpenCL targeting FPGAs already exclusively uses the AOC model, as JIT-generation of FPGA hardware designs (which can take hours of tool run time) is generally not practical.

Also, some parts of the lengthy OpenCL boilerplate host-code can be shortened by using C++-based wrappers. In our experiments, though, this was not always successful. Not all of the C++ wrappers were working correctly, and we had to fall back on the original C-style host code in some cases.

6.3 CUDA

In contrast to OpenCL, CUDA is a very mature technology now. This is not only due to the CUDA being available much longer than OpenCL, but also due to NVIDIA being able to very closely match the software tools to the actual hardware. A concern for portability to non-NVIDIA platforms is not existent in that ecosystem. While this bears the risk of vendor lock-in, which may not be acceptable in all use-cases, the close intertwining of NVIDIA tools and NVIDIA hardware leads to very robust development and execution environments. We observed no major glitches or errors, neither in the two hardware platforms, nor in the development tools.

However, a number of rules of thumb have proven useful in working with NVIDIA technology. Note that the following comments are not intended as a summary of the very informative NVIDIA CUDA documentation [7]. They just share some key observations that were especially useful when implementing the benchmarks in CUDA.

Know your system, know your tools. Most optimizations or parallelization decisions rely on basic knowledge of the underlying hardware, or how the tools and libraries interact with them. For CUDA, this means that the developer must be aware of the fact, that the massive parallelism of the GPU can be utilized best when using *thousands* of threads, which is very different from programming multi-core CPUs. To feed these threads with data, and utilize the usually very large memory bandwidth of GPU-based systems, the programmer must know that memory is accessed in very wide words. This implies that the peak memory bandwidth is only reached, when every data item contained in the wide word fetched is actually used. This can be guaranteed when employing certain access patterns (e.g., when all threads access memory at sequential addresses). NVIDIA describes these details in their documentation under the keyword *memory coalescing*.

Quantity is not Everything Utilizing many threads is not always enough. For example, the basic algorithm used in points2image would allow to run *tens of thousands* of threads in parallel, but the computation within each is so small, that the speed-up is not as high as expected. This lower speed-up is also due

```
int px = int(imagepoint.x + 0.5);
int py = int(imagepoint.y + 0.5);
if(0 <= px && px < w && 0 <= py && py < h)
    {
        int pid = py * w + px;
        if(msg.distance[pid] == 0 ||
           msg.distance[pid]*100 > (point.data[2]))
            {
                msg.distance[pid] = float(point.data[2] * 100);
                msg.intensity[pid] = float(intensity);

                msg.max_y = py > msg.max_y ? py : msg.max_y;
                msg.min_y = py < msg.min_y ? py : msg.min_y;
            }
```

Listing 8: Excerpt from the original (serial) points2image-computation that is performed for each point in the point cloud. The race condition that can appear is that different points/thread access the *same* `pid` and want to modify the *same* distance/intensity values.

```
  int px = int(imagepoint.x + 0.5);
  int py = int(imagepoint.y + 0.5);
  if (0 <= px && px < w && 0 <= py && py < h)
    {
      pid = py * w + px;
      cm_point = point.data[2] * 100.0;
      oldvalue = msg_distance[pid];
      atomicCAS((int*)&msg_distance[pid], 0, __float_as_int(cm_point));
      atomicFloatMin(&msg_distance[pid], cm_point);
```

Listing 9: Excerpt from the CUDA points2image-computation that is performed for each point in the point cloud. The race condition from Listing 8 is now solved with atomic CAS and min operations.

to the embedded GPUs we examined having far fewer cores than desktop GPUs, and especially than the dedicated GPGPUs used in HPC settings.

Lock who is Talking Another recommendation is to look *very carefully* at the selected code for potential data races. They require synchronization, locking, or use of atomic operations to alleviate. points2image is a very good example: Getting a speed-up on the embedded GPUs was difficult enough. But that speed-up was reduced even further when atomic operations had to be added to ensure correct (deterministic) operation. Listing 8 and Listing 9 show how this was achieved.

The ndt_mapping benchmark also required inter-thread synchronization in the form of locking. But this was less costly than in points2image, as it was used less frequently in ndt_mapping.

It is also important to keep in mind, that the *order* for writing and reading data in *different* threads is completely arbitrary. E.g., a comparison may yield a certain result, but the value on which the comparison was based may have been changed by a *different* thread before the next command after the comparison in the *original* thread is even executed.

A practical problem that arose in the benchmark kernels was their use of floating point data types in sections that required atomic operations. As the CUDA library has only limited support for these operations, some of them had to be emulated using integer atomics and explicit data type conversions, such as the `__float_as_int` shown in Listing 9. An alternative approach, but most likely slower, would have been to use integer-based explicit locking around the floating-point computations.

Do not keep a Low Profile One of the most important tools to use during CUDA development is the profiler. Not only the NVIDIA `nvprof` profiler, which shows the runtime of GPU kernel invocations, but also a regular profiler, such as `gprof`, to identify the compute-intensive parts of the original serial implementation beforehand.

Print is not yet dead While NVIDIA provides `gdb` support for debugging, they also provide `printf`-support for kernel code. This is often much faster for debugging than interactively using `gdb`.

Porting Porting CUDA applications from one NVIDIA platform to another was very easy. Especially in our case of using the Jetson TX2 and AGX Xavier, which both have very similar memory systems. The porting effort mainly consisted of tuning specific execution parameters (e.g., the number of threads to run in parallel) based on trial runs.

When switching to a less similar GPU platform (e.g., from the embedded platforms to a desktop/server GPU), slightly more work is necessary: In general, these platforms do *not* share the physical memory between the host CPU and the GPU. Instead, they use physically separate memories (which may be located in the same virtual address space, though), and require *copy* operations between the memories for efficient operation. As these copy operations can be slow (e.g., small transfers via a PCI Express bus have a relatively high overhead), some developer effort should go into optimizing their use.

Know your Hardware The importance of this aspect cannot be overstated, which is why it is being highlighted here again. Even an experienced can overlook seemingly minor hardware specifics, which may have a major impact

in the long run. The concrete example motivating this statement is on the use of power management profiles in the NVIDIA embedded platforms.
Modern GPUs (and CPUs, too) support many different power profiles. These specify when to increase or decrease the clock frequency, or raise or lower (or even turn off) supply voltages to certain parts of the system. The aim of these profiles is to save power, or meet certain performance goals.
In our study, the initial CUDA implementations had a somewhat disappointing performance. This changed only once we recognized that the default power profile for the NVIDIA embedded platforms aims for *low power*. Only after manually switching to a performance-optimized profile using the `nvpmodel` tool (which is not part of the CUDA SDK) the expected speed-ups were achieved.
Note that the GPU benefits more from the performance-profile than the CPU, thus the higher speed-ups of the parallel version over the sequential baseline. Also, we recommend *against* using a demand-driven dynamic power management. While this is supported in the hardware, it interacts poorly with the relatively short benchmark kernels as it leads to an oscillating behavior: The clock frequency is low in idle state, then the short test kernel starts (and is so fast that it finishes almost immediately), the GPU recognizes the past demand for computation, the clock frequency is raised, but since test kernel has already finished, the GPU recognizes the drop in compute demand and lowers the clock frequency again. This leads to the kernel always executing slowly at the low clock frequency.
The impact of the memory being physically shared between CPU and GPU in the embedded platforms is also hard to overstate: The relatively short benchmark kernels would suffer significantly when executed on a desktop/server-class GPU, due to the relatively high overhead of the explicit data copies required. On the embedded platforms, the switchover between CPU and GPU execution is seamless (no copies required) and carries only the overhead of actually starting the GPU threads.

Do not expect much C++ support Even while the CUDA compiler understands some C++, it does not support the STL, exceptions, or new/delete memory allocation, three of the most frequently used features of C++.
Thus, for one of the examples, the largest effort when porting the baseline sequential C++ code to CUDA involved the removal/rewriting of STL calls. When developing code intended for later CUDA acceleration, it might be beneficial to avoid using the STL in the first place.

