An alternative OpenMP Backend for Polly

EuroLLVM 2019 Student Research Competition

Michael Halkenhäuser
TU Darmstadt
michael.halkenhaeuser@stud.tu-darmstadt.de

Lukas Sommer
Embedded Systems and Applications Group
TU Darmstadt
sommer@esa.tu-darmstadt.de

1 Introduction

LLVM's polyhedral infrastructure framework *Polly* [1] may automatically exploit thread-level parallelism through OpenMP. Currently, the user can only influence the number of utilized threads, while other OpenMP parameters such as the *scheduling type* and *chunk size* are set to fixed values. This in turn, limits a user's ability to adapt the optimization process for a given problem. Additionally, a project is required to support the GNU OpenMP runtime in order to benefit from the automatic thread-level parallelization.

In this paper we will present an extension of Polly's current OpenMP backend, which not only provides the aforementioned two additional customization options but is also based on LLVM's OpenMP runtime.

Our contribution comprises an OpenMP backend extension for Polly and an evaluation of how the different user options influence program performance. Through the additional user options implemented in this work, significant performance gains are possible in a number of cases, while being overall competitive to the current backend. Furthermore, by supporting the use of an additional runtime library this work may allow for a broader employment of polyhedral techniques.

2 Approach

We will now take a closer look at the introduced customization options and their theoretical effect. Generally, the OpenMP API[1] offers three *scheduling types*: *static*, *dynamic* and *guided*. All kinds of scheduling support an additional parameter called *chunk size*, which defines the size of a thread's share of work or its processing order.

static scheduling will (try to) distribute the work evenly among the available worker threads. Therefore, the chunks are determined before the actual computation begins and then provided to every worker. This approach has the advantage that threads know their iterations upfront and do not have to request any further information, which would imply additional runtime calls. Hence, static scheduling is well-suited for computation tasks where each chunk costs about the same amount of time, such that the worker threads complete (roughly) simultaneously.

dynamic scheduling assigns chunks on active request of a worker thread. After completion of a chunk, another request is issued by the respective worker until the complete amount of work is done. For example a chunk size of one allows for direct interleaving of worker threads, which minimizes the overhead of waiting for other workers to finish but also leads to many runtime calls, requesting the next chunk of work.

guided scheduling tries to strike a balance between *static* and *dynamic* by assigning rather big chunks of work at the beginning. Upon completion, the next chunk has to be requested exactly as in the case of dynamic scheduling. However, the chunk size will be reduced successively, until it reaches the provided minimum, at which point the behavior of this scheduling type is not distinguishable from the dynamic type. As such the number of requests for next chunks is reduced, while still being able to compensate load imbalances of worker threads.

At present, the OpenMP backend of *Polly* supports only dynamic scheduling paired with a chunk size of one.

Since the implementation of our backend is derived from [1], the OpenMP parallelization is performed in the same spot of Polly's workflow, by using the program's polyhedral AST and reconverting it into LLVM-IR, accordingly. This is accomplished by restructuring the basic blocks of the original program and adding OpenMP runtime calls to a newly created outlined function. The implementation of our backend follows the structure of the existing OpenMP backend, therefore we achieve the exact same coverage as the current backend and create similar structures, wherever applicable.

3 Evaluation Setup

In order to investigate the performance of our backend and compare it to the existing one, we chose *polybench*[2] as experimental platform. This benchmark suite offers 30 sample programs, which are suitable for polyhedral transformations, together with parameterizable datasets. Since only 18 benchmarks contained automatically parallelizable loop structures, we focused on those programs.

The runtime of each benchmark was measured multiple times with the *small* and *large* dataset sizes and the final results were calculated using the 10% truncated mean of 120 (*small* dataset) and 36 (*large* dataset) runs.

[1] See e.g.: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[2] See: https://sourceforge.net/projects/polybench/

All experiments were conducted using a Linux platform (kernel version 4.16) equipped with an AMD Ryzen 5 1600X and 32 GiB of RAM.

4 Results

In this section we will first evaluate the impact of different *chunk sizes* and *scheduling strategies* as well as the *number of threads* within our backend. Afterwards, the performance will be compared to the current OpenMP backend and Clang 8.

