Received: 15 January 2020 | Revised: 26 May 2020 | Accepted: 13 August 2020

DOI: 10.1002/cpe.6016

SPECIAL ISSUE PAPER

WILEY

An in-depth introduction of multi-workgroup tiling for improving the locality of explicit one-step methods for ODE systems with limited access distance on GPUs

Matthias Korch | Tim Werner

Department of Computer Science, University of Bayreuth, Bayreuth, Germany

Correspondence
Tim Werner, Department of Computer Science, University of Bayreuth, 95440 Bayreuth, Germany.
Email: werner@uni-bayreuth.de

Funding information
Deutsche Forschungsgemeinschaft, Grant/Award Number: KO 2252/3-2

Summary

This article considers a locality optimization technique for the parallel solution of a special class of large systems of ordinary differential equations (ODEs) by explicit one-step methods on GPUs. This technique is based on tiling across the stages of the one-step method and is enabled by the special structure of the class of ODE systems considered, that is, the *limited access distance*. The focus of this article is on increasing the range of access distances for which the tiling technique can provide a speedup by joining the memory resources and the computational power of multiple workgroups for the computation of one tile (*multi-workgroup tiling*). In particular, this article provides an extended in-depth introduction and discussion of the multi-workgroup tiling technique and its theoretical and technical foundations together with a new tuning option (mapping stride) and new experiments. The experiments performed show speedups of the multi-workgroup tiling technique compared with traditional single-workgroup tiling for two different Runge–Kutta methods on NVIDIAs Kepler and Volta architectures.

KEYWORDS

GPUs, hexagonal tiling, limited access distance, multi-workgroup tiling, ODE methods, RK methods, trapezoidal tiling

1 | INTRODUCTION

Many scientific simulations use systems of differential equations as mathematical models to approximate phenomena of the real world. This article considers initial value problems (IVPs) of systems of ordinary differential equations (ODEs):[1]

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t \in [t_0, t_e]. \tag{1}$$

The classical numerical solution approach, which is also used by the methods considered in this article, applies a time-stepping procedure that starts the simulation at time t_0 with initial state $\mathbf{y}_0$ and performs a series of time steps $t_\kappa \to t_{\kappa+1}$ for $\kappa = 0, 1, 2, \ldots$ until the final simulation time t_e is reached. At each time step κ, by applying the right-hand-side (RHS) function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, a new simulation state $\mathbf{y}_{\kappa+1}$ is computed which approximates the exact solution function $\mathbf{y}(t)$ at time $t_{\kappa+1}$.

We consider the parallel solution of IVPs by explicit one-step methods on GPUs, which—in contrast to implicit and multi-step methods—only use one input approximation $\mathbf{y}_\kappa$ and a number of evaluations of the RHS function to compute the output approximation $\mathbf{y}_{\kappa+1}$. Since for many IVPs ODE methods are memory bound, optimizing the locality of memory references is important. To reduce the time-to-solution on GPUs, we use the

well-known technique of *kernel fusion*, which exploits the on-die memories of the GPU (caches, scratchpad, registers) to increase data reuse. In our previous work,[2] we focused on kernel fusion for problems with an arbitrary coupling of the RHS. Since this allowed the fusion of different stages of the method only if they were independent, we lifted this restriction in a subsequent work[3] by a specialization in problems with a limited access distance, which commonly arise, for example, from stencil-based problems (e.g., partial differential equations [PDEs] discretized by the method of lines) or block-structured electrical circuits. By exploiting this special structure, we could derive tilings across the stages where each tile is processed by a single workgroup. This led to improved performance, but only for small access distances.

In our previous work on multi-workgroup tiling,[4] we extended the single-workgroup tiling approach across the stages to a multi-workgroup tiling, where several workgroups and hence multiple GPU cores collaborate on each tile to improve the performance for larger access distances. While other works have already employed multi-core tiling for CPUs, to the best of our knowledge, nobody has yet applied multi-workgroup tiling on GPUs to increase data reuse of ODE methods with limited access distance. Our multi-workgroup tiling is implemented as part of an automatic framework that can generate CUDA or OpenCL kernels for different tiling parameters automatically. We also presented a detailed experimental evaluation, which shows that the maximum access distance for which a speedup can be obtained could be increased by two orders of magnitude.

This special issue article is based on our previous work on multi-workgroup tiling,[4] but extends it by the following:

- a more detailed introduction of multi-workgroup tiling and the theoretical and technical foundations thereof,
- a detailed description of the fusion of dependencies and memory optimizations employed for multi-workgroup tiling,
- an introduction of three strides for mapping workitems to system components and an experimental evaluation of those strides,
- theoretical considerations on tile heights and widths for multi-workgroup tiling,
- a detailed description of multi-workgroup barriers and experiments evaluating the performance of those barriers,
- re-done experiments on the NVIDIA Volta GPU with an improved version of our framework yielding slightly better results,
- experiments for evaluating the performance of multi-workgroup tiling on the NVIDIA Kepler GPU and AMD Polaris GPU,
- and additional detailed profiling.

The remaining article is structured as follows:

- *Section 2:* The background required for understanding this article is introduced, that is, the structure of linear one-step methods, the limited access distance, and modern GPU architectures.
- *Section 3:* Discussion of the related work on this topic.
- *Section 4:* A detailed introduction of multi-workgroup tiling is given, that is, the theoretical and technical foundations thereof, the fusion of dependencies and the memory optimizations for multi-workgroup tiling, and the required synchronization across multiple workgroups by multi-workgroup barriers.
- *Section 5:* An experimental evaluation of multi-workgroup tiling is performed for two selected RK methods, that is, Bogacki–Shampine 2(3) with four stages and Verner 5(6) with eight stages, on two different generations of NVIDIA GPUs, Kepler and Volta, and on an AMD Polaris GPU.

2 | BACKGROUND

2.1 | Structure of explicit one-step methods

Explicit linear ODE methods have in common that they do not require the solution of a nonlinear system of equations, but only a number of evaluations of the RHS function and linear combinations of the function results. The most general class of explicit one-step methods is the class of explicit RK methods, which also covers extrapolation methods and explicit iterated RK methods.[1] At each time step κ, they compute the output approximation $\mathbf{y}_{\kappa+1}$ from the input approximation $\mathbf{y}_\kappa$ by a sequence of s intermediate stages such that for $i = 1, \dots, s$

$$\mathbf{Y}_i = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{i-1} a_{ij} \mathbf{F}_j, \tag{2}$$

$$\mathbf{F}_i = \mathbf{f}(t_\kappa + h_\kappa c_i, \mathbf{Y}_i), \tag{3}$$

where $A = (a_{ij}) \in \mathbb{R}^{s \times s}$ is a method specific strictly lower triangular matrix, and $\mathbf{c} \in \mathbb{R}^s$ is a method specific coefficient vector. The output approximation is then obtained by

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} b_j \mathbf{F}_j \tag{4}$$

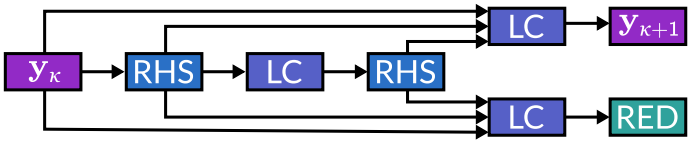

FIGURE 1 Data flow graph of a time step of the Heun–Euler method, a popular RK method. This data flow graph illustrates that the time step of a RK method can be described by three basic operations: Evaluations of the right-hand side (RHS), linear combinations (LC), and reduction operations (RED)

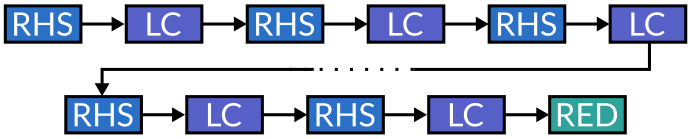

FIGURE 2 A dependency chain of RHS → LC links, which makes up or is a part of many one-step or multi-step methods. Note that the LC operations in the chain also take previous RHS operations or LC operations as argument. However, those dependencies are omitted in this figure

using the method specific weights $\mathbf{b} \in \mathbb{R}^s$.

To perform the step size control efficiently, some explicit RK methods (embedded RK methods) provide additional weights $\hat{\mathbf{b}} \in \mathbb{R}^s$ that can be used to compute the error vector

$$\mathbf{E} = h_\kappa \sum_{j=1}^{s} (\hat{b}_j - b_j)\mathbf{F}_j, \tag{5}$$

which allows a cheap estimation of the local error by

$$\epsilon_{\kappa+1} = \|\mathbf{E}\|. \tag{6}$$

Remarks. Please note that for some methods A, $\mathbf{b}$, and $\hat{\mathbf{b}}$ are not fully populated, thus thinning out the dependencies between the computations, which also can be exploited for improving the runtime.[2]

Hence, the time step of an explicit RK method can be described by a data flow graph consisting of three different types of basic vector operations (e.g., see Figure 1 for the data flow graph of the Heun–Euler method):

- Linear combinations (LC) for Equations (2), (4), and (5),
- RHS function evaluations (RHS) for Equation (3), and
- Reduction operations (RED) for the norm in Equation (6).

A data flow graph with the same basic operations can also be used to describe the class of general linear methods. In the case of an explicit method, the data flow graph is cycle-free, otherwise the method is implicit. For one-step methods, only a single system state is carried over from the preceding to the current time step. However, for multi-step and general linear methods, not only a single but several system states or temporal derivatives are carried over not only from the last but from several preceding time steps to the current time step. The dependencies from preceding time steps to the current time step can be modeled in the data flow graph by time step distances.[5]

Although the data flow graph of an explicit one-step method may have an arbitrary structure following the above mentioned rules, for many explicit one-step methods, such as explicit Runge–Kutta (RK) methods, parallel iterated RK (PIRK) methods, and explicit extrapolation methods,[1] its transitive reduction is or contains a dependency chain of RHS → LC links with an optional reduction operation in the end of the chain for step size control, as it is shown in Figure 2. In addition to being dependent on the preceding RHS evaluation, linear combinations in this chain may also depend on other previous RHS evaluations and linear combinations.