Better Algorithms before better Hardware While it would have been possible to reimplement the unsupported STL functionality in a manner compatible

with CUDA, it proved to be more beneficial to instead look at the concrete requirements of the algorithm during parallel execution. Stepping away from the STL abstractions, new data structures were introduced that allowed a far better parallelization.
In two cases, these new data structures could also be successfully backported to the sequential code versions, and led to significant speedups there. In Chapter 7, we compare our parallelized code against both the original as well as the improved sequential baselines.

Combining Acceleration Approaches It would have been interesting to extend this study to combining different parallelization approaches. On the NVIDIA embedded platforms, both OpenMP and CUDA could have been employed simultaneously. OpenMP to use multiple CPU cores in parallel to the CUDA-programmed GPU. The results of such a combination are difficult to estimate without actually implementing it, as interference effects (e.g., CPUs and GPU competing for bandwidth to the physically shared memory) might negate some of the speed-ups achievable by the higher degree of parallelism.

Evaluation 7

The first aspect we want to investigate in our quantitative evaluation, is the *programmer productivity*. As described in Section 2.3, we employ a simultaneous tracking of working hours spent versus achieved performance to assess programmer productivity.

Next to the total amount of work spent on the parallel and heterogenous implementation of a benchmark, this metric also provides information about how quickly different levels of performance can be achieved with the various programming models investigated in this survey.

Also, to reiterate from Section 2.3, the key focus of this study are the *methodological aspects* of programming parallel heterogeneous embedded computing platforms, *not* benchmarking the implementations for absolute performance. Due to our use of real, but completely generic (non-platform optimized) algorithms extracted from Autoware, as well as the highly diverse target hardware platforms, the raw performance numbers are not comparable. They are not indicative of either the performance of the platforms, nor of the programming tools. Given these caveats and purely for reference, we show them in Appendix A.

7.1 Points-to-Image Benchmark

The results of the simultaneous tracking of working hours versus achieved performance for the smallest (in terms of code size) benchmark points2image can be found in Figure 7.1.

7.1.1 OpenMP Implementation

For OpenMP development on the Jetson TX2 platform, the plot contains only a single data-point. This is because for the parallel implementation with OpenMP, it was sufficient to parallelize only a single top-level loop, which took roughly one hour. So for this rather small example, parallelization with OpenMP is very easy and yields reasonable performance with very little effort.

7.1.2 CUDA Implementation

The parallelization with CUDA requires some initial effort to restructure the application and partition the execution between host and accelerator device. During this restructuring, the application is not executable. Thus, the plotted line starts only after about eight hours of work.

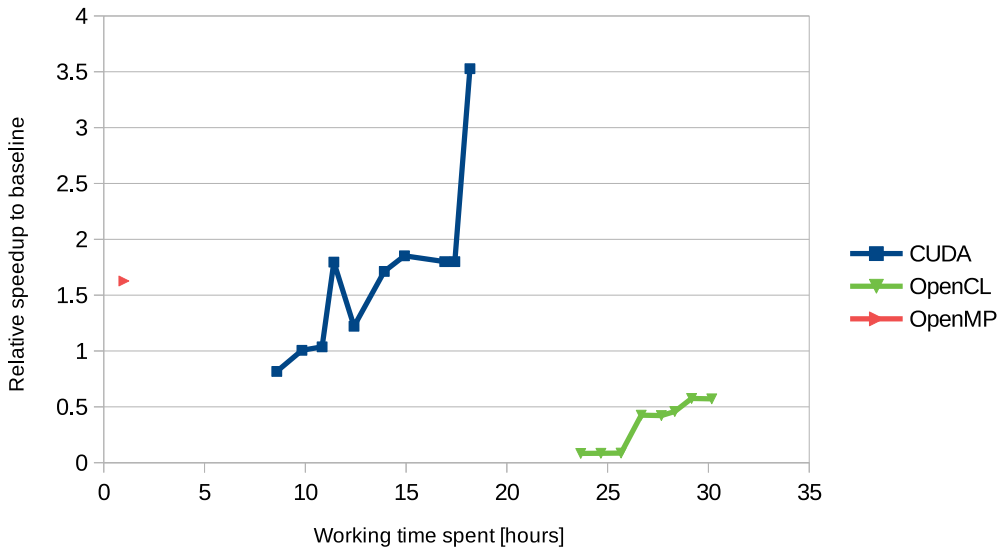


Figure 7.1.: Programmer productivity for benchmark points2image.

After this initial phase, a number of development iterations is spent on parallelizing the application, mainly on figuring out correct placement of data in shared memory and scheduling explicit data-transfers. The final peak in performance is caused by the activation of a faster clocking mode on the Nvidia Jetson TX2 platform. After this, the relative performance gain is higher for CUDA than for OpenMP, due to the fact that with higher number of threads on the GPU, more points can be processed in parallel (cf. Section 6.1).

7.1.3 OpenCL Implementation

For OpenCL, the initial effort for restructuring the application is even higher, as our plotted line shows. This is partially due to platform instability issues we experienced with the platform OpenCL development initially started on. Because of the immaturity of the OpenCL stack for the platform, development had to be moved to the Jetson TX2 and the platform was excluded from consideration for this report. This move caused some additional delay in the early phase of the implementation.

As the application becomes executable after restructuring for OpenCL, performance increases step by step. At this point of our analysis, it is worth highlighting that the hotspot of the points2image benchmark consists of a code region where intensity and distance values from different loop iterations are updated. As this region is characterized by loop-carried dependencies, our OpenCL version initially implemented this part on the host for the sake of simplicity, but with a corresponding performance penalty.

As a refined attempt to further increase performance, we also offloaded this region to the accelerator and used atomic operations to resolve the simultaneous memory accesses.
However, in order to compute correctly after introducing atomics, this benchmark requires additional device-level synchronization, invoked from within the kernel code, similarly to that implemented in the CUDA version using the API call `__threadfence_system()`.
This call provides memory consistency by ensuring that *all* writes to *all* memory regions issued by the calling thread are completed. Their results are visible to *all* threads in the device, *all* host threads, and *all* threads in peer devices [7].
To the best of our knowledge, there is no equivalent call in OpenCL *1.2*, and therefore the usage of atomics, and thus a possible performance gain, had to be dropped. For this reason, the OpenCL version was not able to catch up with the other two models in terms of performance.
Note that OpenCL *2.x* actually does provide such a call, but OpenCL 2.x was not available on the two OpenCL platforms considered (Jetson TX2 and Renesas V3M).

7.2 Euclidean Clustering Benchmark

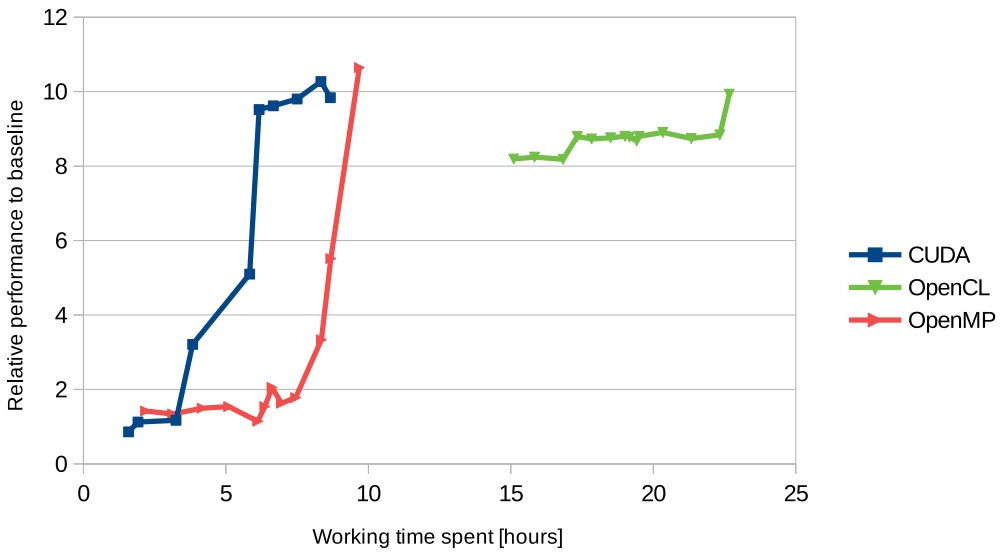


Figure 7.2.: Programmer productivity for benchmark euclidean_clustering.

