OpenMP Target Offload Utilizing GPU Shared Memory

Mathias Gammelmark(✉), Anton Rydahl, and Sven Karlsson

Technical University of Denmark, Anker Engelunds Vej 1, 2800 Kgs, Lyngby, Denmark
{magam,svea}@dtu.dk, rydahlanton@gmail.com

Abstract. Memory resources are an important aspect to consider when designing high performing programs. This is especially true for programs running on graphical processing units, GPUs, yet this is not something trivially done using current OpenMP target offloading. In this paper, we examine methods for implementing parallel programs running on GPUs, which rely on locally shared memory resources and intricate synchronization. Employing the methods, we show you can achieve between 1.5 to 9 relative speedup over a range of compilers. We evaluate portability by running experiments on two systems, utilizing different GPU technologies and vendors. We further investigate scheduling, synchronization and execution time of our experiments, to better understand the overhead associated with using OpenMP, compared to architecture specific languages. Lastly, we argue that improved GPU scheduling could yield a potential speedup of 3.

Keywords: GPGPU Programming · OpenMP Target Offloading · Shared Memory · Fine-Grained Parallelism

1 Introduction

As supercomputers advance towards exascale levels of performance, there is an increasing trend of incorporating accelerators such as Graphics Processing Units, *GPU*s, to enhance computational density and efficiency. However, the integration of GPUs into supercomputing systems introduces additional complexity for developers, necessitating the evolution of parallel programming models to keep pace with this trend. While architecture-specific languages and tools such as the Compute Unified Device Architecture, *CUDA* [17], provides lower-level control of the GPU hardware often allowing higher performing programs through hardware intrinsic functions, they also place a greater burden on developers, requiring them to invest more effort and time in development.

Open Multi-Processing, *OpenMP* [19], is an application programming interface, *API*, which has long been a popular choice for shared memory parallelism within the realm of High Performance Computing, *HPC*. OpenMP provides developers with a range of library routines and compiler directives that help

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
S. McIntosh-Smith et al. (Eds.): IWOMP 2023, LNCS 14114, pp. 114–128, 2023.
https://doi.org/10.1007/978-3-031-40744-4_8

manage the low-level details of parallelization, allowing developers to focus on high-level logic. An important addition in OpenMP 4.0 [20] is the introduction of target offload, which allow developers to offload part of programs to GPUs using a similar syntax to traditional OpenMP parallelization, bringing the high portability of OpenMP to GPU programming.

Target offload in OpenMP is still in an early stage of development across many compilers, including different implementations of the standard. This can often lead to a significant difference in behavior and performance, for the same lines of code when using different compilers. Additionally, GPUs provide fast local memory resources, which is shared between groups of threads providing very fast access. This locally shared memory is important to consider for high performing GPU applications, as it enables fast data sharing and pre-fetching. Yet this area of controlling locally shared memory is not fully mature and can result in a high degree of variation in performance across compilers.

This paper aims to analyze the practical aspects of implementing portable programs with effective utilization of GPU shared memory with OpenMP, specifically focusing on block/chunk algorithms by conducting experiments on two different system, employing both NVIDIA and AMD hardware.

Furthermore, we compare the performance of compiler generated code and performance portability across different compilers. Lastly, we will profile the resulting GPU kernels, to further investigate the underlying scheduling and synchronization of the target regions.

The main contributions of this paper are as follows. We:

- examine the effectiveness of manual parallelization and automatic loop-based work-sharing constructs in OpenMP target offloading for fine-grained control of parallelization and evaluate their suitability for algorithms reliant on GPU shared memory and blocking.
- compare performance and usability of target offloading for blocking algorithms depending on local synchronization and memory resource sharing for the most common compilers on modern hardware.
- identify areas where OpenMP target offloading lags behind, compared to a equivalent program written in CUDA.

In Sect. 2, this paper provides an overview of OpenMP target offloading and introduces the two algorithms used for experimentation. Section 3 presents our experimental results, both considering methods of parallelization, compiler capabilities and in-depth profiling of GPU scheduling. Section 4 gives a brief summary of related works and Sect. 5 concludes the paper by summarizing key findings.

2 Background

GPUs are highly parallel processors, which excel especially at handling large workloads and processing where similar work is required for a large number of elements. They can typically run thousands of threads at the same time,

spread across hundreds of cores, making it crucial to consider ways to maximize parallelization in one's programs.