2.2 | Limited access distance

Since in the general case, the right-hand side may be arbitrarily coupled, a LC → RHS dependency acts as a global barrier restricting data reuse across this dependency. In this article, we lift this restriction by a specialization in ODE systems with a limited access distance, that is, systems where the

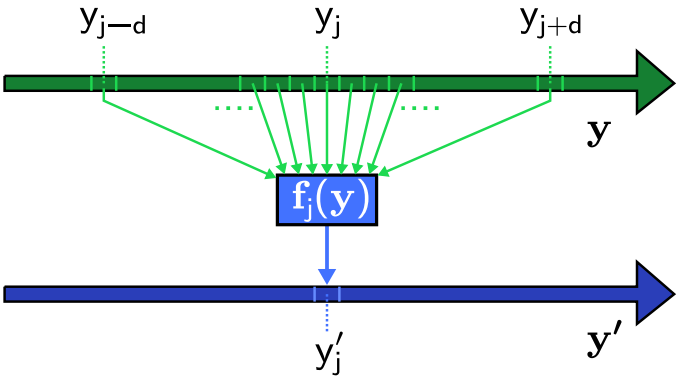

FIGURE 3 Illustration of the dependencies of the evaluation of the RHS for a problem with limited access distance. For such problems, in order to evaluate the jth component of the RHS f, only the components in the limited access distance d around j are required from the argument vector $\mathbf{y}$, which is the subset $\{y_{j-d}, \ldots, y_{j+d}\}$

evaluation of component j of the RHS, $f_j(t, \mathbf{y})$, only accesses components of $\mathbf{y}$ located nearby j. More precisely, the access distance $d(\mathbf{f})$ is the smallest value d, such that each component function $f_j(t, \mathbf{y})$, $j = 1, \ldots, n$, accesses only the subset $\{y_{j-d}, \ldots, y_{j+d}\}$ of the components of the argument vector $\mathbf{y}$ (see Figure 3). The access distance of $\mathbf{f}$ is limited if $d(\mathbf{f}) \ll n$. ODE systems with limited access distance commonly arise, for example, from stencil-based problems (e.g., partial differential equations [PDEs] discretized by the method of lines) or block-structured electrical circuits.

2.3 | Modern GPU architectures

Although modern GPU architectures from Intel, AMD and NVIDIA differ in several small details, their basic designs are very similar.[6] Modern GPUs typically consist of many cores (also called "compute units" by OpenCL and AMD and "streaming multiprocessors" by NVIDIA) and a small shared L2-cache. Additionally, GPUs use DRAM as main memory, which is accessed via a global memory space. The cores have a wide SIMD architecture and use fine-grained interleaved multithreading. Each core typically has small L1 data caches, some scratchpad memory, and a large register file. Consequently, data reuse via the register file is very important on GPUs. This register file is usually large enough to store all automatic variables for most kernels. Consequently automatic variables in the source code of a kernel typically translate to registers. Nevertheless register spilling may still occur for kernels that have a large working set of automatic variables. Unfortunately, the small caches of modern GPUs make register spilling expensive. However, GPUs work against register spilling by allowing a kernel to allocate more registers per thread, which in the process reduces the maximum number of concurrent threads per core.

In order to execute a kernel, the host program running on the CPU has to specify an iteration space consisting of a number of workgroups (CUDA thread blocks) and a number of workitems (CUDA threads) per workgroup. The GPU then dispatches the workgroups to the cores. Each workgroup is processed by a single core, but a core may process several workgroups simultaneously. As a consequence, a workgroup can also only exploit the memory resources and the computational power of a single core. Each core then maps the workitems 1 : 1 to the SIMD lanes of its concurrent SIMD threads (warps, wavefronts). In order to cooperate with each other, workitems of the same workgroup may use the scratchpad memory supplied by each core and workgroup-wide barrier instructions. Note that NVIDIA and AMD advertise the SIMD lanes of a vector FPU as "CUDA cores" and "streaming cores," respectively. This, however, contradicts the commonly accepted definition of a core to which we stick in this article.

GPUs typically support 1, 2, 4, 8, and 16 byte gather and scatter instructions on global memory, where the respective amount of data is moved from/to each SIMD lane. If the SIMD lanes in a scatter or gather instruction request data from the same cache line, GPUs avoid redundant transfers by coalescing the requests to a single transfer whenever possible. Nevertheless, misaligned and uncoalesced memory accesses are often smoothed out by the caches of the GPU.

The L2-cache of a modern GPU consists of several parallel partitions, with the global memory space being mapped to the partitions in a block cyclic manner. Consequently, L2 requests for different partitions are served in parallel, while L2 requests for the same partition are served with an exclusive read and write policy. Additionally, each partition has an atomic unit for performing atomic operations. Hence, in order to exploit the bandwidth of all cache partitions or atomic units, the memory accesses or atomic operations simultaneously processed by the GPU have to spread over a larger range of addresses.

3 | RELATED WORK

Kernel fusion is one of the most important state-of-the-art techniques to optimize data reuse on GPUs, and several domain-specific approaches have been proposed. For example, an automatic approach to reduce the power consumption of GPUs via kernel fusion was proposed,[7] the application to stencil computations was also considered,[8] and finally an automatic kernel fusion for nested map and reduce problems on CUDA GPUs by a

source-to-source compiler was introduced.[9] Kernel fusion with multi-workgroup barriers also was considered for dynamic programming algorithms, bitonic sort, and FFT.[10,11] However, the multi-workgroup barriers were only used to reduce the overhead of kernel launches and not to share data between different workgroups.

Tiling is a classical locality optimization technique used on CPUs and GPUs to break down the working sets of loops so that the resulting tiles fit into faster levels of the memory hierarchy. One important theoretical model that allows to determine possible tilings for loops with dependencies is the polyhedral model. The polyhedral model is used to develop a generic algorithm that can automatically create trapezoidal tilings of stencil codes for GPUs,[12] which was improved by a tiling scheme using hexagonal tiling along the time dimension and one spatial dimension and classical tiling along the remaining spatial dimensions.[13] Both tiling algorithms were implemented in the PPCG polyhedral code generator and were evaluated for several standard stencils.[12,13] However, up to now, PPCG can only generate single-workgroup tilings. The collaboration of several CPU cores on a single tile (*multi-core tiling*) was exploited for stencil codes.[14] Some approaches aiming at tiling for stencil code start with a description of the stencil problem to be solved and perhaps also a description of the numerical algorithm to be used in a domain specific language (DSL). Examples are PATUS,[15] Pochoir,[16] Halide,[17] and ExaStencils.[18] Their motivation is a step-by-step generation of program code from a high-level description which allows the application of optimizations at different levels of abstraction, like automatic parallelization and tiling.

Locally recursive nonlocally asynchronous (LRnLA) algorithms[19] also aim at tiling of stencil codes. For this, they split up the iteration domain (several spatial domains and one time domain) recursively into polyhedral tiles, where the fine-grained tiles correspond to the locally recursive property of the algorithm, thus aiming to increase data reuse in a cache oblivious manner. By contrast, the coarse grained tiles correspond to the nonlocally asynchronous property of the algorithm, that is, giving a set of tasks which may be executed by multiple threads or processors. LRnLA algorithms were used to speed up the simulation of light propagation in an optical whispering gallery mode microresonator on CPUs,[20] and the simulation of multi-phase fluid flows on GPUs.[21] In contrast to our work based on ODE methods and problems with limited access distance, LRnLA algorithms are specialized in stencil problems. Hence, LRnLA algorithms exploit several spatial dimensions for their tiling, while our work exploits the system dimension of the ODE system only. Also, throughout our work we develop an holistic process to generate optimized tilings for an arbitrary ODE method step by step starting with a data flow graph representation and ending with executable code. As a proof of concept, we have implemented this holistic process by an automated prototype framework.

ODE methods, in contrast to typical PDE and stencil approaches, compute several stages within one time step, and often step size control takes the effect of a barrier between time steps. Therefore, locality optimizations for ODE methods often focus on the loop structure inside one time step. For example, generic loop tiling of the internal and external stages of peer methods along the system dimension was investigated on CPUs.[22] If a limited access distance can be exploited, additional loop transformations are possible. For example, for explicit RK methods, a time-skewing strategy for Adams–Bashforth methods provides a temporal tiling of time steps.[23]

The *application of kernel fusion to ODE methods on GPUs* for general ODE systems was also considered.[2] For those systems it is only allowed to fuse RHS → LC, RHS → RED, LC → LC and LC → RED dependencies, while a global barrier is required for each LC → RHS dependency. Fusion of LC → RHS dependencies by exploiting the limited access distance was considered to realize hexagonal and trapezoidal *tiling across the stages* with single-workgroup tiling.[3] Finally, hexagonal and trapezoidal *tiling across the stages and time steps* with single-workgroup tiling was also considered.[5]

4 | MULTI-WORKGROUP TILING OF EXPLICIT ONE-STEP METHODS

4.1 | From single- to multi-workgroup tiling

Both of our approaches for tiling across the stages for ODE methods, single and multi-workgroup tiling, are based on the data flow representation of ODE methods and specialize on problems with *limited access distance*. While even for those problems a LC → RHS dependency is still not fusible to a single kernel, a distributed fusion to a pair of kernels is possible. This distributed fusion leads to two parameterized 2-dimensional tiling schemes along a dependency chain (trapezoidal and the hexagonal tiling), as depicted in Figure 4, where the tiles being independent of each other, for example, for the trapezoidal tiling the shrinking trapezoids in a row, are processed in parallel by the same 1D kernel, while the 1D kernels processed one after another in their topological order. The tiling schemes have the tile width (tile size along the system dimension) and tile height (tile size along the time dimension) as parameters.