For euclidean_clustering, we show two plots in Figure 7.2 and Figure 7.3, with speedups computed relative to two different serial baseline implementations. One baseline implementation is the result of the extraction of the original

code from Autoware, the second in Figure 7.3 was obtained by backporting some optimizations to the serial code (see Section 4.2.2 for more details).

7.2.1 OpenMP Implementation

For the parallelization with OpenMP (this time using the Nvidia Jetson TX2), a number of different parallelization strategies were tried to determine the best approach, which explains the rather flat section at the beginning of the plotted line. After identification of the best strategy, this strategy could be realized very quickly within just three development iterations, resulting in the steep performance increase achieved in the final OpenMP implementation.

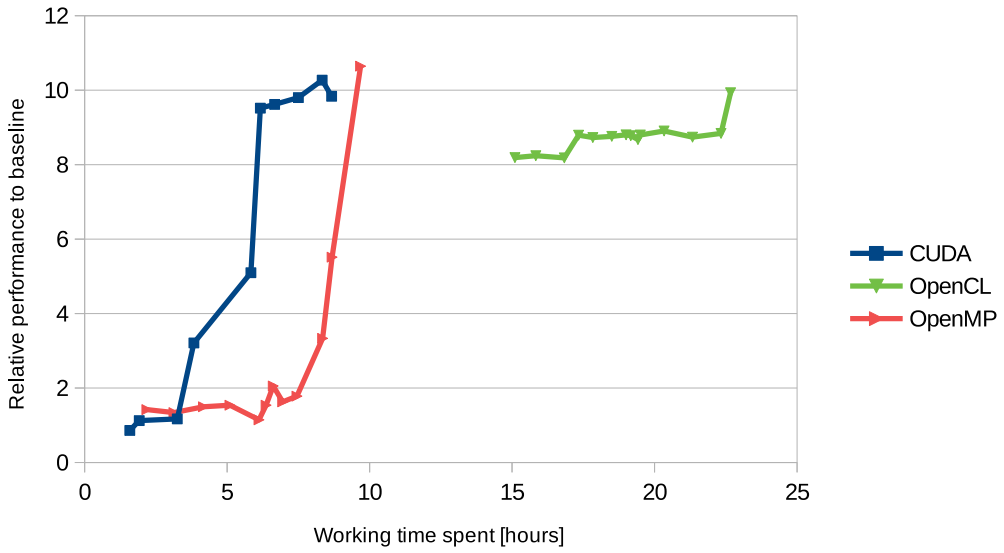


Figure 7.3.: Programmer productivity for benchmark euclidean_clustering after backporting optimizations to the serial baseline. Note the different scaling on the y-axis compared to Figure 7.2, as they show measurements relative to two different serial baseline implementations.

7.2.2 CUDA Implementation

Compared to the previous benchmark, the parallelization with CUDA on the Nvidia Jetson TX2 does not require that many invasive changes to the serial implementation, therefore the initial, non-assessable effort is smaller. The experimentation phase to determine a good parallelization approach is shorter than for OpenMP, and, once found, the strategy is successfully realized in roughly three hours. Afterwards, the parallelization is further refined, resulting in a performance similar to OpenMP.

7.2.3 OpenCL Implementation

The Jetson TX2 was also used as a target for OpenCL development using the pocl system. Its use to implement the euclidean_clustering benchmark requires a similar (actually slightly less) initial effort to that for the points2image benchmark. As before, that initial effort is spent to restructure and partition the application. Development focused on a single successful parallelization approach right from the beginning. This strategy is iteratively improved over time and eventually results in a final performance on par with CUDA and only slightly lower than OpenMP.

7.3 NDT-Mapping Benchmark

The development time vs. performance plot for our last benchmark ndt_mapping is shown in Figure 7.4.

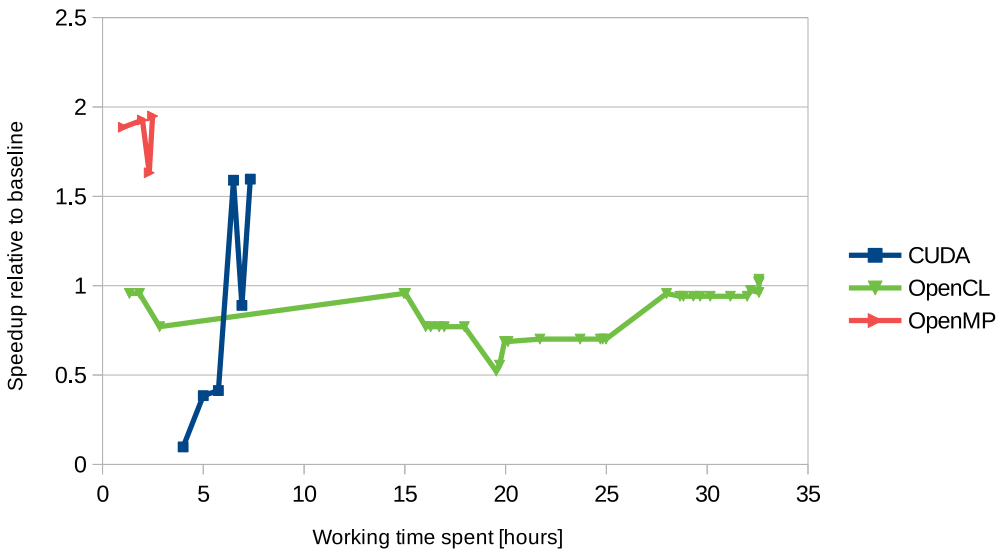


Figure 7.4.: Programmer productivity for benchmark ndt_mapping.

7.3.1 OpenMP Implementation

OpenMP again allows for a very fast parallelization within less than three hours and practically no initial effort for code restructuring. The transformation of some data-parallel loops already yields good performance. An additional experiment to achieve even further performance by employing atomics does not realize further gains, as the implementation of the atomics in the OpenMP run-time is relatively slow on the Nvidia Jetson TX2 platform. This leads to a temporarily lowered performance. After these changes have been reverted, some further optimizations allow the final implementation to

achieve a speedup of close to 2x even compared to the improved sequential baseline.

7.3.2 CUDA Implementation

CUDA again requires some initial effort for code restructuring before the actual parallelization. This is realized in three development iterations and leads to a steep increase in performance on the Nvidia Jetson TX2. Further performance gains by moving computations back to the CPU were attempted, but had to be reverted. The final version of the CUDA implementation achieves a performance similar to that of the OpenMP version.

7.3.3 OpenCL Implementation

For this last benchmark, OpenCL requires significantly less initial effort compared to the previous two benchmarks, because the multiple kernels necessary for offloading with OpenCL are independent from each other and can be implemented one after the other. No complex synchronization mechanisms (such as the fence API call that would have helped the points2image benchmark) were required.

However, we were not able to run the code on either of the embedded OpenCL platforms: The Renesas V3M did not support the double-precision arithmetic required by the algorithm, and the Jetson TX2 with pocldid not support the 64-bit atomics required for efficient parallelization.

Therefore, we chose an alternate solution. For this benchmark, we considered a full desktop-class PC with an Intel Core i5 CPU and an AMD Vega-56 GPU. A mature OpenCL 2.0 implementation is available for that platform that operates flawlessly and provides both of the features lacking on the embedded platforms. We still hold true to our initial plan and show the performance gains relative to the sequential version of the code on the platform CPU (here: one core of the Core i5 CPU) versus the platform GPU (here: the Vega-56 PCI Express card).