GPUs are typically structured in many physical groups of tightly packed cores, utilizing a Single Instruction Multiple Data, *SIMD*, architecture, for maximizing the density of cores, allowing significantly higher core count compared to conventional CPUs. NVIDIA refer to these groups as Streaming Multiprocessors, *SMs*, containing 32 cores, while AMD designates them as Compute Units, *CUs*, containing 64 cores. Each of these SM/CU will process the same instruction across all cores and through smart hardware and software mapping enable execution of multiple threads at the same time, all executing the same instructions in lockstep. This results in great performance when the flow of execution is identical across all threads, but when a branch is encountered the efficiency will decrease as some cores will remain idle, while the neighbors are executing other branches.

The cores are not only densely clustered in groups, but is also tightly interconnected in the groups, sharing the same registers, L1 cache, and locally shared memory, enabling significantly faster data sharing and synchronization within a group compared to outside [16]. It is therefore important to structure programs in such a way, that most communication is happening within one group, to best utilize the local interconnection. This is already deeply embedded in the architecture-specific languages, such as CUDA and OpenCL, where GPU work is organized in *blocks* or *workgroups* respectively, representing a block/group of threads which work together within the same SM/CU. This organization enables efficient sharing and synchronization between the threads, as they all remain within the same group. Beyond the blocks/workgroup, synchronization becomes limited, although some data sharing can still be accomplished through the use of atomic operations and global memory access. If global synchronization is required, it becomes necessary to wait for all blocks/groups to complete before continuing.

There are several approaches to writing GPU programs using OpenMP target offloading. Section 2.1 discusses the available parallel constructs in OpenMP and Sect. 2.2 discusses the example applications being used for the experiments.

2.1 OpenMP Target Parallelism for GPUs

The analogous concept to *blocks* and *workgroups* in OpenMP is the `teams` construct, which similarly represents a group of threads working tightly together. We consider two main methods for organizing work into blocks. The first is manually defining the work-sharing and the number of teams in the `target teams` region, using the `num_teams` directive, with the intention of starting a team for each block. The number of teams created is a higher bound for OpenMP 5.0 and will be defined by the implementation, but both a lower and higher bound have later been introduced, making this methods more portable for newer versions of OpenMP. The second method is using the `distribute` work-sharing construct to parallelize a for-loop across all available teams, and then let the implementation decide how many actual teams are used, see Listing 1.1. This has the

advantage of being independent of the actual number of teams created, making the methods much more portable. The negative side to this method is that it can introduce an additional overhead of managing an outer loop iterating over the blocks, depending on how the implementation decides to handle the distribution. This effect will be examined further in Sect. 3.3.

To fully utilize the hardware a second level of parallelism must be created using a parallel region. Similarly, to the `teams` construct, the parallel region can either be created by directly by specifying the number of threads, or by distributing a loop using the `parallel for` construct. The number of threads in a team can be specified using the `thread_limit` clause, but this also defines an upper limit in OpenMP 5.0, meaning that the actual number of threads in the team is implementation specific, but similarly this clause also received a lower bound in a later version of OpenMP, see Listing 1.2. Using the `parallel for` work-sharing construct is again the more portable solution but might add additional overhead from handling the loop logic and automatic work-sharing.

The main reason we have for separating the team and thread parallelization in OpenMP is to enable allocation of shared memory, which is done by handling array and variable initialization before starting the thread parallelization. A second reason is synchronization, which can be managed using either `barrier` constructs within parallel regions or by ending a loop parallelization for the work-sharing constructs. It is important to note that forking and joining of the loop parallelization can incur a significant overhead, depending on how the work-sharing is implemented.

In Listings 1.1 and 1.2 are code snippets illustrating the overall structure of both the manual work-sharing and the automatic `distribute` and `parallel for` based work-sharing. The outer most pragma is responsible for allocating all blocks and assigning them to the teams and initializing the shared memory, where the inner most pragma is responsible for distributing the blocks work across all available threads within the team.

```
1  #pragma omp target teams distribute thread_limit (BLOCK_SIZE ) nowait depend (...)
2  for (int block_id = 0; block_id < n_blocks ; ++block_id ) {
3      int temp [BLOCK_SIZE ];
4  #pragma omp parallel for num_threads (BLOCK_SZIE ) shared (tmp )
5      for (int thid = 0; thid < BLOCK_SIZE ; ++thid ) {
6          // Do Someting ...
```

Listing 1.1. Worksharing constructs with distribute and parallel for

```
1  #pragma omp target teams num_teams (num_blocks ) thread_limit (BLOCK_SIZE ) nowait depend (...)
2  {
3      int temp [BLOCK_SIZE ];
4  #pragma omp parallel num_threads (BLOCK_SZIE ) shared (tmp )
5      {
6          // Do Someting ...
```

Listing 1.2. Manual worksharing using `num_teams` and `thread_limit`

In OpenMP 5.0 the `loop` construct was introduced with higher restrictions, allowing potentially better static analysis and mapping to hardware [21]. The usage is very similar to the `distribute` and `parallel for` constructs, but allows mapping to either teams or parallel regions, based on the `bind` clause. However, the `loop` construct is still not widely supported across compilers and will only be examined using the NVC compiler.