In *single-workgroup tiling*,[3] each tile is processed by a single-workgroup only and a tile is only able to use the private on-die memory resources and computational power of *one* GPU core. Consequently, many tiles are required to saturate the GPU, which is why even medium tile sizes cause massive register spilling and cache thrashing. Since larger access distances also require larger tile widths for the tiling to be efficient or even possible, single-workgroup tiling becomes inefficient for quite small access distances.

By contrast, in *multi-workgroup tiling* (see Figure 5), a group of n consecutive workgroups, which are assumed to be executed simultaneously, collaborates on the same tile. In such a tile, the components of a basic operation are evaluated by the workitems of the workgroups in parallel, while the basic operations in the dependency chain are processed one after another. As a consequence, several workgroups of the same 1D kernel and hence

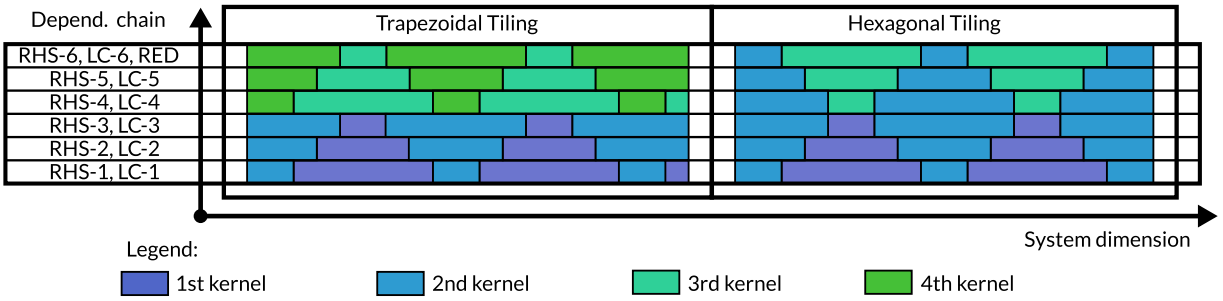

FIGURE 4 Tiling schemes along the dependency chain of LC → RHS links. In both schemes the tiles of a kernel are processed in parallel either by a single workgroup each (single-workgroup tiling) or by multiple workgroups each (multi-workgroup tiling), and the kernels are executed one after another in their topological order

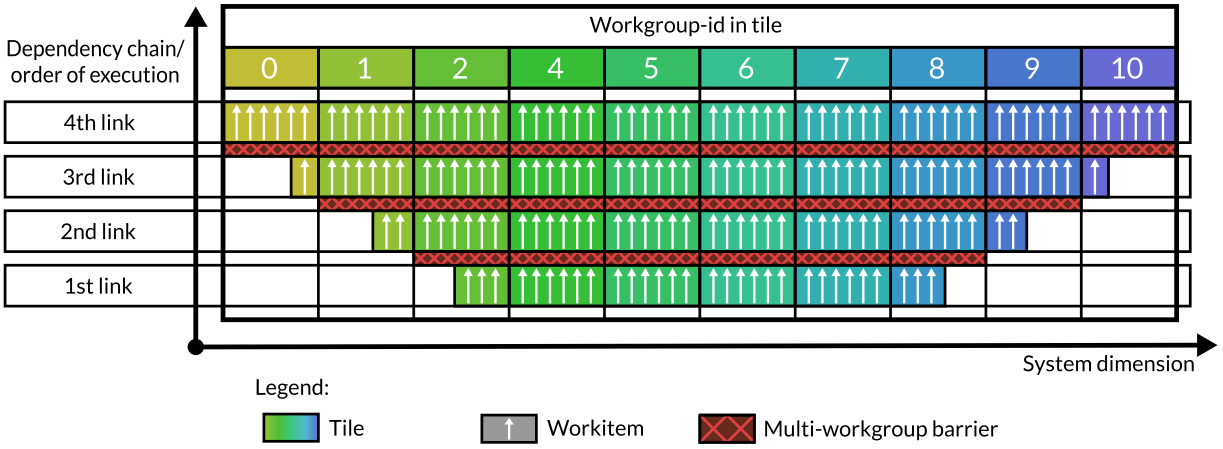

FIGURE 5 A multi-workgroup tile along a fused RHS → LC dependency chain. In such a tile consisting of several cooperating workgroups, the workitems compute their assigned components concurrently for each link in the dependency chain. At the end of each link, a multi-workgroup barrier is called, spanning the workgroups of the tile, so that there are no data races for the fused LC → RHS dependencies

multiple GPU cores collaborate on each tile. This in the process makes larger tile widths more efficient, since now a tile can use the computational power and memory resources of several or even all cores and fewer tiles are required to saturate the GPU.

Obviously, one can choose the amount of workgroups per tile so that all the workgroups concurrently processed by the GPU to cooperate on a single tile, but one can also choose a smaller amount of workgroups per tile so that the GPU processes several multi-workgroup tiles concurrently. Choosing more concurrent tiles may cause less synchronization overhead. Additionally, more concurrent tiles also offer a less synchronized instruction mix during execution, which may increase the overall resource utilization. However, choosing more concurrent tiles also decreases the tile size and thus increases the amount of data transferred via DRAM. As a consequence, many concurrent small tiles are expected to perform better for small access distances, while few large tiles or even a single very large tile are expected to perform best for large access distances.

To avoid deadlocks for our implementation of multi-workgroup tiling, we additionally require that, first, the workgroup scheduler dispatches the workgroups to the cores in an ascending order starting with the 0th workgroup, and, second, a workgroup is only assigned to a core if a core has a free slot for this workgroup. Unfortunately, GPU vendors typically do not guarantee any concurrent execution or the order of execution for different workgroups of the same kernel on their GPUs. However, the restrictions above hold true on all NVIDIA and AMD GPUs considered. This requirement can be avoided by choosing a persistent threading approach as an alternative, that is, launching only that many workgroups so that the GPU is saturated and having each group of workgroups process several multi-workgroup tiles one after another. As a drawback, this persistent threading approach lacks synchronization so that the tiles concurrently processed by the GPU may spread over a larger range of memory.

4.2 | Fusion of dependencies

Because we assign each workitem in a tile the same components for all basic operations, which the tile computes, the fusion of RHS → LC, RHS → RED, LC → LC, and LC → RED dependencies does not cause dependencies between the workitems of the multi-workgroup tile. Thus, they can be fused completely via the register file.

However, dependencies between different workitems and workgroups in a tile occur, if a LC → RHS dependency is fused. Hence, this fusion requires communication between the workgroups of the tile and the synchronization of communication steps to avoid data races. Because of that, a multi-workgroup tile requires three steps to evaluate a fused LC → RHS dependency:

1. The workgroups of the tile write those components of the linear combination which the workgroups have computed beforehand and which the workgroups later require to evaluate the RHS from the registers to a global memory buffer, which is private to this tile.
2. The workgroups of the tile execute a barrier spanning across all the workgroups of the tile to ensure that all writes to the buffer have finished and are visible.
3. The workgroups of the tile evaluate the RHS and, for that purpose, read from the buffer.

Depending on the position of the evaluation of the RHS in the tile, the tile has to use a different implementation of the RHS:

- In one implementation, all the data read by the RHS is a consecutive range in memory, which may be a vector residing in DRAM or a buffer for fusing a LC → RHS dependency.
- In the other implementation, there are two different memory ranges: One range is the halo of the tile, which is a vector residing in DRAM, while the other range is the center, that is, the buffer for transferring the data along a LC → RHS dependency.

4.3 | Synchronization across multiple workgroups

To implement the synchronization in a multi-workgroup tile, we need a barrier spanning across several workgroups of the same kernel. Unfortunately, on modern GPUs signal-based synchronization is only supported between the threads of a single-workgroup via a workgroup wide barrier instruction, but not across multiple workgroups. Consequently, one has to implement a barrier across several workgroups via busy waiting. We have implemented two barrier variants[11] and also considered the barrier from the NVIDIA cooperative groups library:

- *Barrier-A (barrier with a shared atomic counter):* In this version, in order to reduce the pressure on the shared atomic counter, only a designated master workitem per workgroup increments and busily waits on an atomic counter, while the remaining workitems of the workgroup are synchronized with the master workitem upon entering and leaving the tile barrier via workgroup wide barriers.
- *Barrier-B (barrier with a master workgroup):* In this version, upon entering the barrier each workgroup of the tile sets its own entered flag from false to true. Then, all workgroups of the tile, except a designated master workgroup, start waiting on their private exit flags, which are initially set on false. In the meanwhile the master workgroup waits busily for all other workgroups having set their entered flag to true. After all entered flags have been set, the master workgroup starts to set the exit flags to true so that all remaining workgroups can leave the tile barrier.
- *Barrier-C (barrier from NVIDIA's cooperative groups Library):* This barrier can only be used to synchronize all workgroups of a kernel. As a consequence, this barrier only allows one concurrent tile on the GPU and requires a persistent threading approach. Moreover, it is only available for Pascal, Volta and Turing GPUs. While NVIDIA does not document any implementation details of this barrier, its low performance suggests that it is a not optimally implemented software barrier.

Since the L2-cache of a GPU consists of several parallel partitions, where each partition has an exclusive read/write/atomic policy, both the atomic increments and the reads of the atomic counter of Barrier-A are serialized. However, the counters of different concurrent barriers may be accessed in parallel, given they have a suitable stride for harnessing the parallelism of the L2-cache. By contrast, the flags of the Barrier-B may be read and written concurrently, also given that they have a suitable stride. Furthermore, Barrier-B also has a higher latency caused by the master workgroup first gathering and then broadcasting the flags. Consequently, Barrier-A is expected to perform better for tiles consisting of few workgroups, while the Barrier-B is expected to perform better with tiles consisting of many workgroups.