4.1 Chunk sizes (12 Threads, Dynamic Sched.)

Our results indicate that in case of *large* datasets, a variation of the chunk size with the following values {1, 2, 3, 4, 6} might reduce performance in seven out of 18 benchmarks while nine results stay roughly the same when compared to the baseline with a chunk size of one. However, in two cases (*ludcmp*, *trmm*), an increased chunk size leads to considerable speed-ups between 1.25× and 3.25× – it is important to note that in case of the *trmm* benchmark the peak performance was not reached with the highest chunk size.

When switching over to *small* datasets, we observe an overall significant drop in performance as the amount of total work and therefore the resulting number of distributable chunks is decreased. As a result, only a fraction of available threads is actually provided with work.

Overall, our evaluation shows that the chunk size is an important parameter to fine-tune the performance of automatically parallelized applications.

4.2 Scheduling types (12 Threads, Chunk Size 1)

In this setup the similarities between *guided* and *dynamic* scheduling strategies may be observed as the first one may only achieve significantly different speed-ups (up to 4.5×) in two benchmarks (*ludcmp*, *trmm*). Static scheduling on the other hand may even exceed these speed-ups (reaching up to 8.5×) in the small and large dataset settings, but will also decrease performance in eight cases.

4.3 Number of Threads (Chunk Size 1)

Lastly, we will evaluate different thread numbers: {4, 8, 12}, always using four threads as our baseline. As described in the previous section, *guided* and *dynamic* scheduling are very similar and therefore *guided* will not be discussed since the results did not provide any additional information.

Overall dynamic scheduling will benefit from more available threads, but in case of small datasets the performance of three benchmarks slightly decreases. Similar results may be observed when static scheduling is taken into account: only small problem sizes suffer from a decrease in performance (five cases). Especially *trmm* will double its execution time.

Our assumption is that the setup of threads, library calls and collection of results creates a considerable overhead, such that a higher amount of threads may hamper performance significantly.

4.4 Backend comparison

We will now accumulate and use the results of the previous section: a chunk size of one is *usually a good choice* and the maximum number of threads performs best in case of large datasets. Therefore, we will compare the different scheduling kinds of our backend with said settings against the current OpenMP backend and use Polly (without vectorization) as a reference. Since in case of small datasets Polly without OpenMP parallelization significantly outperformed both backends in nearly every benchmark, we will only concentrate on the results of the large datasets.

In general we can observe that the results are comparable and there is only one benchmark (*syrk*) where our alternative backend is at least 5% slower with any scheduling strategy (0.7-0.9×). However, our backend may surpass the performance of the current backend (and Polly) in six out of 18 cases ranging from at least 1.1× speed-up (compared to the GNU backend) to 2.7× for *ludcmp* and even reach 8.9× for *trmm*.

4.5 Overall evaluation

Finally, we want to perform a comparison between both backends and vanilla clang-8 `-O3` (as baseline) on the large dataset: using dynamic scheduling, chunk size one and enabled vectorization (both backends).

The gathered results show that automatic parallelization may obtain significant speed-ups, starting at 2.5× and reaching around 25-30× for specific benchmarks. Generally, the achieved results are comparable, but in five cases our backend is able to gain significantly higher speed-ups than the GNU backend (*bicg*: 3.7 vs. 2.9, *gesummv*: 6.7 vs. 3.9 *ludcmp*: 1.2 vs. 0.4, *mvt*: 4.2 vs. 3.4, *trmm*: 10.0 vs. 1.1).

Nevertheless, we have to point out that automatic parallelization is not able to achieve positive results in every benchmark. In two cases only around 1.0× speed-up is possible and performance may even decrease (*atax*) in this setting.

5 Conclusion

These results suggest that our implementation was overall successful. Our additional customization options allow for significant performance gains in particular cases and carry no clear drawback. As such we think that this work would be a considerable addition to Polly and plan to contact the developers in the foreseeable future. Currently, the alternative backend is hosted at: https://github.com/mhalk/polly.

References

[1] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.