2.2 Applications

The different parallel patterns are considered for two algorithms utilizing blocking and GPU shared memory. The first algorithm is the parallel scan based on the Blelloch algorithm [2], which computes the sum of all previous elements in an array. Scan, or also called prefix sums is widely used, eg. for binning and stream compaction in the AMReX framework [28], enabling large scale dynamic particle simulations on GPUs or for list compaction in tree construction algorithms as presented by Wu et al. [26]. Multiple variations of this algorithm exists, with current state of the art based on decoupled look-back [15]. However, for the context of this paper a more simplistic approach is taken, following the presented implementation by Harris et al. in *GPU gems 3* [11]. The overall idea of the algorithm is to calculate the prefix sum locally within a block using the Blelloch algorithm and writes out the total sums of the blocks to an auxiliary array, which then recursively have the prefix sum calculated, until the auxiliary array can fit into a single block. The prefix sum of the blocksums, then corresponds to the sum off all previous blocks, which then can be used to find the global prefix sum.

The second application considered is the parallel four-way LSB radix sorting algorithm, which additionally utilizes the parallel scan for iteratively sorting elements 2 bits at a time, from the lowest bit to the highest bits. A simplified version of the four-way radix sort as described by Linh Ha et al. [10] is used for the experiments, which is omitting the parallel order checking and is using a direct mapping instead of the coalesced block mapping presented. This is done to keep the complexity low, as the focus is held on how to best control the parallelism. Other versions of the parallel radix sort exist, such as 16-way radix sort, sorting by 4 bits at a time, or current state algorithms such as one-sweep radix sorting [1] using the previously mentioned single pass prefix sum, with decoupled look-back.

Both applications are heavily using blocking iteration, where data is processed in blocks and are heavily relying on shared memory. For this reason, we are forced to use separated teams and thread parallel regions, as described in Sect. 2.1.

3 Results

The applications are evaluated using multiple compilers, on two different HPC systems, using GPUs from both AMD and NVIDIA. Before reviewing the experimental results, we'll provide a short description of the two systems in Sect. 3.1 and the compilers in Sect. 3.2.

Table 1. Architecture information for the nodes used in the two systems. The information in 1a has been gathered from the A100 white-paper [16] for the GPU and CPU info have been reported with `lscpu` within the batch jobs. The information in 1b has been gathered from the LUMI documentation [4].

CPU Information

Model name	Intel Xeon Gold G226G
GPU count	2
Sockets	2
Cores per socket	16
L1d cache	32KiB
L1i cache	32KiB
L2 cache	1024KiB
L3 cache	22528KiB

GPU Information

Model name	Ampere A100
RAM	40 GiB
SM count	108
Shared mem / L1	192 KiB (*per SM*)
L2 cache	40 MiB

(a) Ampere Node

CPU Information

Model name	AMD EPYC 7A53
GPU count	4 (*8 logical*)
Sockets	1
Cores per socket	64
L1d cache	32 KiB
L1i cache	32 KiB
L2 cache	512 KiB
L3 cache	256 MiB

GPU Information

Model name	Instinct MI250X
RAM	128 GiB (*per module*)
CU count	110 (*per chip*)
Shared mem	64 KiB (*per CU*)
L1 cache	16 KiB (*per CU*)
L2 cache	8 MiB (*per chip*)

(b) LUMI Node

3.1 HPC System

The first system is a local cluster at the Technical University of Denmark, managed by DCC [9], which provides access to NVIDIA's A100 PCIE 40GB GPUs [16]. Only a single GPU is utilized for the experiments, with exclusive access to the node to minimize external contributors to noise and overhead.

The second system is the LUMI-G supercomputer [4], which is equipped with four AMD MI250X GPUs, each with a total of 128GB memory. The MI250X GPU is a multi-chip module containing two GPU dies, meaning that each node contain 8 logical GPU partitions where each pair, shares the 128GB memory. As we are using a single GPU for our experiments, this effectively means that only half of one MI250X is utilized.