In order to compare the performance and the scaling of the three barrier types, we have written a small microbenchmark. This benchmark measures the throughput of n concurrent barriers with m workgroups per barrier, by launching $n \cdot m$ workgroups, where each workgroup calls the barrier one thousand times inside a for loop. Note that the required concurrency of the barriers in this benchmark is only given, if the GPU can process $n \cdot m$ workgroups concurrently. The results of this microbenchmark performed on Volta can be seen in Figure 6.

This microbenchmark shows that, as expected, for a small number of workgroups per barrier Barrier-A is faster than Barrier-B. But Barrier-A also scales worse. Consequently, for about more than 300 workgroups per barrier, Barrier-B overtakes. Since on Volta this corresponds to a single tile with four workgroups per core, which is not well performing for multi-workgroup tiling because of the high synchronization overhead, Barrier-A is always better than Barrier-B on Volta. Although one might expect NVIDIA's Barrier-C to have a good performance, the experiments show that it is much slower than Barrier-A and Barrier-B. Also those measurements show that for Barrier-A, the performance is almost constant for up to

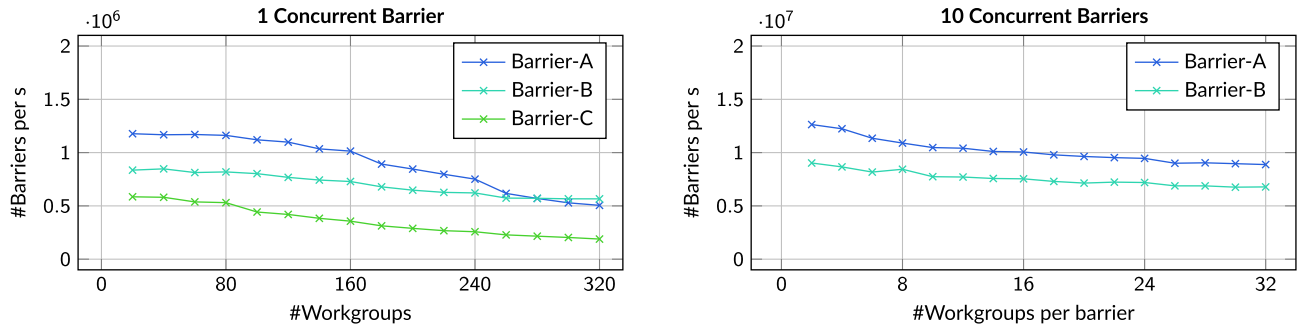

FIGURE 6 Comparison of the barrier types with one concurrent barrier (left) and 10 concurrent barriers (right) on Volta. Note the different scales of both axes between both plots

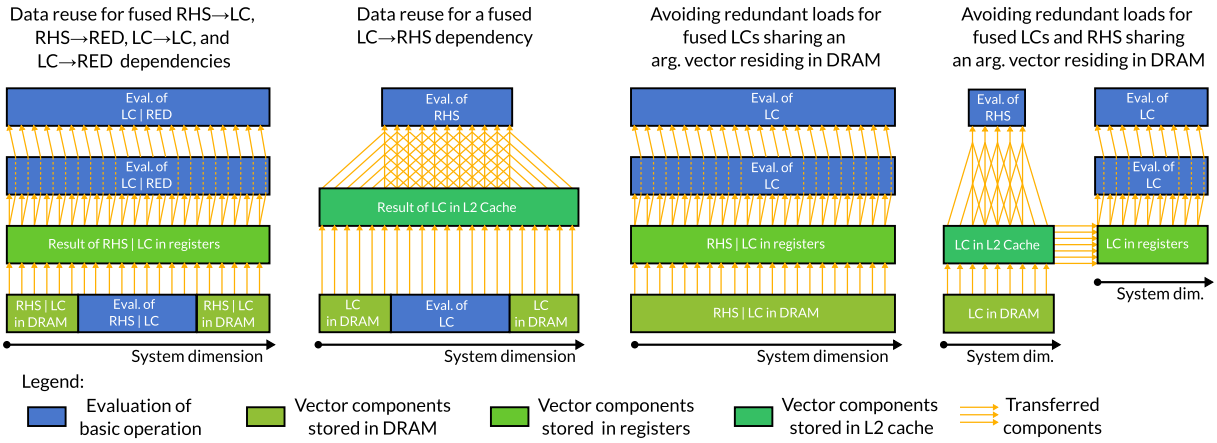

FIGURE 7 Data reuse in a multi-workgroup tile. Note that data reuse via registers requires the same mapping between workitems and vector components for all basic operations. However, data reuse via registers is not possible for the argument vector of an evaluation of the RHS, since the RHS may require random access on its argument vector within the limited access distance.

160 workgroups per barrier. This suggests that even for this amount of workgroups the sequential execution of the atomic increment of Barrier-A does not impact the performance that much compared with its latency. Also, by comparing the test series of a single concurrent barrier to the one with 10 concurrent barriers, one can see that throughput of barriers per second also roughly increases by a factor of 10. This shows that even for many concurrent barriers, the memory subsystem does not form a bottleneck. As a final comparison, previous experiments[11] have shown that on a GeForce 280 GTX from 2008, which lacks a L2-cache, Barrier-B is faster than Barrier-A if there are more than four workgroups per barrier.

4.4 | Memory optimizations in a multi-workgroup tile

Multi-workgroup tiling can exploit the on chip memories for data reuse by (see Figure 7):

- Using registers to transfer data along fused RHS → LC, RHS → RED, LC → LC and LC → RED dependencies: Because all basic operations in a kernel have the same mapping between vector components and workitems, for fused RHS → LC, RHS → RED, LC → LC, and LC → RED dependencies, a workitem computes all the components of the tail of the dependency which it requires for evaluating the head of the dependency. As a consequence the data can be transferred via the register file. Note that the tiling also causes a boundary case, where only the workitems in the middle of the tile actually evaluate the tail of the dependency, while the workitems at the edge of the tile load the components from DRAM into a registers.
- Using the L2-cache to transfer data along a fused LC → RHS dependency: In this case a workitem evaluating the RHS requires access to the components of the linear combination, which are within the limited access distance of its assigned components. However, those components of the linear combination are computed by other workgroups of the same tile. As a consequence, the data can only be transferred by a buffer residing in global memory, which is cached by the L2-cache. Note that the tiling again causes a boundary case, where the workitems at the edge of the tile require components of the linear combination computed by a previous kernel for evaluating the RHS.

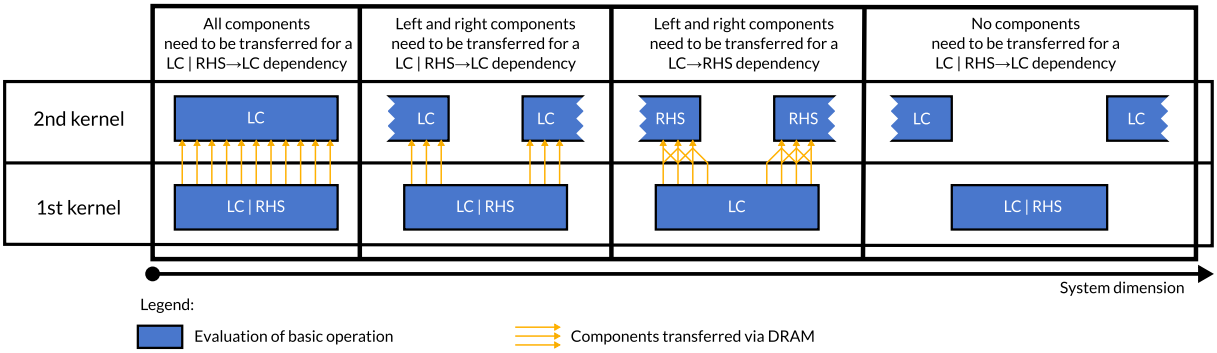

FIGURE 8 Data transfer via DRAM between two kernels, each processing its own tiles. The three cases that all of the components, the left and right components, and no components need to be transferred are exemplarily explained for several dependencies. Note that a LC → RHS dependency needs to transfer more components than RHS → LC or LC → LC dependencies because of the components within the limited access distance

- Using registers to avoid redundant loads when several linear combinations sharing an argument vector that resides in DRAM are fused: Again, because all basic operations in a kernel have the same mapping between workitems and vector components, a workitem can first load its assigned components of an argument vector, which is shared by several linear combinations and resides in DRAM, into registers and then use those registers as an argument to evaluate the depending linear combinations.
- Using the caches to reuse cached components of a vector, which resides in DRAM and is a shared argument of a fused RHS and one or several linear combinations: While in this case the limited access distance of the RHS does not allow data reuse via the register file, one can still exploit that the limited access distances causes all the workitems of a tile to access the argument vector in a local fashion. As a consequence, the components of the argument vector only need to be read from DRAM once.
- Writing back only those vector components to DRAM which are actually read by a succeeding kernel.

For a tile loading components of a basic operation into registers and for writing back those components to DRAM for a later kernel to read, our framework automatically detects three cases (see Figure 8):

- no components covered the tile need to be transferred,
- the upper and lower components covered by the tile need to be transferred, and
- all the components covered the tile need to be transferred.

For fusing more than two links of the dependency chain one has the choice between:

- single buffering, which requires two barriers for each fused link, but only one global memory buffer for data transfer, and
- double buffering, which requires one barrier for each fused link, but two buffers.

Consequently, single buffering has a higher overhead but a smaller working set than double buffering. Since the multi-workgroup barrier has a high cost, experiments have shown that double buffering is significantly better than single buffering.

In order to reduce the memory footprint of the buffering one can employ the following additional two optimizations:

- A designated buffer for fusing a LC → RHS dependency is not needed, if the tile writes back all the components of the respective linear combination to DRAM for a later kernel to read, since in this case this memory can be used, which reduces the working set.
- In the case that the lower and upper components of the linear combination are written back to DRAM for a later kernel to read, those components do not have to be written to the buffer.