However, even after a comparatively long phase of experiments with the four different kernels making up the ndt_mapping benchmark, the OpenCL code is only roughly on par with the original *serial* implementation, and significantly slower than OpenMP and CUDA.

Despite the combination of Core i5 CPU and Vega-56 GPU being much more powerful than the embedded platforms we looked at, it does have a crucial disadvantage: It does require explicit copies of input and output data to the GPU via the relatively slow Gen3x16 PCI Express bus. In contrast, both the Renesas V3M as well as the Jetson TX2, however, have *physically shared*

memory that is accessible to both the CPU and GPU cores in these SoCs *without* the need for data copies.
But the discrete Core i5 / Vega-56 platform suffers from these data transfer overheads. Profiling using the AMD CodeXL analysis tool shows that the data transfer time is initially similar to that of the actual computation time. But the data transfers actually become larger and slower over time, as the ndt_mapping algorithm accumulates more data over its entire run time, which has to be transferred back and forth between CPU and GPU for processing.

7.4 Discussion of Results

7.4.1 Development Effort

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

OpenMP Implementation

Across all three benchmarks investigated in our implementation survey, OpenMP typically requires the least effort to parallelize an application. As already discussed in Section 6.1, 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.

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. 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.

OpenCL Implementation

For the majority of the benchmarks, OpenCL requires a large amount of 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. The comparably 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.

7.4.2 Maintainability

The ranking with regard to the required development effort also correlates with the results that we get from our metrics for *maintainability*. The first metric we investigate is the code expansion, which provides information about how much the total code volume for the benchmarks increases during parallelization. Figure 7.5 shows the increase in code volume relative to the original serial baseline implementation.

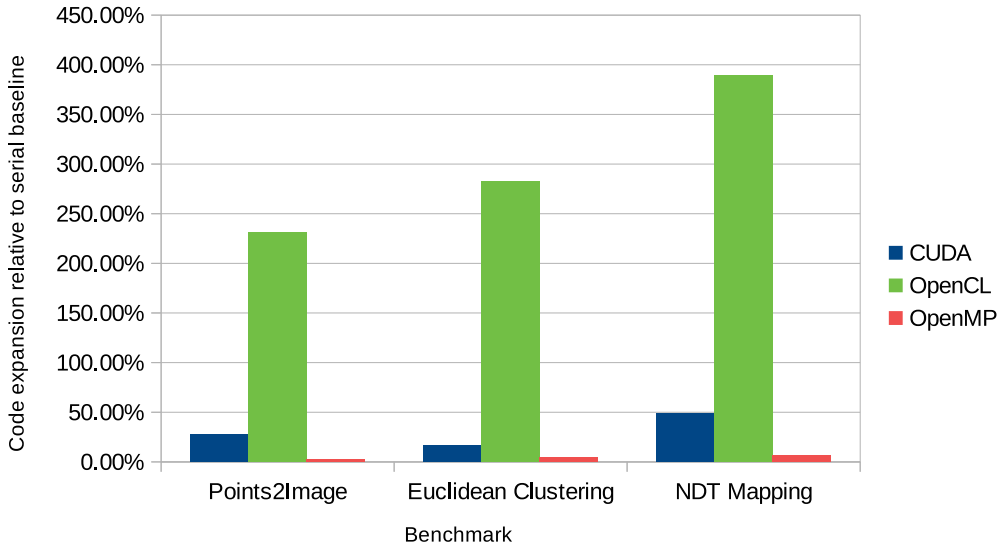


Figure 7.5.: Increase in code volume relative to serial baseline.

OpenMP causes the least increase in code size, the lines added to include OpenMP compiler directives make up less than 10% of the original code's size. For CUDA, we see a moderate increase in the range between 15 and 50%. The restructuring and partitioning into multiple files (separate host and device code) necessary for OpenCL, together with the amount of boilerplate code that needs to be included on the host side (see Section 6.2), results in a large code size expansion for OpenCL. Compared to the baseline implementation, the total code volume in our study has been observed to at least double.
As discussed in Section 2.3, the code expansion metric does not consider *in-place* changes to the code. Therefore, we also evaluated the total number of line changes using the `git diff` tool. The results for this evaluation are shown in Figure 7.6, which lists the total number of line changes (added or deleted) in relation to the LoC of the original, serial implementation.

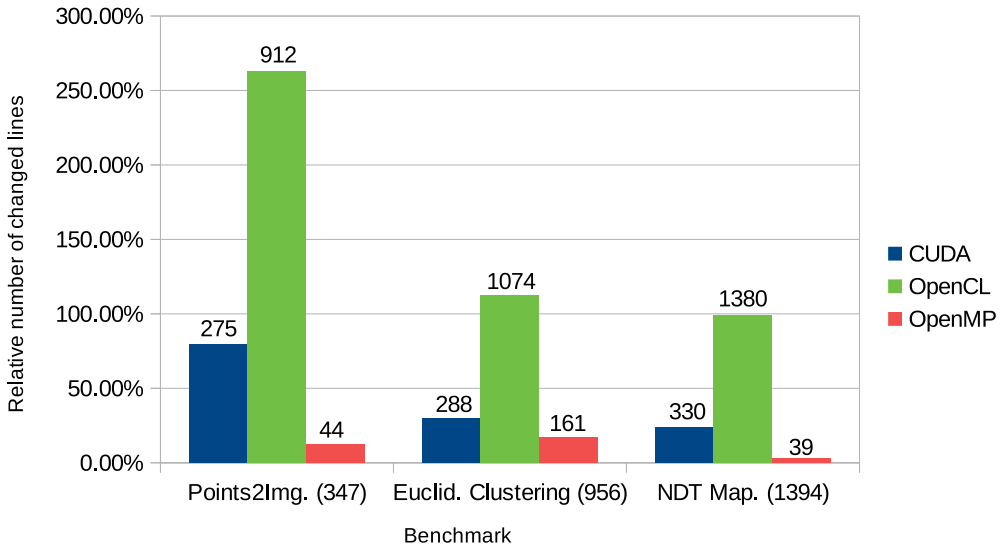


Figure 7.6.: Number of changed lines relative to serial baseline. The numbers in the parentheses give the LoC of the serial implementation.

Because OpenMP allows to reuse the serial implementation almost unchanged in most cases, and only requires to add parallel semantics through compiler directives, the number of changes is relative 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 leads to sweeping changes in the code base, ranging from 99% to 263%.

While the previous two metrics provide information about how much the code was changed, they do not indicate the complexity that was added through these changes. To assess this additional complexity introduced by the use of parallel programming models into an application's code, we use the *Complexity Count* metric we defined in Section 2.3. The counts for all benchmarks and models are given in Table 7.1.

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

Table 7.1.: Complexity count (see Section 2.3 for definition & examples).

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, as already discussed in Section 6.2 and also visible in the two previous plots, 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 genuine `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 made implicit, resulting in a notably smaller complexity count.

Note that many of these weaknesses of the original OpenCL approach have been addressed with the newer SYCL language, which can be considered to be a direct successor to OpenCL. Listing 11 shows an excerpt of a SYCL implementation of euclidean_clustering that demonstrates the conciseness of the more modern, but not yet established approach.

7.4.3 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.

OpenMP Implementation

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. The adaptation of the execution to the new platform is mostly left to the compiler, and only rarely influenced by giving platform-specific parameters in the code. This hands-off approach often results in the new platform reaching similar

speedups (relative to the serial implementation) as on the platform the OpenMP code was originally developed on.

CUDA Implementation

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. A small number of platform-specific parameters, e.g., the number of threads employed, needs to be adapted, but still, the porting of each benchmark took only about 30 minutes to complete.