A more detailed description of both systems is found in Table 1.

3.2 Compilers

The applications are developed using C++17 and are compiled with the `-O3` flag for all experiments. The specific version of each compiler is listed in Table 2.

Table 2. Compilers used on the compute nodes from Tables 1a and b. No commit is present for the NVC compiler as it has not been installed from Git. Additionally, both clang and cray are mapping to HIP rocm version 5.2.21153.

Compiler	Version	Commit
`clang DCC`	16.0.0	`710a834c4c822c5c444fc9715785d23959f5c645`
`clang LUMI`	16.0.0	`710a834c4c822c5c444fc9715785d23959f5c645`
`nvc`	22.5	-
`cray clang`	15.0.0	`324a8e7de6a18594c06a0ee5d8c0eda2109c6ac6`
`gcc`	13.0.0	`44baa34157cf81306be23eacece751aa020985d4`

The first compiler is the C language family front-end for the LLVM project *Clang* [12], which we use across both systems. We have compiled Clang to support target offloading according to the LLVM compile guide for OpenMP [13]. The second compiler is the `nvc++` compiler from NVIDIA, which is part of NVIDIA's HPC SDK [18] and is provided on the DCC system. We will refer to this compiler as *NVC*. The third compiler used is the `cray clang++` compiler, which is provided by the LUMI system as part of their Cray Compiling Environment *CCE* [14], which we will refer to as *Cray*. The last compiler used is the GNU Compiler Collection *GCC* [24]. GCC is also compiled to support target offloading, where we followed the guide *Offloading Support in GCC* [3].

3.3 Parallel Scan

In this section we will compare the manual work-sharing with the automatic loop based work-sharing on the DCC system. The two methods will be compared using the parallel scan algorithm, presented in Sect. 2.2. Additionally, an equivalent CUDA implementation is used to compare similar levels of complexity with an explicit kernel definition. Figure 1 illustrates the execution time for both the manual work-sharing, automatic work-sharing with `parallel for` and the `loop` construct based work-sharing introduced in OpenMP 5.0.

Manual work-sharing demonstrates substantially better performance for both the Clang and NVC compilers, and we see that Clang deliverers comparable results to the pure CUDA implementation for large arrays. Additionally, it is noteworthy that NVC achieves a speedup of approximately 7 with manual work-sharing for large arrays compared to the `parallel for` constructs. For small arrays, Clang exhibits decreased performance, potentially attributed to overhead between kernels, which we will further investigate in Sect. 3.5.

We encountered difficulties with the GCC compiler, as it did not allocate more than 28-32 threads per team. To prevent errors, we had to reduce the blocksize to 16 for GCC, alternative to 256 used for the remaining compilers. However, as mentioned in Sect. 2 this behavior is still compliant with the standard.

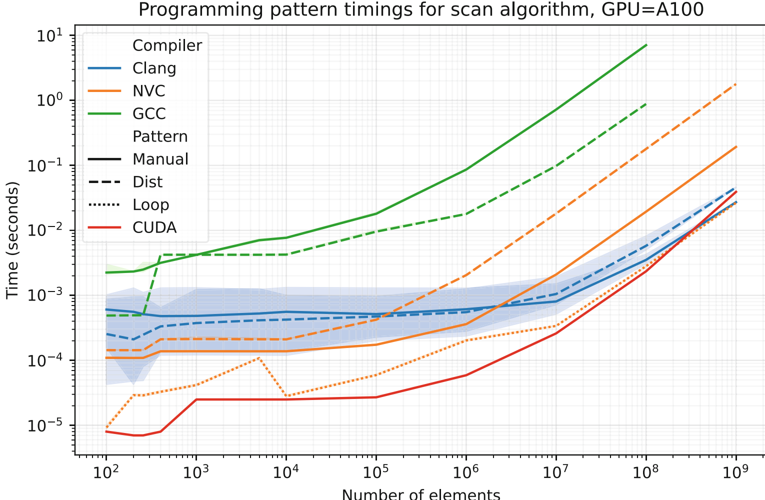

Fig. 1. Execution time for the 3 different parallelization methods across multiple compilers, including the equivalent CUDA implementation. All tests are performed on the DCC system, with A100 GPUs, and execution times include 95 percentile intervals for 200 runs.