Note that instead of using a buffer to store the resulting components of the linear combination, one can also write all the resulting components of the linear combination back to the corresponding vector residing in DRAM, even if those components are not needed for a later kernel. However, this not only increases the working set, but also the overall memory requirements of the ODE solver. On the plus side, the RHS now always reads from a consecutive memory range, which simplifies the RHS and consequently reduces the overhead.

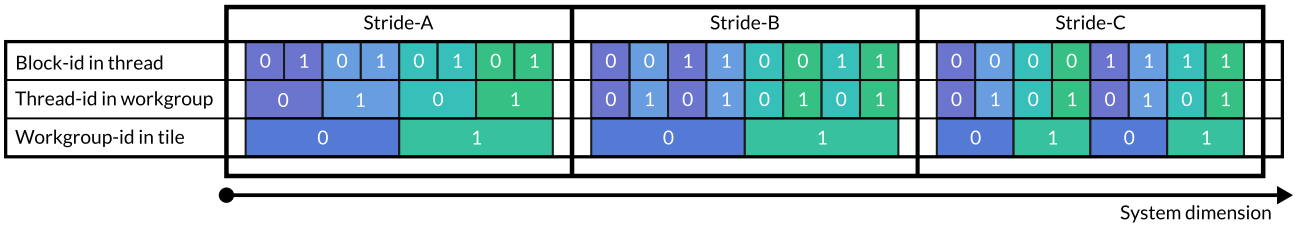

FIGURE 9 The implemented three strides between the blocks processed by a thread, with *Stride-A* spanning the block of the thread, *Stride-B* spanning the threads of the workgroup, and *Stride-C* spanning the workgroups of the tile

4.5 | Mapping workitems to vector components

Since GPUs typically support 4, 8, and 16 byte gather instructions, a thread should process blocks of 1, 2, or 4 times the SIMD width consecutive components in case of single precision and blocks of 1 or 2 times the SIMD width consecutive components in case of double precision to achieve a good coalescing for a wide range of problems, including stencils.

Moreover, it may be beneficial for a thread to evaluate multiple of those blocks with a given stride for optimizing the register allocation, load balance, and locality and for allowing arbitrarily large tile widths. There are several options for choosing the strides between those blocks, where larger strides result in less locality but in a better load balance. The following three strides as shown in Figure 9 were implemented:

- *Stride-A* spanning the block of the thread (best locality, worst load balance),
- *Stride-B* spanning the threads of the workgroup (mediocre locality, mediocre load balance), and
- *Stride-C* spanning the workgroups of the tile (worst locality, best load balance).

4.6 | Tile width and height considerations

For choosing the tile width and height there are several restrictions depending on the tiling scheme:

- *Hexagonal tiling:* All hexagons in the dependency chain must have the same height. Moreover, all the hexagons in the odd columns must have the same width and all hexagons in the even columns must have the same width.
- *Trapezoidal tiling:* All trapezoids in a row must have the same height, while trapezoids in different rows may have different heights. Also, in a row the odd trapezoids must have the same width and the even trapezoids must have the same widths, while trapezoids from different rows may have different widths.

While the tile height is an independent parameter, the tile width is determined by several interacting parameters:

- n_{comp}: the number of components per block
- n_{blocks}: the number of blocks per thread
- n_{threads}: the number of threads per workgroup
- n_{wgtile}: the number of workgroups per tile
- n_{wgcore}: the number of number of concurrent workgroups per core
- n_{tiles}: the number of concurrent tiles on the GPU

We call this set of parameters a *tile width configuration*, which results in a tile width w of

$$w = n_{\text{comp}} \cdot n_{\text{blocks}} \cdot n_{\text{threads}} \cdot n_{\text{wgtile}}. \tag{7}$$

In order to achieve a good performance, all parameters of the tile width configuration need to be chosen in a suitable way. First, one should choose the number of blocks per thread, for which a small value of 1, 2, 3, or 4 typically yields the best performance. Next, one has to choose

the number of threads per workgroup and the number of concurrent workgroups per core (n_{wgcore}) so that there is only a small number of workgroups per core (to reduce the barrier overhead), that there is not much register spilling, and that a core can hold the maximum number of threads for the given register allocation of the kernel. Depending on the occupancy rules of the GPU, there are typically only a few promising values for both.

Finally, one has to choose the number of workgroups per tile, which indirectly determines the number of concurrent tiles on the GPU (n_{tiles}). For maximizing resource usage, the following equations should hold true:

$$\left\lfloor \frac{n_{\text{wgcore}} \cdot n_{\text{cores}}}{n_{\text{wgtile}}} \right\rfloor = n_{\text{tiles}} \quad \text{and} \quad \left\lfloor \frac{n_{\text{wgcore}} \cdot n_{\text{cores}}}{n_{\text{tiles}}} \right\rfloor = n_{\text{wgtile}}, \tag{8}$$

where n_{cores} is the number of cores on the GPU.

5 | EXPERIMENTAL EVALUATION

5.1 | Setup and experiments

We have added the ability to generate multi-workgroup tilings for explicit one-step methods along a user defined dependency chain to our automatic prototype framework.[2,3] This framework allows a user to solve an arbitrary IVP by an arbitrary explicit ODE method of several supported classes (RK methods, PIRK methods, peer methods, Adams–Bashforth methods). This generality is achieved by an internal data flow representation of the ODE method from which, after several transformations, CUDA or OpenCL kernels with all memory optimizations from Section 4.4 can be generated and executed automatically with automatic memory management. To automatically tune performance, the framework performs an exhaustive online search over a predefined set of values to optimize several performance parameters, for example, tile heights and tile width configurations. Compared with our previous experiments,[4] we have further improved our framework. Mainly, we focused on an improved autotuning functionality, which now allows each kernel to have its own tile width configuration, and hence different tile widths for different rows of the trapezoidal tiling, and also different tile widths for the odd and even columns in the hexagonal tiling. Overall, those optimizations yielded improved results for both tiling schemes and most access distances.

Since double precision (DP) is usually desirable for scientific simulations, we consider two NVIDIA GPU architectures that provide reasonable DP performance: Volta (Titan V, 80 cores, year: 2017) and Kepler (GeForce GTX Titan Black, 15 cores, year: 2014). For these GPUs, we let our framework generate CUDA code and perform all measurements in double precision. Note that there are further graphics cards equipped with the same Kepler or Volta GPUs, but a vastly different SP/DP performance and DRAM bandwidth, for example, the Volta Quadro GV 100 or the Kepler Quadro K6000.

As a third test GPU, we use a AMD Polaris (Radeon RX 480, 36 cores, year: 2016) with the OpenCL implementation of AMD. Since this card only has a low double precision performance, we use single precision for our measurements on this card. Note that we encountered several difficulties on this GPU:

- The register usage of a kernel cannot be queried using OpenCL, which forces our autotuning to check more candidates.
- It may take several 100,000 cycles for an atomic increment on the counter of a barrier to become visible in the L1-cache of the cores. However, the problem can be fixed by modifying the barrier so that it does not perform its busy waiting on its counter by a regular load, but by an atomic increment with 0 as argument, which bypasses the L1-cache.
- Some kernels with multi-workgroup tiling cause an error or deadlock if they compute more than two components per workitem for unknown reasons. Since current autotuning functionality of our framework cannot recover from those errors, we have to limit all kernels on Polaris to two components per workitem.
- While in CUDA one can specify a maximum workgroup size and a minimum number of workgroups per core for this size, in OpenCL one can only enforce a kernel to run with a given workgroup size, which is less than or equal to the maximum workgroup size of the device. However, one cannot enforce a minimum number of concurrent workgroups per core. As a consequence, and since it has to be guaranteed that the GPU executes at least the workgroups of a single tile concurrently, a tile may not consist of more workgroups than the GPU has cores. Unfortunately, Polaris supports up to 2560 workitems per core, but only a maximum workgroup size of 1024. This, and the problem of the kernels causing an error or deadlocking with more than one block per thread limits the maximum tile width on this GPU.

As ODE methods we consider two embedded explicit RK methods of different order: Bogacki–Shampine 2(3) with four stages and Verner 5(6) with eight stages. The IVP we consider is BRUSS2D, a chemical reaction diffusion system of two chemical substances, discretized on a 2D