OpenCL Implementation

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 our three benchmarks, only the the euclidean_clustering kernel was not affected by this diversity in tools and hardware platform capabilities. Here, the porting effort was similar to CUDA and OpenMP.
But the other two benchmarks 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. That effort includes not just porting and debugging the kernel itself, but also modifying the kernel-specific validation suite and generating new golden output data.

7.5 Excursus: Targeting FPGAs with OpenCL

Porting OpenCL applications originally written for devices with multiple compute units (e.g. GPUs) to platforms with a completely different computational architecture, such as the highly customizable FPGA logic on the Xilinx Zynq SoC, has special challenges.
FPGA vendors often have to extend OpenCL with their own, vendor-specific extensions to adapt the language to the underlying architecture of the FPGA, and also require a *completely different* coding style than for GPUs in order

to leverage the pipeline parallelism on FPGAs. Thus, porting an OpenCL application from GPU to FPGA might require to rewrite large fractions of code.

We undertook that step by implementing the points2image benchmark in the "single work-item"-style (see Section 6.2) required for FPGA hardware synthesis. Starting with the original OpenCL code targeting the GPU, the FPGA coding itself required about 25 hours.

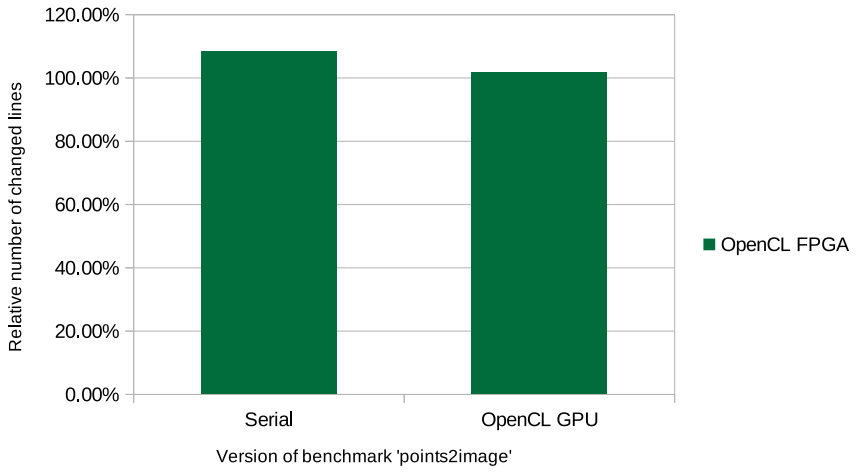


Figure 7.7.: Number of changed lines for OpenCL on FPGA, relative to serial baseline and OpenCL implementation for GPUs.

However, looking at the number of changed lines in Figure 7.7 even relative to the GPU code, it is clear that the FPGA implementation is not just a "port" of the GPU version, but a completely new coding artifact.

This development time for the FPGA includes not just the actual coding time, but also the relatively long compilation times typical for FPGA targets. For this benchmark, it took an average of 20 minutes to "compile" (build in FPGA parlance) the OpenCL kernel into FPGA logic. This is one to two orders of magnitude slower than compiling for the GPU.

In order to minimize the number of times a FPGA building process was launched, we thus utilized the Xilinx SDSoC emulator, instead of going to real hardware. Using the emulator on average required two minutes to load its development environment.

While they have matured considerably in recent years, OpenCL development tools on FPGAs still have instabilities. As an example, the initial SDSoC versions we used contained a known (to the vendor) out-of-memory bug that made it impossible to run the full number of test cases for the points2image benchmark. This was only fixed with the release of SDSoC 2018.3 late in the project.

With their reliance on single-work item OpenCL kernels to build deep pipelines, the tools are also susceptible to the presence of potentially concurrent memory accesses in the code. When these accesses cannot be statically disambiguated (as was the case for points2image), hardware synthesis can no longer create an efficient pipeline, and falls back to realizing slow sequential computations. Combined with the lack of thread-level parallelism on FPGAs, the performance of the FPGA realization of the points2image benchmark is not competitive with our other implementations.

Thus, targeting FPGAs has to be considered carefully on a case-by-case basis. While related work has shown the potential for significant speedups at low energy consumption (often even exceeding that of GPUs!), the actually achievable quality of results is highly dependent on the concrete code and its interaction with current versions of high-level hardware synthesis tools.

Conclusion and Strategic Advice

8

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. But we can state which characteristics are desirable in a programming model for it to be a good match to the demands of the automotive industry. Also, the results from this survey cannot serve as the *sole* base for determining the best programming model for a particular purpose. Given the time constraints, the scope of this survey had to be limited, in particular with regard to the number of benchmarks we were able to investigate. 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 highest-level (most abstract) programming model that we have been investigating is **OpenMP**. OpenMP relies heavily on high-level compiler directives to add parallel semantics to otherwise serial code, and to guide the compiler in its effort to efficiently parallelize the application. It thus allows for a very good programmer productivity and to realize a parallelized implementation in relatively short periods of time. Additionally, with OpenMP it is possible to develop the core functionality of the algorithm as serial code, which is easier to test and debug. In addition, existing serial implementations can be reused without large up-front effort to restructure the application. Despite the fact that the parallelization is performed at a relatively high-level with OpenMP, the resulting implementations offer competitive performance. This is achieved by the interaction of compiler and the OpenMP run-time. On the other hand, the reliance of OpenMP on the compiler for parallelization also has some downsides: If the compiler does not yet (fully) support the desired constructs, the constructs become practically unusable and cannot be emulated using lower-level mechanisms. This was the case for the OpenMP device offloading features, which could not yet be examined in this study.

CUDA strikes a balance between high-level abstractions and explicit parallelization. The latter is targeted towards exploiting the specific execution platform. In combination, this allows reasonable programmer productivity and good performance. CUDA takes a completely different approach

than OpenMP by requiring restructuring of the application to some extent. However, the complexity is still manageable, because far less host code is required, e.g., in comparison to OpenCL. CUDA of course benefits from the continued large investment by Nvidia, the company that invented CUDA and still develops it to this day. 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-lockin. Alternatives for moving CUDA outside the Nvidia ecosystem (e.g., AMD HIP/ROCm) are only slowly appearing and as yet unproven.

In contrast to CUDA, implementations with **OpenCL** require much more host code and 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. In our experience, OpenCL's biggest strength is at the same time also its biggest weakness: As an open standard, OpenCL is available from many different vendors on a large range of platforms. But the vendors support different versions of the standard, with different support for optional features, and extended with their own, proprietary vendor-specific extensions. This wide spectrum of choices makes the use of OpenCL across different platforms often more difficult than anticipated. We experienced this in our study, e.g., with regard to the use of atomics or double-precision floating point arithmetic. As a result, two applications, though both are written in OpenCL, can feel like having been developed in two completely different programming languages, similar to what we experienced when moving one of our benchmarks from GPU to FPGA.

Based on our findings, we give the following recommendations to for use of parallel heterogeneous computing in the automotive domain.

8.1 Recommendations on using OpenMP

OpenMP as a programming model with high-level abstractions and compiler directives has proven to able to deliver programmer productivity and good performance at the same time. Despite its roots in x86/POWER-dominated HPC, this held true even for its use on smaller ARM-based automotive platforms. We expect these platforms to profit from the very active development of OpenMP, with the standard having been revved to version 5.0 in November 2018. Further improvements could derive from the intensive exchange of ideas between the OpenMP and OpenACC communities, with the latter pushing for even higher abstraction levels than OpenMP. With the excellent results, in terms of productivity vs development effort, we achieved using

OpenMP in this study, we would recommend that the automotive industry has its own experts participate in the, still somewhat HPC-focused, OpenMP community to raise awareness for the characteristics of automotive workloads (e.g., smaller kernels, more impacted by thread launch overheads).