The automatic work-sharing using `distribute` and `parallel for` yields lower performance across all array sizes for both NVC and Clang but demonstrates an improvement for GCC while also allowing the block-size to remain at 256. During the debugging process, it was found that GCC utilizes only 32 threads per team, deviating from the more ideal number of 128 threads used by both NVC and Clang. This clearly indicates improved performance stability when utilizing the work-sharing constructs, which further enables the use of much larger blocks which cannot be supported with the manual work-sharing or CUDA, without additional code complexity. However, the performance stability also comes with added overhead, which is indicated by the previously mentioned decreased performance for both NVC and Clang.

The OpenMP 5.0 `loop` construct demonstrates performance close to CUDA across all array sizes, and generally outperforms other patterns for NVC, with a speedup of almost 8 compared to the manual work-sharing. As earlier mentioned we only have support for this work-sharing construct in NVC, which limits further usage, but does show promise for future implementations of OpenMP target offloading.

In general, the manual work-sharing achieves better performance in our experiments, compared to the automatic based on the `distribute` and `parallel for` constructs. However, its effectiveness heavily relies on the implementation, leading to decreased portability across systems and compilers, as shown for the GCC compiler.

3.4 Radix Sort

Next, we examine the manual work-sharing using the four-way parallel radix sorting algorithm. The experiments will both run on the DCC and LUMI system. GCC is omitted due to problems related to the available number of threads in a teams region. The goal of the experiment is to give a better understanding of the expected performance for the tested compilers when using separated team and thread parallelism, to utilize the locally shared GPU memory.

The experiment consists of sorting an array of random numbers, with number of elements ranging from 100 to 10^8. Time is measured from the first target region is called, until the last target region completes and synchronizes. The sorting algorithm is run once as a warm-up to ensure that the run-time is fully initialized, before starting to measure the time. The resulting data is additionally verified after completion, to ensure correct behavior. Timings do not include allocation of data, as all allocations are handled before sorting is started.

Figure 2, illustrates the total execution time, using multiple compilers on both the A100 and MI250X GPUs. Similarly to the previous experiment we have an equivalent CUDA implementation measured on the same A100 GPUs, hinting to what execution time can ideally be expected from a very simple code without using any advanced features.

Clang still shows significant variation in performance for smaller arrays and shows an execution time that approaches the pure CUDA implementation for large arrays. Investigating the percentile interval of the Clang time, shows a relatively constant difference between the 0.01 percentile and 0.99 percentile, ranging from $0.010s$ to $0.012s$ over all sizes of the array larger than the block-size. The constant range of the variation indicates that it is not originating from the kernels themselves, but rather an overhead applied to each executed target region. We will investigate this further in the next section.

On LUMI, we see increased performance for Clang and similar performance for Cray. Clang show significantly reduced statistical variation between runs, compared to equivalent runs on the DCC system, with more than 30 times less variation for most array sizes. Whether the slightly better average performance for Clang on LUMI is due to the hardware, or due to a more efficient OpenMP run-time, is not possible to say from these experiments and is not examined further in this paper. However, it is still worth mentioning that Clang achieves better and clearer results on AMD, than on NVIDIA hardware.

NVC shows better performance with smaller arrays compared to Clang on A100, but falls behind around 10^5 elements, indicating a less efficient parallelization and mapping to the hardware. Similarly to the previous experiment, NVC does not show any large variation between runs and maintains a steady 10 times slowdown compared to the CUDA implementation across all array sizes. NVC could potentially achieve better performance using the `loop` construct, as observed in Fig. 1, where similar performance to CUDA is observed for some sizes.

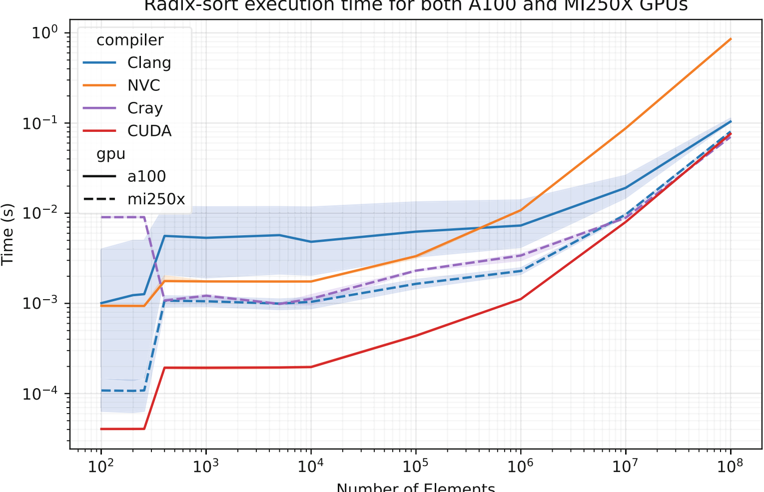

Fig. 2. Execution time of the radix sort implementation across a range of number of elements from 100 to 10^8 elements, with 95 percentile intervals for 200 runs. Experiments are conducted on both A100s from the DCC and MI250X on LUMI. The high variation for the Clang generated code is examined further in Sect. 3.5.