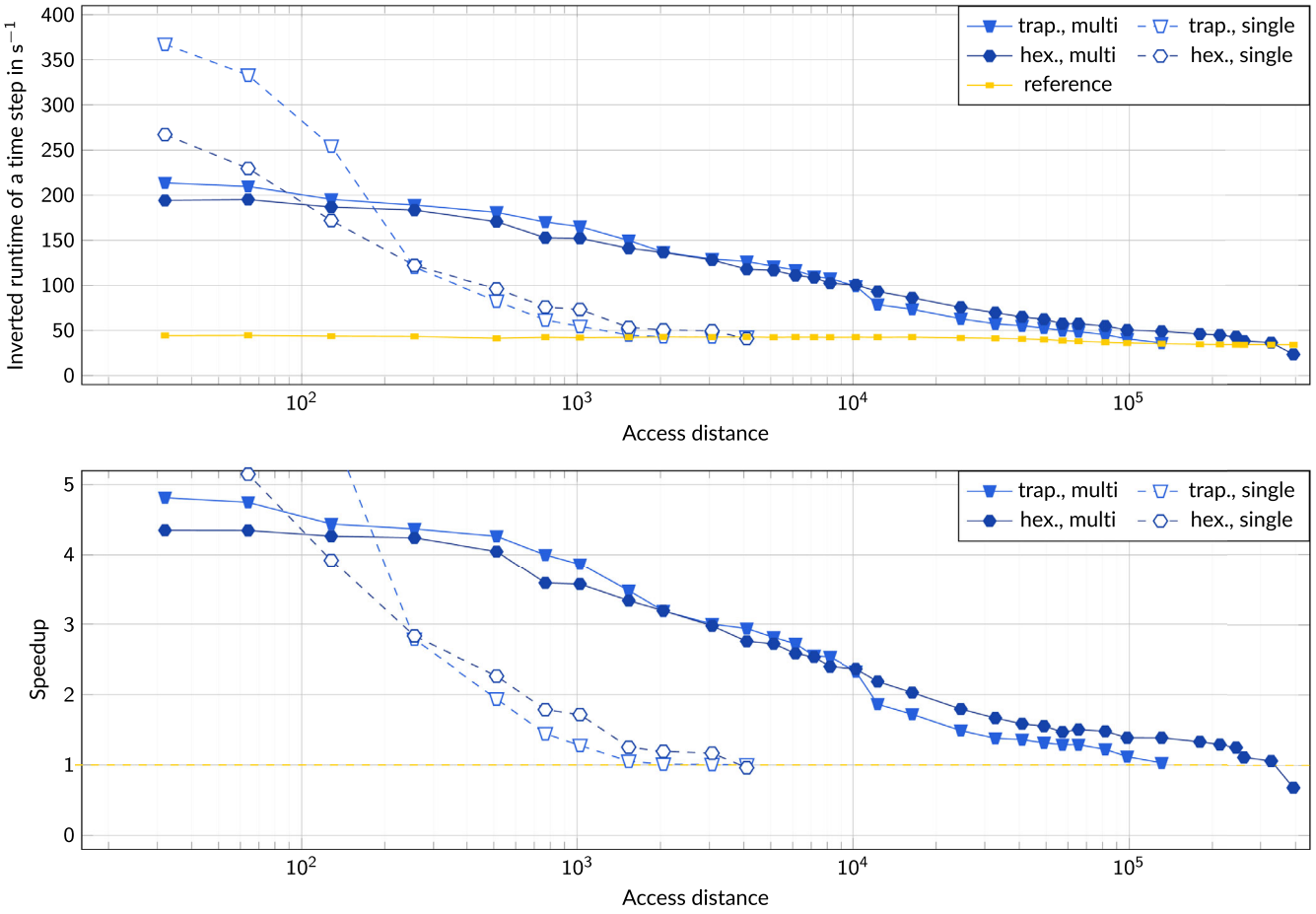

FIGURE 10 Impact of the access distance on the inverted runtime of a timestep and the speedup compared with the reference implementation for the two tiling schemes using the optimal tile width configuration and tile height for Verner's method on Volta

grid by a 5-point diffusion stencil. Hence, BRUSS2D has a limited access distance d of two times its x-dimension. For simulating the impact of the access distance, we set the x-dimension to $d/2$ and adjust the y-dimension so that the overall size of the ODE system n remains almost constant ($n = 16 \cdot 1024 \cdot 1024$ on Volta and Kepler and $n = 32 \cdot 1024 \cdot 1024$ on Polaris). Note that due to the problem statement, we only exploit the limited access distance of BRUSS2D resulting in a 2D iteration space (system dimension and time). However, we do not exploit the 2D spatial structure of the stencil, which would result in a 3D iteration space (x-, y-, and time dimension). As reference for the speedup computation we use a nontiled implementation which fuses all but the LC → RHS dependencies.

With this setup the following experiments have been performed:

- Figures 10-13: Impact of the access distance on the runtime (displayed as the inverted runtime of a time step) and the speedup for the two tiling schemes using the optimal tile width configuration and tile height for Verner's method on Volta, for the Bogacki–Shampine method on Volta, for Verner's method on Kepler, and for Verner's method on Polaris.
- Figure 14: Impact of the access distance on the speedup of the tiling schemes for different tile heights with the optimal tile width configuration using Verner's method on Volta.
- Figure 15: Best tile width and best number of concurrent tiles as a function of the access distance for several fixed tile heights using Verner's method with hexagonal tiling on Volta.
- Figure 16: Impact of the blocks per thread and the stride on the performance for Verner's method and the hexagonal tiling with a tile height of 4 on Volta.
- Figure 17: Profiling results of the utilization of double precision units, the utilization of instruction issue slots, the utilization of DRAM bandwidth and of the DRAM volume for Verner's method on Volta and Kepler.

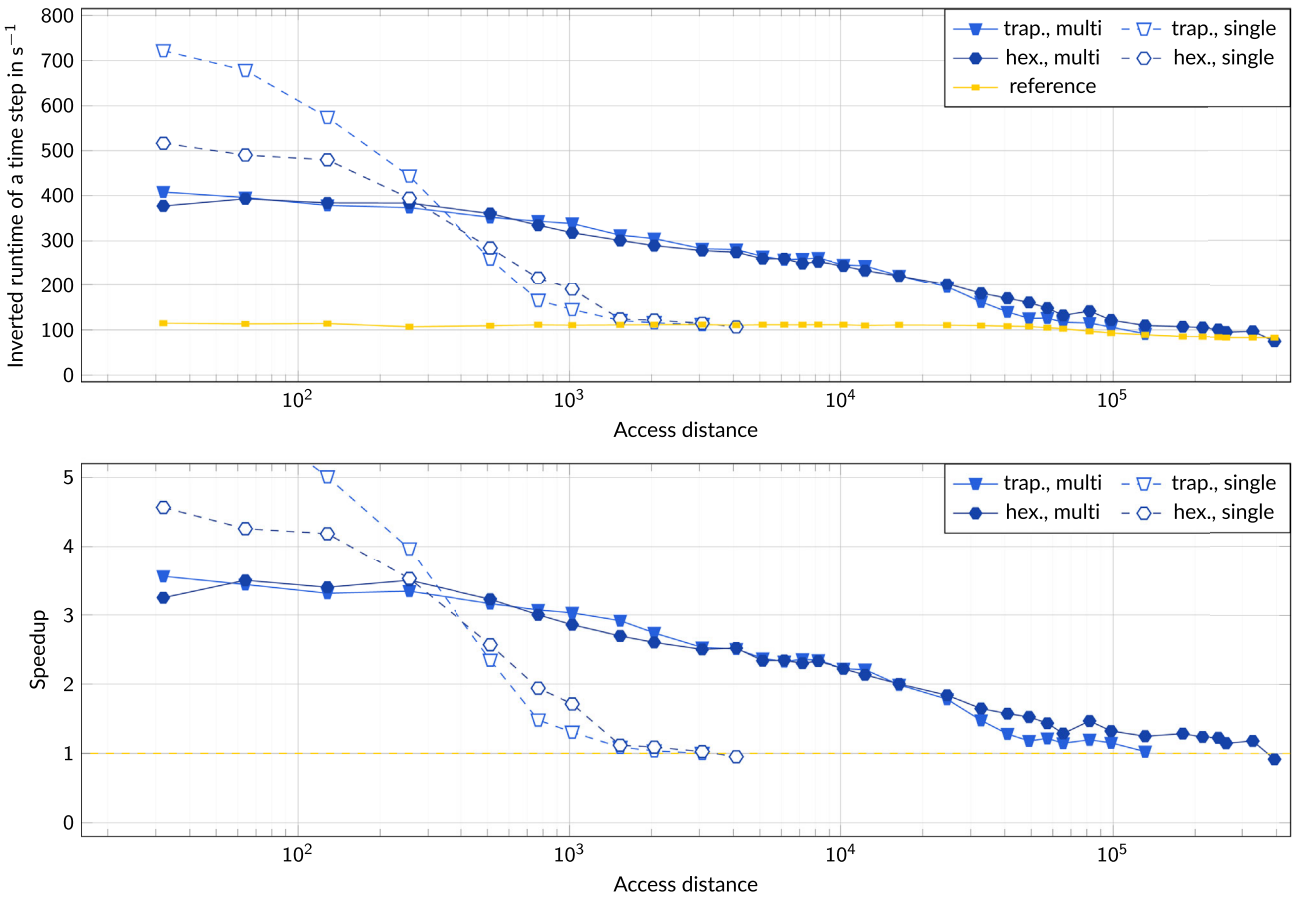

FIGURE 11 Impact of the access distance on the inverted runtime of a timestep and the resulting speedup compared with the reference implementation for the two tiling schemes using the optimal tile width configuration and tile height for the Bogacki–Shampine method on Volta

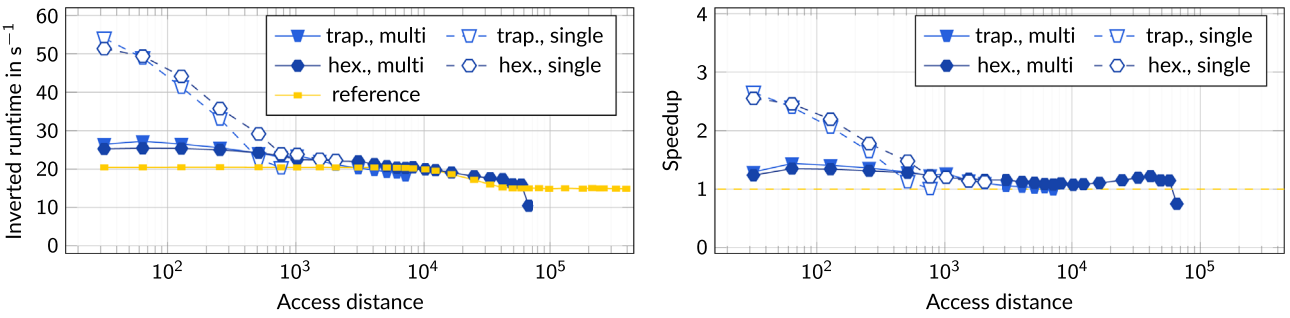

FIGURE 12 Impact of the access distance on the inverted runtime of a timestep and the speedup compared with the reference implementation for the two tiling schemes using the optimal tile width configuration and tile height for Verner's method on Kepler

5.2 | Discussion

5.2.1 | For which access distances is the multi-workgroup tiling worthwhile? (see Figures 10, 11, 12, 13)

