OpenMP Device Offloading for Embedded Heterogeneous Platforms - Work-in-Progress

Lukas Sommer, Andreas Koch
Embedded Systems and Applications Group, TU Darmstadt, Germany
{sommer, koch}@esa.tu-darmstadt.de

Abstract—**The growing computational demands of automotive applications require the use of powerful embedded, heterogeneous computing platforms in vehicles. OpenMP, and in particular its device offloading features, are a promising candidate programming model for these platforms.**

In this work, we show how typical automotive workloads can be implemented and optimized with OpenMP device offloading. To this end, we also adapt the LLVM OpenMP runtime to embedded, heterogeneous platforms. Our evaluation shows that OpenMP device offloading can deliver performance similar to that of optimized CUDA implementations.

I. Introduction

As modern driver-assistance and autonomous driving functionalities demand large amounts of computational power onboard of vehicles, the automotive industry is starting to adopt embedded, heterogeneous platforms, such as the Nvidia Drive system, to supply the required computational power.

Programming these systems can be a challenging task and requires a programming model suitable for the platform. In their study, Sommer et al. [1] identified OpenMP as a very interesting candidate, in particular due to its ease-of-use and maintainability. With the device offloading features introduced in version 4.0, and further refined in newer versions of the OpenMP standard, OpenMP now also allows to target *heterogeneous* systems, e.g., combining a multi-core CPU and a GPU. However, in their study [1], Sommer et al. also found the compiler support for OpenMP device offloading to still be limited on embedded systems. Since the study has been conducted, the OpenMP support in compilers has evolved, e.g., with the LLVM compiler infrastructure now supporting OpenMP device offloading on ARM-based systems.

In this work, we show how OpenMP offloading can be used for automotive workloads on embedded heterogeneous platforms by accelerating three automotive workloads from the open-source DAPHNE benchmark suite [2] on an embedded platform combining a multi-core CPU and a GPU. We also modify the LLVM OpenMP runtime to facilitate management of shared memory on embedded platforms.

II. Implementation

We use the three automotive benchmark kernels points2image (P2I), euclidean_clustering (EC) and ndt_mapping (NDT) from the open-source DAPHNE suite [2] to demonstrate how OpenMP device offloading can be used to accelerate performance-critical parts of automotive applications. All three kernels represent typical automotive workloads and have been extracted from the open-source Autoware framework for autonomous driving [3].

As OpenMP constructs for device offloading differ from the parallel constructs for CPUs, a serial implementation in pure `C++` is used as starting point for the study. For better comparability with the hand-written CUDA implementation provided in the DAPHNE suite, the same performance-critical sections of the program are offloaded to the GPU.

After adding the OpenMP offloading constructs to the code (e.g., `omp target teams distribute`), it would already be possible to run the kernels and offload the OpenMP target regions to a GPU. However, this implementation does not yet exploit one of the most important differences between *embedded* heterogeneous platforms and heterogeneous systems typically found in HPC domains: on embedded platforms, the different components of the system often *physically* share the same memory, as it is the case for our target system, the Nvidia Jetson platform [4].

We therefore modify the LLVM OpenMP runtime to allow for allocation of memory that can be accessed by both, the host CPU and the GPU. Using the standard OpenMP function `omp_target_alloc`, it is now possible to allocate memory accessible by *both* components, and avoid expensive data-copies. These data-copies made up for 88% of the execution time for points2image and 49% for euclidean_clustering, so allocating memory usable by both CPU and GPU significantly reduces the kernel execution time. For the benchmark ndt_mapping, execution was not even possible with application data allocated twice (once in host memory, once in GPU memory) because this wasteful allocation exceeded the system memory.

It is also possible to further optimize the ndt_mapping application performance: Profiling the kernel shows that one of the target regions is 10x slower than its CUDA counterpart. Further investigation of the region shows that the atomic update (`omp atomic update`) is compiled to a somewhat inefficient PTX code sequence, whereas the hand-written CUDA version uses the CUDA builtin function `atomicAdd`.

But using OpenMP's `declare variant` mechanism, a specialized function for the atomic update can be defined. This specialized function for the CUDA architecture will automatically be selected by the compiler when offloading to the CUDA architecture, whereas a generic version of the function will be used for other architectures, keeping the OpenMP-based implementation portable.