3.5 Profiling

In this section we perform a more in-depth examination of the kernel execution using profiling tools, to get a better understanding of the observed difference between NVC, Clang and CUDA on the DCC system.

The experiment have a similar setup as the previous experiment, where sorting of an array random numbers is performed, but here we fix the size to 10^6 elements for all tests. Profiling has been performed using `NVIDIA Nsight Systems version 2021.3.3.2-b99c4d6`, which is shipped with `CUDA 11.5`[1] and is using `nsys profile --trace=cuda <program><args>` for all tests.

Figure 3 illustrates the average execution time for each kernel included in the radix sorting algorithm, which also includes all the kernels from the scan algorithm. A significant increase in kernel execution time is measured for NVC, across all kernels which utilize the separated team and thread parallelization for utilizing the shared memory, with slowdowns of between 5 to 10 compared to Clang and CUDA. However, we measure better performance for the `AddBlockSums` kernel, achieving similar results to Clang and CUDA, indicating that the lost performance is due to the separated team and thread parallelization. Clangs kernel execution achieves similar result to the CUDA implementation, indicating

[1] CUDA version 11.5 is the highest version still fully supported by the version of Clang used.

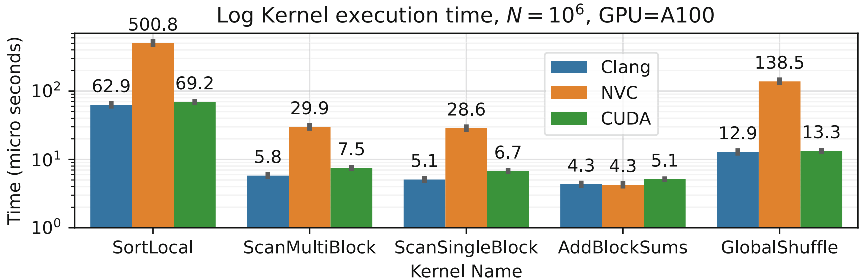

Fig. 3. Profiling kernel execution time for NVC, Clang and CUDA, using A100, with 5 runs per test, with each kernel being executed multiple times per run.

that the manual work-sharing is working ideally for Clang and that the variation observed in earlier experiments is not contributed to by the kernel execution itself.

By examining the profiling data further, we reveal that Clang utilizes synchronization with the host, resulting in additional latency from the communication between the host and device. This should not be necessary as the `nowait` and `depend` clauses are used to run the target regions asynchronously. This does correlate with the increase in time between kernels observed in Fig. 4, and could explain the large variation in execution time from earlier experiments. In contrast, both NVC and CUDA launch all kernels asynchronously and handles scheduling on the device, effectively removing the latency added by the communication.

Figure 4 illustrates this idle-time between kernels and show a significant amount for Clang with an average of 46.5 μs, compared to 9.2 μs observed for NVC and 5.0 μs observed for CUDA. As the overall kernel execution time is significantly better for Clang, compared to NVC, it is with high probability that the lost performance and large variation can be contributed to the scheduling and synchronization of the kernels.

We theorize that improving the scheduling in Clang has potential to yield a significant speedup for the overall run-time. To determine the potential, the combined execution time of all kernels is calculated while taking into account an estimated improved idle-time for each kernel. Realistic values for the estimated idle-time can be derived from the average values obtained for NVC or CUDA. Based on these specific experiments, it is found that Clang could achieve a speedup of 2.37 with an idle-time similar to NVC, and a 3.24 speedup with an idle-time similar to CUDA. It is important to note that these results are highly application-specific. Nonetheless, they highlight the potential for substantial performance enhancement in Clang, despite already achieving impressive performance without adding much complexity for the GPU target offloading.