On Volta, there is a wide range of access distances where multi-workgroup tiling yields a significant speedup over the reference implementation and single-workgroup tiling. For example, for Verner's method, multi-workgroup tiling becomes faster than single-workgroup tiling for access distances larger than $2.6 \cdot 10^2$, yields a speedup of 2.4 over the reference implementation for an access distance of $1.0 \cdot 10^4$, a speedup of 1.3 for an access distance of $2.5 \cdot 10^5$, and a speedup smaller than one only for access distances larger than $3.3 \cdot 10^5$. This wide range of access distances, for which multi-workgroup tiling yields a speedup over the reference implementation, is a remarkable improvement over single-workgroup tiling, which only

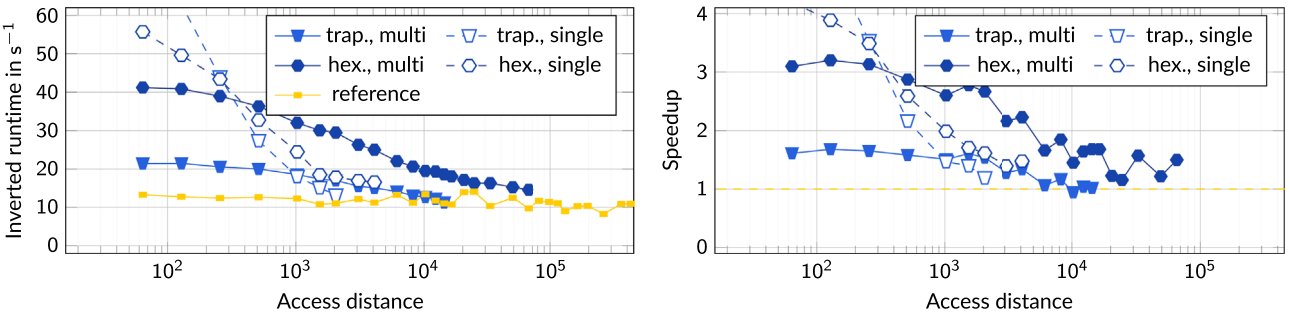

FIGURE 13 Impact of the access distance on the inverted runtime of a timestep and the resulting speedup compared with the reference implementation for the two tiling schemes using the optimal tile width configuration and tile height for Verner's method on Polaris

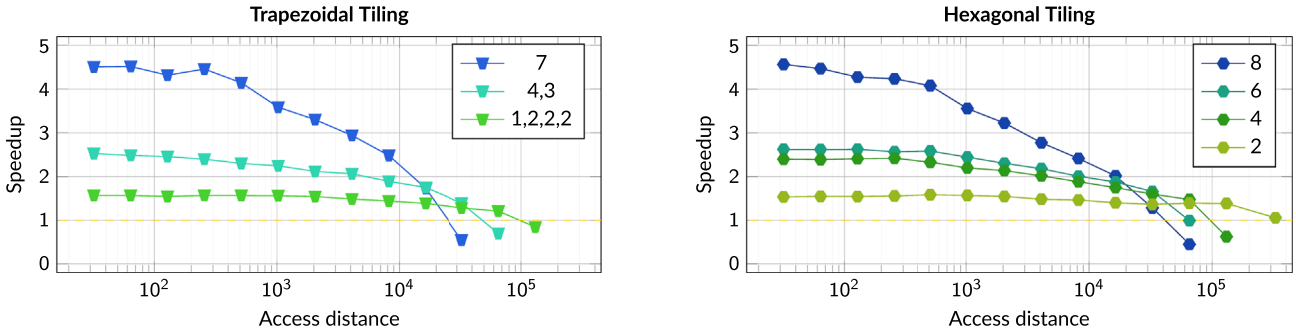

FIGURE 14 Impact of the access distance on the speedup of the tiling strategies for different tile heights with the optimal tile width config using Verner's method on Volta. For the trapezoidal tiling, the legend entries denote a sequence of individual tile heights along the dependency chain

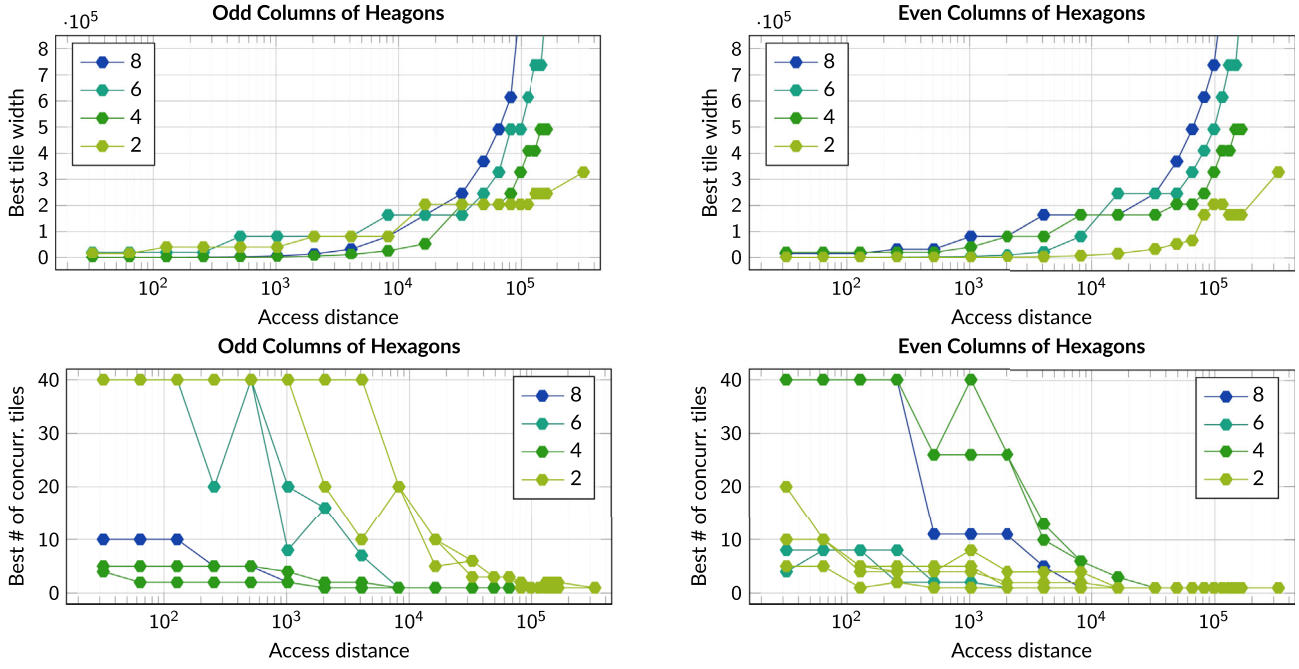

FIGURE 15 Impact of the access distance on the best tile width (top) and the best number of concurrent tiles (bottom) for several fixed tile heights using Verner's method with hexagonal tiling on Volta. Note that the tiling starts with the odd hexagons. As for the best number of concurrent tiles, each line of the same color denotes the best amount for one kernel of the respective height

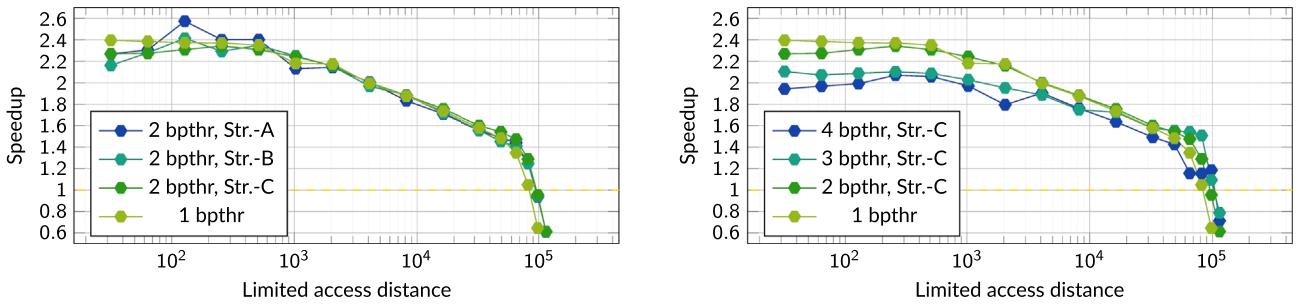

FIGURE 16 Impact of the blocks per thread and the stride on the performance for Verner's method and the hexagonal tiling with a tile height of 4 on Volta

yields a speedup for access distances smaller than $1.5 \cdot 10^3$. Unfortunately, the experiments also show that multi-workgroup tiling does not work well on Kepler, and only small speedups of about 1.2 are observed for access distances of up to $5 \cdot 10^4$. The speedup of multi-workgroup tiling on Polaris is in between of Kepler and Volta, for example, a speedup of 2.6 for an access distance of $1.0 \cdot 10^3$, and still a speedup of 1.5 for an access distance of $1.0 \cdot 10^4$. Note that, if there were not the previously mentioned problems with OpenCL, multi-workgroup tiling would probably have a better performance for medium and large access distances on Polaris.

5.2.2 | How does the number of stages of the method impact the performance? (see Figures 10 and 11)

The more stages a method has, the larger the speedup over the reference implementation for small access distances is. This may be explained by methods with more stages having a lower arithmetic intensity. For example, a limited access distance of 32 results in a speedup of 4.8 for Verner's method and a speedup of 3.6 for the Bogacki–Shampine method. Moreover, for larger access distances one might expect the tiling to become less efficient the more stages a method has, since those methods also have a higher working set. Yet, the measurements show that for both methods investigated the tiling offers a similar speedup if the access distances are large.