8.2 Recommendations on using CUDA

Although CUDA is more explicit in its parallelization than OpenMP or OpenACC, it still has provided reasonable programmer productivity. To exploit the advantages of the CUDA programming style and overcome the risk of a vendor-lockin at the same time, the automotive industry could weigh in on and support efforts that try to make features of the CUDA language available on a wider range of platforms through an open standard, such as HIP[1].

```
1   dim3 threaddim(THREADS);
2   dim3 blockdim((vector_size+THREADS-1)/THREADS);
3   hipMalloc(&a, vector_size * sizeof(float));
4   hipLaunchKernel(vector_add, threaddim, blockdim, a, b, c);
5   hipDeviceSynchronize();
```

Listing 10: HIP translation of the CUDA example in Listing 1, Page 14.

As an example, Listing 10 shows the CUDA code from Listing 1 translated to HIP. The only changes are some minor syntactical edits, and a slightly different structure for the kernel invocation. Through the use of HIP, the industry could benefit from the existing knowledge of experienced CUDA developers and the productivity provided by the underlying programming model, while avoiding the risk of a vendor lock-in.

8.3 Recommendations on using OpenCL

Much of the up-front effort to implement an application in OpenCL was due to the large amount of host code necessary, and the required partitioning between host and device code files. With SYCL as a spiritual successor to OpenCL, the Khronos Group provides a modern, open and royalty-free standard designed to overcome these limitations of OpenCL.

SYCL is currently receiving increasing attention and the ecosystem is growing. To the best of our knowledge, in addition to the four existing SYCL implementations (ComputeCpp[3], triSYCL[36], hipSYCL[11], sycl-gtx[34]), Intel recently has announced its plan to contribute support for SYCL into Clang/LLVM[13].

To put SYCL in context of this study, some details need to be explained. SYCL is a cross-platform abstraction layer that builds on the underlying concepts,

portability and efficiency of OpenCL. It enables code for heterogeneous processors to be written in a *single-source* style using only standard C++ constructs. With SYCL, developers can program at a higher level than OpenCL C/C++. In this way, SYCL hides a large number of OpenCL complexities, and thus substantially reduces the amount of host-side code needed over traditional OpenCL. This approach allows developers to take advantage of highly parallel hardware using standard C++ code, but also provides access to lower-level code through integration with OpenCL, C/C++ libraries, and even other frameworks such as OpenMP [33].

The single-source programming provided by SYCL enables both host and kernel code to be contained in the same C++ source file, in a type-safe way and with the simplicity of a cross-platform asynchronous task graph. Some examples of the C++ features supported in SYCL are templates, classes, operator overloading, static polymorphism, and lambdas.

For a preliminary assessment, we have ported our euclidean_clustering benchmark to SYCL. We observed that much of the typical OpenCL boilerplate code (i.e. platform discovery, device query, etc) could be replaced by fewer calls that define SYCL objects such as a selector (`cl::sycl::default_selector`) and a queue (`cl::sycl::queue`) at a more abstract level.

Regarding the kernel itself, the legibility of its corresponding code was notably improved with the SYCL convention of submitting computations to a queue, here `myQueue` shown in Listing 11, for execution. This submission accepts a command group as first parameter (represented by its handler `cgh`), which is a way of encapsulating a device-side operation and *all* its data dependencies in a single object. Then, it defines accessors to input and output data, i.e. SYCL objects that point to data placed in global memory.

The definition of accessors and other objects (global & local sizes and offset) is simplified with the `auto` datatype. Moreover, the usage of lambdas (`myKernel`) allowed us to further reduce verboseness. Namely, there is no more need to explicitly set kernel arguments like in OpenCL (`setArg()`) since their values can be captured with the lambda capture-clause (`[=]`). Finally, the actual kernel execution happens after invoking the `parallel_for` API. Quantitatively, we found a reduction in our *complexity count* metric (defined in Section 2.3) from 120 in OpenCL down to 70 in SYCL.

For portable programming without the risk of vendor lock-in, SYCL is a very promising contender. Since the required software layer can be built directly on top of existing OpenCL stacks (an approach actually used by Codeplay for layering their ComputeCpp SYCL atop their ComputeAorta OpenCL product), we expect a more intensive use in the mid-term (15-18 months).

```
myQueue.submit( [&](cl::sycl::handler& cgh) {
/* Defining accessors in read and write modes
 * as the kernel reads from and writes to
 * buffers "buff_cloud" and "buff_sqr_distances". */
 auto acc_cloud = buff_cloud.get_access
                  <cl::sycl::access::mode::read>(cgh);
 auto acc_distances = buff_distances.get_access
                      <cl::sycl::access::mode::write>(cgh);

/* Defining global, local sizes, and offset. */
 auto myRange = cl::sycl::nd_range<1>(
                    cl::sycl::range<1>(global_size),
                    cl::sycl::range<1>(local_size),
                    cl::sycl::id<1>(offset)
                );

/* Constructing the lambda outside of the parallel_for call.
 * For this parallel_for, the lambda is required to take
 * a single parameter: an item<N> of the same dimensionality
 * as the nd_range - in this case one. */
 auto myKernel = ([=](cl::sycl::nd_item<1> item) {
  int n = cloud_size;

 /* Retrieving the global id as an id<1>. */
  int j = item.get_global_id(0);

  if (j < cloud_size) {
   for (int i = 0; i < n; i++) {
     float dx = acc_cloud[i].x - acc_cloud[j].x;
     float dy = acc_cloud[i].y - acc_cloud[j].y;
     float dz = acc_cloud[i].z - acc_cloud[j].z;
     int array_index = i*n + j;
     acc_distances[array_index] =
            ((dx*dx + dy*dy + dz*dz) <= tol);
   }
 });

/* Calling the parallel_for() API with two parameters:
 * the nd_range and the lambda constructed above. */
 cgh.parallel_for<class initRadiusSearch>(myRange, myKernel);
});
```

Listing 11: Snippet of the SYCL implementation of the euclidean_clustering benchmark: The the `initRadiusSearch` kernel is submitted to a queue for execution.

8.4 Future use of Domain-Specific Languages (DSLs)

The focus of this study was the evaluation of *existing* off-the-shelf solutions for programming heterogeneous parallel computing systems for automotive applications.

However, another approach of achieving high productivity in the development of high performance solutions exists in the use of *Domain-Specific Languages*. These allow the domain experts to formulate solutions at the *native* abstraction level of the domain. The mapping to the execution platform would then not be performed by platform experts, but by using automated tools that encapsulate the knowledge of the platform experts.

The state-of-the-art in programming tools has advanced in recent years to a degree that this no longer science fiction, or doomed to failure like the search for "magic" general auto-parallelizing compilers of the late 1980s.

Using modern software frameworks and tools, it has become possible to quickly implement such tool-chains for focused DSLs. The term "focused" means that these languages are really only applicable in their domain (e.g., convex optimization problems) and do *not* aim to be general (e.g., Turing-complete) programming languages.

Research at the *Embedded Systems and Applications* group of TU Darmstadt has demonstrated the feasibility of using such DSLs for programming parallel CPUs, GPUs, and FPGAs in a number of domains (network security, convex optimizations for collision avoidance in trajectory planning, generating accelerated simulation models for biomedical problems, highly efficient computations for geometric problems etc.)

With this tight focus, the mapping of the DSL abstractions to efficient execution mechanisms on the parallel heterogeneous execution platforms becomes far simpler, than in general parallelizing compilers: Many difficulties, e.g., pointer disambiguation or iteration space analysis, do not occur at all, or only at far reduced complexities.

While the actual realization of robust tool flows and frameworks to support this development approach is the subject of active research (e.g., at Stanford, ETH Zurich, and TU Darmstadt), no fundamental roadblocks to its use have yet been found.

In one specific area, namely that of accelerating the inference step of machine learning scenarios, the DSL-centric approach is already in common use: Trained networks are expressed in a stylized sequence of API calls (e.g., Cafe, TensorFlow etc.), or in DSL-like file formats (e.g., ONNX), and then submitted

to back-ends for actual compilation to hardware targets like GPUs, FPGAs, or dedicated ML accelerators.
Given the excellent results already shown in this approach, we expect it to become much more important in the future!