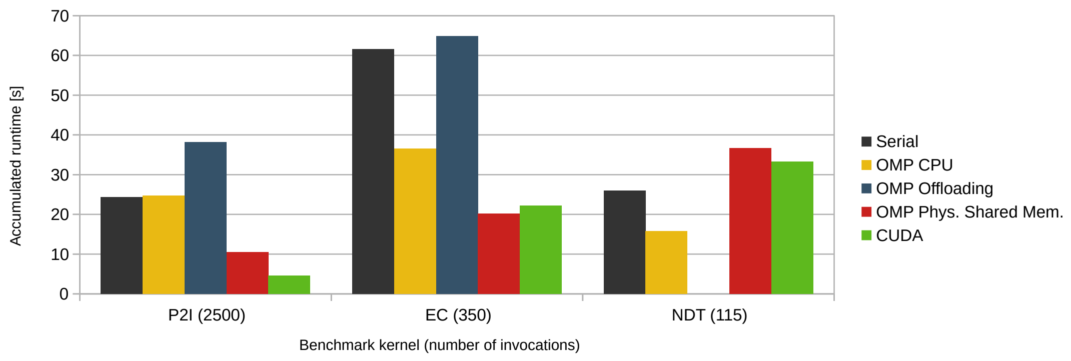

Fig. 1: Accumulated runtime of benchmark kernel execution in seconds.

III. Evaluation

We use an Nvidia Jetson AGX Xavier platform to compare a total of five different implementations for the three different kernels:

- Serial baseline implementation in pure C++ (Serial).
- CPU-only OpenMP-implementation (OMP CPU).
- OpenMP offloading implementation assuming separate memory (OMP Offloading).
- OpenMP offloading implementation optimized for physically shared memory (OMP Phys. Shared Mem.).
- Hand-written CUDA implementation (CUDA).

Fig. 1 shows the accumulated runtime for each kernel (averaged over three runs), the number of invocations corresponds to the *full* data-set of the DAPHNE benchmark and is given in parentheses. The bar for *OMP Offloading* for ndt_mapping is missing due to the reasons explained in the previous section.

For the first two applications, the use of physically shared memory (OMP Phys. Shared Mem.), as enabled by our modified version of LLVM's `libomptarget`, dramatically improves the execution time compared to the version assuming separate memory. In case of the points2image kernel, there is still a significant gap between the OpenMP offloading version with physically shared memory and the hand-written CUDA implementation, but the OpenMP offloading nevertheless clearly outperforms the CPU-only OpenMP implementation. The OpenMP offloading implementation with physically shared memory of euclidean_clustering even outperforms the optimized CUDA implementation by a small margin, making the OpenMP offloading the fastest implementation of this benchmark kernel. For ndt_mapping, there is a small difference in performance between OpenMP offloading with physically shared memory and CUDA implementation, and both versions are not able to keep up with the CPU-only implementation of this kernel.

Table I further investigates the difference between OpenMP offloading with physically shared memory and hand-written CUDA implementations by looking at the execution time per invocation of the different GPU regions as given by `nvprof`. As the region execution time is almost equal for points2image, the performance difference is most likely caused

Benchmark	Region	Calls	OMP PSM [µs]	CUDA [µs]
P2I	Region 1	2,500	118	121
EC	Region 1	1,726	541	1,728
	Region 2	191,132	8.8	7.4
NDT	Region 1	115	12,022	9,784
	Region 2	115	35,008	33,829

TABLE I: Average runtime per kernel invocation in µs.

by the OpenMP runtime itself. For euclidean_clustering, the execution time for the second region is almost identical, the small advantage of OpenMP offloading over CUDA stems from the difference in execution time of the first region. With the OpenMP offloading version for ndt_mapping, both regions are slightly slower than their CUDA pendants. However, an investigation of the original implementation shows that the implementation of the specialized atomic update using OpenMP `declare variant` reduces the execution time of Region 1 by a factor of more than 7x (from 87,890µs to 12,022µs).

IV. Conclusion & Outlook

In this work, we have demonstrated how OpenMP device offloading can be used to accelerate automotive workloads on embedded heterogeneous platforms, and how application performance can further be improved by adapting the OpenMP runtime to the special features of embedded systems and by using advanced OpenMP mechanisms, such as platform-specialized function variants.

The optimized OpenMP offloading implementations developed in this work are available in the public DAPHNE source code repository on Github[1]. The modified version of the LLVM infrastructure is also publicly available on Github[2].

In the future, we plan to further improve the efficiency of the OpenMP runtime on embedded platforms and investigate the use of OpenMP offloading for other embedded and automotive use-cases.

[1]https://github.com/esa-tu-darmstadt/daphne-benchmark
[2]https://github.com/sommerlukas/llvm-offload-jetson

References

[1] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, “Using parallel programming models for automotive workloads on heterogeneous systems - a case study,” in *28th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP’20)*, 2020.

[2] ——, “Daphne - automotive benchmark suite for parallel programming models on embedded heterogeneous platforms - work-in-progress,” in *Proceedings of the International Conference on Embedded Software*, ser. EMSOFT ’19. Piscataway, NJ, USA: IEEE Press, 2019.

[3] S. Kato, S. Tokunaga, Y. Maruyama *et al.*, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS ’18, 2018.

[4] Nvidia Inc., “Cuda for tegra - application note,” https://docs.nvidia.com/cuda/pdf/CUDA-for-Tegra-AppNote.pdf, 2019, accessed: 2020-06-03.