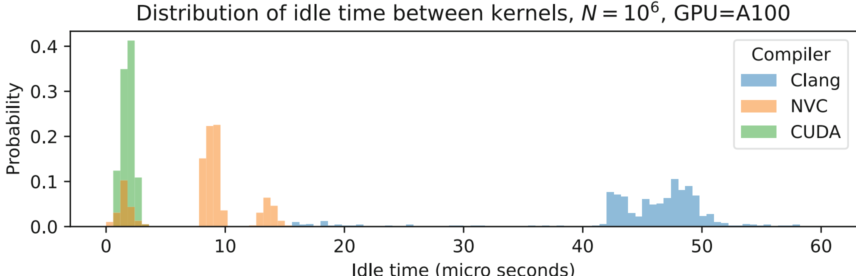

Fig. 4. Profiling idle time between kernels for code generated with NVC, Clang and CUDA, using A100. The idle time is found by the difference in start-time between the consecutive kernels, subtracted by the kernel execution time.

4 Related Work

Previous work by Chapman et al, [5,6] and Daley et al, [7] also report poor performance for `distribute` and `parallel for` constructs when using separate team and thread parallelization, similar to what we observe in Sect. 3.3. Davis et al. [8] further confirm this behavior and additionally show substantial performance improvement using manual team distribution compared to automatic distribution with the `distribute` construct. Chapman et al. [5,6], further show significant improvement using the OpenMP 5.0 `loop` directive, which was also observed in Sect. 3.3.

Rydahl et al. [22] investigated multi-GPU programming using target offloading for stencil operation, similarly performing asynchronous kernel execution using the `nowait` clause. Here multiple target regions were running in parallel allowing overlapping kernel execution, which could help reduce latency related to host-side synchronization.

Tian et al. [25], suggest extension to OpenMP allowing high performing target regions, including allocators for shared memory and synchronization intrinsic within teams. Talaashrafi et al. [23] suggest to automate utilization of shared memory for pre-fetching of read-only data, which could reduce the need for blocking in some algorithms.

Lastly, Zegarra et al. [27], propose a new scan clause for OpenMP, with similar performance as direct programming in OpenCL, but with much less design effort, essentially making it possible to implement our first application using a single OpenMP target construct. Introducing local scanning within teams would additionally help to significantly reduce the design effort for the radix sorting, as the majority of the complexity is related to a local Blelloch scan implementation. Having a local scan clause would additionally allow highly optimized OpenMP implementations taking full advantage of supported architecture features.

5 Conclusions

In this paper, we examined manual team and thread work-sharing and automatic work-sharing based on the `distribute` and `parallel for` constructs, for OpenMP target offloading. We ran experiments using two algorithms heavily relying on shared memory and synchronization within teams. We show larger speedups than 9 for NVC and 1.5 for Clang, when using manual work-sharing compared to the automatic work-sharing constructs `distribute` and `parallel for` and more than a speedup of 7, when using the `loop` construct, compared to the manual work-sharing for NVC.

For both parallel scan and radix sorting we found that Clang suffers from significant overhead for all target regions when running on A100 GPUs, which is revealed to most likely be caused by synchronization through the host, despite using the `nowait` and `depend` clauses, where applications compiled with NVC were able to launch all kernels from the host and handle execution asynchronously on the GPU. Through profiling we show that Clangs kernel execution is comparable to the simple pure CUDA implementation, but that a significant increase in idle-time between kernels is present compared to other compilers. This results in poor performance when applications consist of many smaller target regions, but have diminishing impact when the problem-size increase. Lastly, we estimate a potential speedup of 3 for the profiled applications compiled with Clang, if a scheduling similar to NVC and CUDA can be achieved.

Acknowledgement. This work was partially supported by DeiC National HPC (g.a. DeiC-DTU-N5-20230033) and by the "Compiler development" project (g.a. DeiC-DTU-N5-20230033).

We acknowledge Danish e-infrastructure Cooperation (DeiC), Denmark for awarding this project access to the LUMI supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CSC (Finland) and the LUMI consortium through Danish e-infrastructure Cooperation (DeiC), Denmark, "Compiler development", DeiC-DTU-N5-20230033.

Lastly, we acknowledge DCC [9] for providing access to the A100 GPUs used for experiments as well as interactive environments used throughout development.