Acknowledgments

The authors would like to thank Xilinx for supporting this work by donations of hard- and software, as well as Renesas and Codeplay for their help with the acquisition of the V3M and the timely support for using their programming tools.

Bibliography

[1] *AMD HIP*. https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/. Accessed: 2019-02-05 (cit. on p. 75).

[2] *Autoware Developers Guide*. https://github.com/CPFL/Autoware/wiki/Overview. Accessed: 2018-12-20 (cit. on p. 34).

[3] *Codeplay ComputeCpp*. https://www.codeplay.com/products/computesuite/computecpp. Accessed: 2019-02-05 (cit. on p. 75).

[4] *Codeplay R-Car Guide*. https://developer.codeplay.com/computeaortace/latest/r-car-guide. Accessed: 2019-02-05 (cit. on p. 44).

[5] *CodeXL: a comprehensive tool suite that enables developers to harness the benefits of CPUs, GPUs and APUs*. https://github.com/GPUOpen-Tools/CodeXL. Accessed: 2019-02-05 (cit. on p. 52).

[6] *Continental Press Release*. https://www.continental-corporation.com/en/press/press-releases/ces-2018-117894. Accessed: 2019-02-05 (cit. on pp. 1, 8).

[7] *CUDA C programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2019-02-05 (cit. on pp. 27, 54, 61).

[8] Dustin Franklin. *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/. Accessed: 2019-02-05. Mar. 2017 (cit. on p. 42).

[9] Maurice Howard Halstead et al. *Elements of software science*. Vol. 7. Elsevier New York, 1977 (cit. on p. 13).

[10] *Himeno Benchmark*. http://accc.riken.jp/en/supercom/documents/himenobmt/. Accessed: 2018-12-20 (cit. on p. 42).

[11] *hipSYCL*. https://gpuopen.com/compute-product/hipsycl/. Accessed: 2019-02-05 (cit. on p. 75).

[12] L. Hochstein, J. Carver, F. Shull, et al. „Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers“. In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. Nov. 2005, pp. 35–35 (cit. on pp. 8, 13).

[13] *Intel announces SYCL support - LLVM mailing-list*. http://lists.llvm.org/pipermail/cfe-dev/2019-January/060811.html. Accessed: 2019-02-05 (cit. on p. 75).

[14] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, et al. „pocl: A Performance-Portable OpenCL Implementation“. In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 752–785 (cit. on pp. 29, 51, 53).

[15] David R. Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015 (cit. on p. 25).

[16] S. Kato, E. Takeuchi, Y. Ishiguro, et al. „An Open Approach to Autonomous Vehicles“. In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68 (cit. on pp. 7, 33).

[17] *Khronos Group*. https://www.khronos.org. Accessed: 2018-12-20 (cit. on p. 24).

[18] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. „Bridging OpenCL and CUDA: A Comparative Analysis and Translation“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 82:1–82:12 (cit. on pp. 29, 32, 53).

[19] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. „CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures“. In: *17th IEEE International Conference on Parallel and Distributed Systems*. Tainan, Taiwan, Dec. 2011 (cit. on p. 32).

[20] Thomas J McCabe. „A complexity measure“. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. on p. 13).

[21] Zaur Molotnikov, Konstantin Schorp, Vincent Aravantinos, and Bernhard Schätz. *FAT-Schriftenreihe 287 - Future Programming Paradigms in the Automotive Industry*. Tech. rep. 2016 (cit. on pp. 1, 6, 7, 18).

[22] *Nvidia Drive platform*. https://developer.nvidia.com/drive. Accessed: 2019-02-05 (cit. on p. 1).

[23] *Oclgrind: an OpenCL device simulator and debugger*. https://github.com/jrprice/Oclgrind. Accessed: 2019-02-05 (cit. on p. 52).

[24] *OpenCL Specification*. https://www.khronos.org/opencl. Accessed: 2018-12-20 (cit. on p. 24).

[25] OpenMP Architecture Review Board., ed. *OpenMP Application Programming Interface - OpenMP Standard 4.5*. Nov. 2015 (cit. on pp. 19, 20, 22, 23).

[26] R. van der Pas, E. Stotzer, and C. Terboven. *Using OpenMP - The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation. MIT Press, 2017 (cit. on pp. 21, 23).

[27] *Portable Computing Language | NVIDIA GPU support via CUDA backend*. http://portablecl.org/cuda-backend.html. Accessed: 2019-02-05 (cit. on p. 53).

[28] *Renesas R-Car V3M Started Kit: Hardware Manual*. https://www.renesas.com/eu/en/solutions/automotive/soc/r-car-v3m.html. Accessed: 2019-02-05 (cit. on p. 44).

[29] *Renesas R-Car V3M: Overview*. https://www.renesas.com/eu/en/solutions/automotive/soc/r-car-v3m.html. Accessed: 2019-02-05 (cit. on pp. 44, 45).

[30] *Robot Operating System*. http://www.ros.org. Accessed: 2019-02-05 (cit. on p. 33).

[31] M. Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning, 2012 (cit. on p. 25).

[32] *SDSoC Development Environment*. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html. Accessed: 2018-12-20 (cit. on p. 47).

[33] *SYCL Specification*. https://www.khronos.org/sycl. Accessed: 2018-12-20 (cit. on p. 76).

[34] *sycl-gtx*. https://github.com/ProGTX/sycl-gtx. Accessed: 2019-02-05 (cit. on p. 75).

[35] *The OpenCL C++ Wrapper API*. https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf. Accessed: 2019-02-05 (cit. on p. 25).

[36] *triSYCL*. https://github.com/triSYCL/triSYCL. Accessed: 2019-02-05 (cit. on p. 75).

[37] *Xilinx Press Release*. https://www.xilinx.com/news/press/2018/daimler-ag-selects-xilinx-to-drive-artificial-intelligence-based-automotive-applications.html. Accessed: 2019-02-05 (cit. on pp. 1, 8).

[38] *Xilinx SDSoC Example Repository*. https://github.com/Xilinx/SDSoC_Examples. Accessed: 2019-02-05 (cit. on p. 48).

[39] *Xilinx SDSoC Programmers Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1278-sdsoc-programmers-guide.pdf. Accessed: 2019-02-05 (cit. on p. 47).

[40] *Xilinx ZCU102 Evaluation Board: User Guide*. https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf. Accessed: 2019-02-05 (cit. on pp. 45, 46).

[41] *Xilinx Zynq UltraScale+ MPSoC Data Sheet: Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. Accessed: 2019-02-05 (cit. on p. 45).

[42] *Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html. Accessed: 2018-12-20 (cit. on p. 45).

Appendices

Performance Data

A

	Platform	points2-image (2500)	Euclidean Clustering[2] (350)	(90)	NDT Mapping[3] (115)	(32)
Serial	Nvidia Jetson TX2 CPU	26.91	83[1]	6.02	55.7	9.23
	Nvidia Jetson Xavier CPU	24	61.8	4.92	27.7	4.55
	Renesas V3M CPU	27	355	29.9	161	22.4
	Xilinx Zynq ZCU102 CPU	34.65	–	–	–	–
CUDA	Nvidia Jetson TX2 GPU	7.6	51.34	11.57	38.53	7.39
	Nvidia Jetson Xavier GPU	4.7	21.3	4.63	17.35	5.15
OpenCL	Nvidia Jetson TX2 GPU	101.14	50.85	11.24	127.17	14.67
	Renesas V3M IMP	70.77	–	63.93	–	56.13
	Xilinx Zynq ZCU102 FPGA	549.25	–	–	–	–
OpenMP	Nvidia Jetson TX2 CPU	16.55	47.45	4.31	28.57	4.43
	Nvidia Jetson Xavier CPU	24.8	38.4	2.74	17.19	2.85

Absolute performance data for all implementations on all platforms. Unless noted otherwise, all numbers are the total runtime in seconds for the execution of all available test-samples (number of samples given in parentheses).