References

1. Adinets, A., Merrill, D.: Onesweep: a faster least significant digit radix sort for gpus. arXiv preprint arXiv:2206.01784 (2022). https://doi.org/10.48550/arXiv.2206.01784
2. Blelloch, G.E.: Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (1990)
3. Burnus, T.: Offloading support in GCC (2023). https://gcc.gnu.org/wiki/Offloading. Accessed 17 May 2023

4. Center for Science: LUMI-G documentation, GPU nodes. https://docs.lumi-supercomputer.eu/hardware/lumig/ (2023). Accessed 15 May 2023
5. Chapman, B., et al.: Outcomes of openMP hackathon: openMP application experiences with the offloading model (part I). In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 67–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_5
6. Chapman, B., et al.: Outcomes of openMP hackathon: openMP application experiences with the offloading model (part II). In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 81–95. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_6
7. Daley, C., Ahmed, H., Williams, S., Wright, N.: A case study of porting HPGMG from CUDA to openMP target offload. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 37–51. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_3
8. Davis, J.H., Daley, C., Pophale, S., Huber, T., Chandrasekaran, S., Wright, N.J.: Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In: Bhalachandra, S., Wienke, S., Chandrasekaran, S., Juckeland, G. (eds.) WACCPD 2020. LNCS, vol. 12655, pp. 25–44. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-74224-9_2
9. DTU Computing Center: DTU Computing Center resources (2022). https://doi.org/10.48714/DTU.HPC.0001
10. Ha, L., Krüger, J., Silva, C.T.: Fast four-way parallel radix sorting on GPUs. Comput. Graph. Forum **28**(8), 2368–2378 (2009). https://doi.org/10.1111/j.1467-8659.2009.01542.x
11. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: GPU Gems 3, pp. 851–876. Addison-Wesley Professional (2007)
12. LLVM: Clang: a c language family frontend for LLVM (2023). https://clang.llvm.org/. Accessed 26 May 2023
13. LLVM: Support, getting involved, and FAQ (2023). https://openmp.llvm.org/SupportAndFAQ.html. Accessed 17 May 2023
14. LUMI: Cray compilers (2023). https://docs.lumi-supercomputer.eu/development/compiling/cce/. Accessed 26 May 2023
15. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. Tech. Rep. NVR-2016-002, NVIDIA (2016)
16. NVIDIA: Nvidia a100 tensor core gpu architecture, unprecedented acceleration at every scale (2020). https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. Accessed 15 May 2023
17. NVIDIA: CUDA toolkit documentation v11.5.0 (2023). https://docs.nvidia.com/cuda/archive/11.5.0/. Accessed 26 May 2023
18. NVIDIA: Nvidia HPC SDK documentation (2023). https://docs.nvidia.com/hpc-sdk/archive/22.7/. Accessed 26 May 2023
19. OpenMP Architecture Review Board: OpenMP (2023). https://www.openmp.org/. Accessed 15 May 2023
20. OpenMP Architecture Review Board: Openmp application programming interface version 4.0 (2023). https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. Accessed 15 May 2023
21. OpenMP Architecture Review Board: OpenMP application programming interface version 5.0 (2023). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accessed 15 May 2023

22. Rydahl, A., Gammelmark, M., Karlsson, S.: Feasibility studies in multi-GPU target offloading. In: Klemm, M., de Supinski, B.R., Klinkenberg, J., Neth, B. (eds.) OpenMP in a Modern World: From Multi-device Support to Meta Programming. IWOMP 2022. Lecture Notes in Computer Science, vol. 13527, pp. 81–93. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15922-0_6
23. Talaashrafi, D., Maza, M.M., Doerfert, J.: Towards automatic openMP-aware utilization of fast GPU memory. In: Klemm, M., de Supinski, B.R., Klinkenberg, J., Neth, B. (eds.) OpenMP in a Modern World: From Multi-device Support to Meta Programming. IWOMP 2022. Lecture Notes in Computer Science, vol. 13527, pp. 67–80. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15922-0_5
24. The GCC team: Offloading support in GCC (2023). https://gcc.gnu.org/. Accessed 26 May 2023
25. Tian, S., Chesterfield, J., Doerfert, J., Chapman, B.: Experience report: writing a portable GPU runtime with OpenMP 5.1. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 159–169. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_11
26. Wu, K., Truong, N., Yuksel, C., Hoetzlein, R.: Fast fluid simulations with sparse volumes on the GPU. Comput. Graph. Forum **37**(2), 157–167 (2018). https://doi.org/10.1111/cgf.13350
27. Zegarra, M., Pereira, M., Martorell, X., Araujo, G.: Automatic scan parallelization in openmp. In: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp. 85–90. IEEE (2017). https://doi.org/10.1109/SBAC-PADW.2017.23
28. Zhang, W., Myers, A., Gott, K., Almgren, A., Bell, J.: AmReX: block-structured adaptive mesh refinement for multiphysics applications. Int. J. High Perform. Computing Applications **35**(6), 508–526 (2021). https://doi.org/10.1177/10943420211022811