[1] Runtime after backporting of optimizations from parallel implementation to serial implementation (cf. Section 4.2.2). Original runtime: 505s.

[2] Due to the limited size of the global memory on the Renesas V3M IMP, only the first 90 testcases could be executed. For better comparison we also give runtimes for 90 testcases only (cf. Section 5.3).

[3] Due to the limited size of the global memory on the Renesas V3M IMP, only the first 32 testcases could be executed. For better comparison we also give runtimes for 32 testcases only (cf. Section 5.3).

Bisher in der FAT-Schriftenreihe erschienen (ab 2014)

Nr.	Titel
263	Laserstrahlschweißen von Stahl an Aluminium mittels spektroskopischer Kontrolle der Einschweißtiefe und erhöhter Anbindungsbreite durch zweidimensional ausgeprägte Schweißnähte, 2014
264	Entwicklung von Methoden zur zuverlässigen Metamodellierung von CAE Simulations-Modellen, 2014
265	Auswirkungen alternativer Antriebskonzepte auf die Fahrdynamik von PKW, 2014
266	Entwicklung einer numerischen Methode zur Berücksichtigung stochastischer Effekte für die Crash-simulation von Punktschweißverbindungen, 2014
267	Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 1, 2014
268	Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 2, 2014
269	Schwingfestigkeitsbewertung von Schweißnahtenden MSG-geschweißter Feinblechstrukturen aus Aluminium, 2014
270	Physiologische Effekte bei PWM-gesteuerter LED-Beleuchtung im Automobil, 2015
271	Auskunft über verfügbare Parkplätze in Städten, 2015
272	Zusammenhang zwischen lokalem und globalem Behaglichkeitsempfinden: Untersuchung des Kombinationseffektes von Sitzheizung und Strahlungswärmeübertragung zur energieeffizienten Fahrzeugklimatisierung, 2015
273	UmCra - Werkstoffmodelle und Kennwertermittlung für die industrielle Anwendung der Umform- und Crash-Simulation unter Berücksichtigung der mechanischen und thermischen Vorgeschichte bei hochfesten Stählen, 2015
274	Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components, 2015
275	Hochrechnung von GIDAS auf das Unfallgeschehen in Deutschland, 2015
276	Literaturanalyse und Methodenauswahl zur Gestaltung von Systemen zum hochautomatisierten Fahren, 2015
277	Modellierung der Einflüsse von Porenmorphologie auf das Versagensverhalten von Al-Druckgussteilen mit stochastischem Aspekt für durchgängige Simulation von Gießen bis Crash, 2015
278	Wahrnehmung und Bewertung von Fahrzeugaußengeräuschen durch Fußgänger in verschiedenen Verkehrssituationen und unterschiedlichen Betriebszuständen, 2015
279	Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen – Teil 3, 2015
280	PCM from iGLAD database, 2015
281	Schwere Nutzfahrzeugkonfigurationen unter Einfluss realitätsnaher Anströmbedingungen, 2015
282	Studie zur Wirkung niederfrequenter magnetischer Felder in der Umwelt auf medizinische Implantate, 2015
283	Verformungs- und Versagensverhalten von Stählen für den Automobilbau unter crashartiger mehrachsiger Belastung, 2016
284	Entwicklung einer Methode zur Crashsimulation von langfaserverstärkten Thermoplast (LFT) Bauteilen auf Basis der Faserorientierung aus der Formfüllsimulation, 2016
285	Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf realer Fahrbahn, 2016

286 χMCF - A Standard for Describing Connections and Joints in the Automotive Industry, 2016

287 Future Programming Paradigms in the Automotive Industry, 2016

288 Laserstrahlschweißen von anwendungsnahen Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2016

289 Untersuchung der Bewältigungsleistung des Fahrers von kurzfristig auftretenden Wiederübernahme-situationen nach teilautomatischem, freihändigem Fahren, 2016

290 Auslegung von geklebten Stahlblechstrukturen im Automobilbau für schwingende Last bei wechselnden Temperaturen unter Berücksichtigung des Versagensverhaltens, 2016

291 Analyse, Messung und Optimierung des Ventilationswiderstands von Pkw-Rädern, 2016

292 Innenhochdruckumformen laserstrahlgelöteter Tailored Hybrid Tubes aus Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2017

293 Filterung an Stelle von Schirmung für Hochvolt-Komponenten in Elektrofahrzeugen, 2017

294 Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Feinbleche aus Stahl unter kombinierter Beanspruchung, 2017

295 Wechselwirkungen zwischen zyklisch-mechanischen Beanspruchungen und Korrosion: Bewertung der Schädigungsäquivalenz von Kollektiv- und Signalformen unter mechanisch-korrosiven Beanspruchungs-bedingungen, 2017

296 Auswirkungen des teil- und hochautomatisierten Fahrens auf die Kapazität der Fernstraßeninfrastruktur, 2017

297 Analyse zum Stand und Aufzeigen von Handlungsfeldern beim vernetzten und automatisierten Fahren von Nutzfahrzeugen, 2017

298 Bestimmung des Luftwiderstandsbeiwertes von realen Nutzfahrzeugen im Fahrversuch und Vergleich verschiedener Verfahren zur numerischen Simulation, 2017

299 Unfallvermeidung durch Reibwertprognosen, 2017

300 Thermisches Rollwiderstandsmodell für Nutzfahrzeugreifen zur Prognose fahrprofilspezifischer Energieverbräuche, 2017

301 The Contribution of Brake Wear Emissions to Particulate Matter in Ambient Air, 2017

302 Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC), 2017

303 Experimentelle Untersuchung des Einflusses der Oberflächenbeschaffenheit von Scheiben auf die Kondensatbildung, 2017

304 Der Rollwiderstand von Nutzfahrzeugreifen unter realen Umgebungsbedingungen, 2018

305 Simulationsgestützte Methodik zum Entwurf intelligenter Energiesteuerung in zukünftigen Kfz-Bordnetzen, 2018

306 Einfluss der Kantenbearbeitung auf die Festigkeitseigenschaften von Stahl-Feinblechen unter quasistatisc und schwingender Beanspruchung, 2018

307 Fahrerspezifische Aspekte beim hochautomatisierten Fahren, 2018

308 Der Rollwiderstand von Nutzfahrzeugreifen unter zeitvarianten Betriebsbedingungen, 2018

309 Bewertung der Ermüdungsfestigkeit von Schraubverbindungen mit gefurchtem Gewinde, 2018

310 Konzept zur Auslegungsmethodik zur Verhinderung des selbsttätigen Losdrehens bei Bauteilsystemen im Leichtbau, 2018

311 Experimentelle und numerische Identifikation der Schraubenkopfverschiebung als Eingangsgröße für eine Bewertung des selbsttätigen Losdrehens von Schraubenverbindungen, 2018

312 Analyse der Randbedingungen und Voraussetzungen für einen automatisierten Betrieb von Nutzfahrzeuge im innerbetrieblichen Verkehr, 2018

313 Charakterisierung und Modellierung des anisotropen Versagensverhaltens von Aluminiumwerkstoffen für die Crashsimulation, 2018

314 Definition einer „Äquivalenten Kontakttemperatur“ als Bezugsgröße zur Bewertung der ergonomischen Qualität von kontaktbasierten Klimatisierungssystemen in Fahrzeugen, 2018

315 Anforderungen und Chancen für Wirtschaftsverkehre in der Stadt mit automatisiert fahrenden E-Fahrzeugen (Fokus Deutschland), 2018

316 MULTIC-Tooling, 2019

317 EPHoS: Evaluation of Programming - Models for Heterogeneous Systems, 2019

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863

Verband der Automobilindustrie e.V. (VDA)
Behrenstraße 35, 10117 Berlin
www.vda.de
Twitter @VDA_online

Forschungsvereinigung Automobiltechnik e.V. (FAT)
Behrenstraße 35, 10117 Berlin
www.vda.de/fat

VDA | Verband der Automobilindustrie

FAT | Forschungsvereinigung Automobiltechnik