5.2.3 | Which tiling scheme is better? (see Figures 10, 11, 12, 13)

On Volta the trapezoidal tiling has a slightly better performance than the hexagonal tiling for small access distances, while for larger access distances (e.g., $d \geq 1.1 \cdot 10^4$ for Verner's method or $d \geq 1.5 \cdot 10^4$ for the Bogacki–Shampine method), the hexagonal tiling is better. On Kepler, for Verner's method and access distances, where multi-workgroup tiling is faster than single-workgroup tiling, the hexagonal tiling scheme is better. On Polaris, the hexagonal scheme is far better than the trapezoidal scheme.

5.2.4 | How does the tile height influence the performance depending on the access distance? (see Figure 14)

For smaller access distances, larger tile heights are more efficient because they reduce the DRAM data transfers further. However, larger tile heights also have a larger working set, and hence scale worse when increasing the access distance. By contrast, smaller tile heights yield smaller speedups for smaller access distances, but also degrade far less for larger access distances.

5.2.5 | How does the tile width and the number of concurrent tiles impact the performance depending on the tile height and the access distance? (see Figure 15)

For all tile heights, the best number of concurrent tiles decreases with the access distance while the best tile width increases. This may be explained by many concurrent and small tiles having a lower synchronization overhead and a less synchronized instruction mix, but also reducing the DRAM transfers less than larger fewer tiles, and larger access distances having a tighter memory bottleneck. Also, the larger the tile height, the larger the optimal tile width and the smaller the optimal number of concurrent tiles. This may be explained that for a fixed tile width larger tile heights result in smaller widths for the outer rows of the tile, which makes tiles with a small width and a large height less efficient. The measurements also show that depending on the height of the hexagons, the autotuning results in either a small width for the odd column of hexagons and a large width for

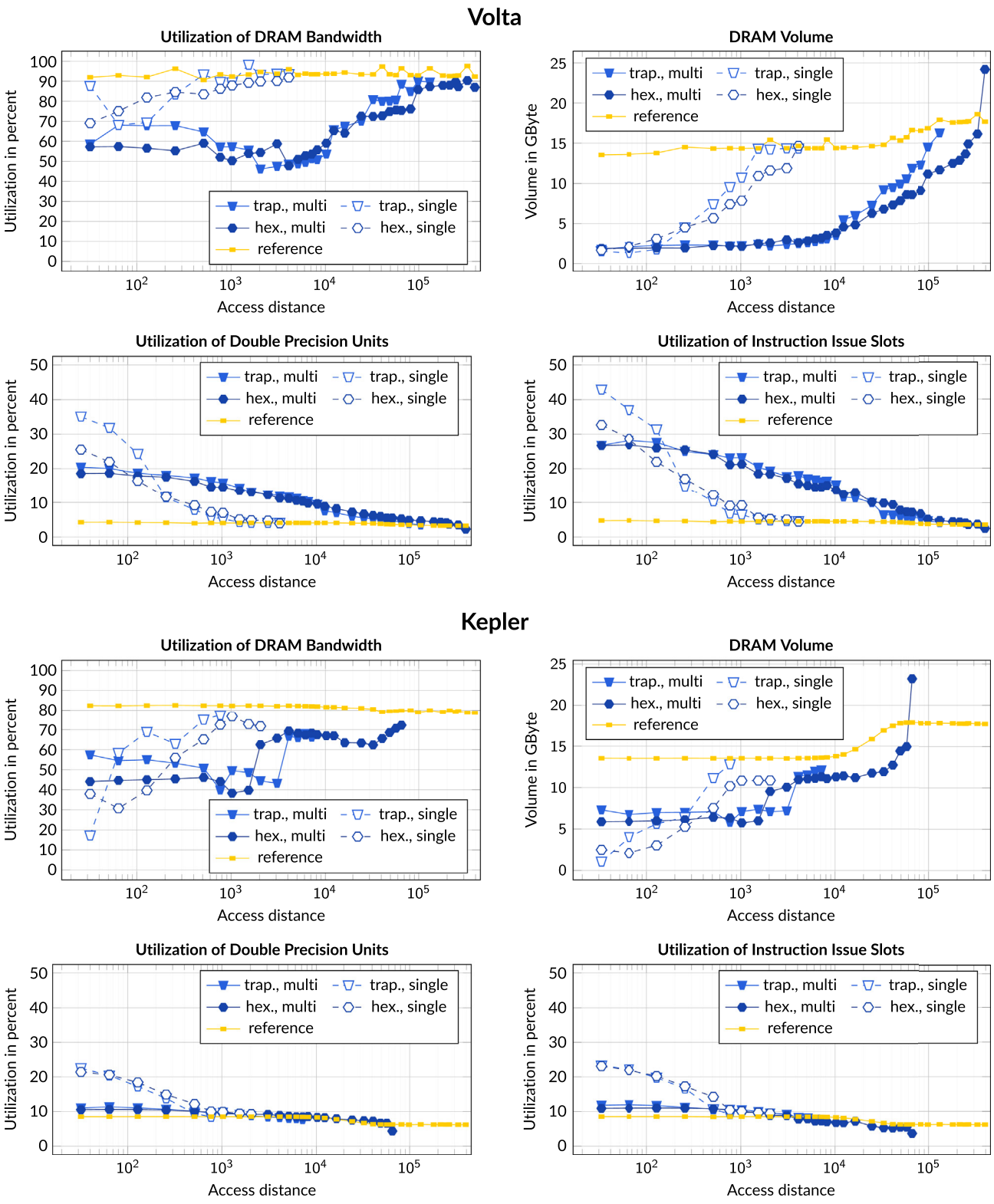

FIGURE 17 Profiling of the utilization of double precision units, the utilization of instruction issue slots, the utilization of DRAM bandwidth, and of the DRAM volume for Verner's method on Kepler

the even column of hexagons or vice versa. This may be explained by, depending on the height, either the odd column or the even column reduces the dram volume more efficiently. A similar effect is observed for the trapezoidal tiling, where the autotuning results in the odd trapezoids in a row having a small width, while the even hexagons have a large width. Furthermore, the graphs for the best number of concurrent tiles show several spikes, which may be explained by many vastly different tile width configurations yielding very similar results for small to medium access distances. As a consequence, while the best tile width configuration may be vastly different for each access distance, the same tile width configuration will yield similar results for similar access distances.

5.2.6 | How many blocks per thread and which strides yield the best performance? (see Figure 16)

For limited access distances of up to $3.3 \cdot 10^4$ one block per thread and all three strides with two blocks per thread yield almost the same performance. Note that in the case of one block per thread the strides introduced beforehand are not applicable. For limited access distances larger than $3.3 \cdot 10^4$ two, three, or four blocks per workitem are better than one block. Also for those limited access distances, for which more than one block is better, Stride-C, which prefers load balancing over locality, typically yields the best performance.

5.2.7 | How do the implementation variants utilize the resources of the GPU? (see Figure 17)

Profiling reveals that the reference variant is hard memory bound with about 0.53 FLOPs per byte. As for multi-workgroup tiling and small limited access distances, the DRAM volume is efficiently reduced compared with the reference variant. However, because of the barrier overhead and the data transfer by the L2-cache along a LC → RHS dependency being less efficient, multi-workgroup tiling can only utilize the DP units by 20%, the instruction issue slots by 27%, and the DRAM bandwidth by 70%, which both limits the achieved speedup for small access distances compared with single-workgroup tiling. For large access distances, the multi-workgroup tiling cannot reduce the DRAM volume efficiently anymore, and, as a consequence, the utilization of DRAM bandwidth increases and multi-workgroup tiling clearly becomes memory bound.

5.2.8 | Why is multi-core tiling on Kepler worse than on Volta, and the performance on Polaris being in between the two? (see Figure 17)

First, Volta has a 20480 KiB register file and a 3072 KiB L2-cache, while Polaris has a 9216 kiB register file and a 1024 KiB L2-cache, and Kepler has a 3840 KiB register file and a 1536 KiB L2-cache. These smaller on-chip memories on Kepler and Polaris limit data reuse and hence make multi-workgroup tiling less efficient. Because of this, for small access distances, multi-workgroup tiling on Kepler only reduces the DRAM volume of the reference implementation from 14 to about 6 GB, while on Volta multi-workgroup tiling reduces the DRAM volume of the reference implementation from 14 to 2 GB. Second, Volta and Polaris have a tighter memory bottleneck than Kepler (Volta: 11.4 DP FLOPs per byte, Polaris: 20 SP FLOPs per byte, and Kepler: 5.1 DP FLOPs per byte), which makes increasing data reuse much more important. This is supported by the profiling results, which show that the reference implementation utilizes the double precision units about 5% on Volta and 9% on Kepler. Both factors not only make single-workgroup tiling but also multi-workgroup tiling on Volta much more important than on Kepler. Since this trend is expected to continue on future GPU generations, multi-workgroup tiling is expected to yield even better results in the future.

6 | CONCLUSION

We have seen that on modern GPU architectures such as NVIDIA Volta multi-workgroup tiling based on the generic notion of the access distance of an ODE system applied to the stages of an explicit one-step method could improve the performance for access distances of up to $\approx 2.5 \cdot 10^5$. This is a tremendous improvement over single-workgroup tiling, which only yielded a speedup for access distances of up to $\approx 1.5 \cdot 10^3$. Hence, by spanning tiles over multiple workgroups a much wider range of applications can benefit from tiling across the stages, in particular stencils on 2D grids. Still, for 3D grids, more specialized approaches exploiting the three-dimensional structure of the problem would be required. Also, since the memory bottleneck becomes tighter and the amount of on-chip memory increases every new GPU generation, on future GPU generations multi-workgroup tiling is expected to yield even higher speedups and to scale for even larger access distances. Hence, it is also expected that in the future even the large access distances arising from 3D stencils may be exploited by multi-workgroup tiling. As future work, we intend to consider tiling over several adjacent time steps for multi-workgroup tiling, to improve the autotuning functionality of our framework, to extend our framework to CPUs and GPU clusters, and to allow the user to describe the data flow graph of the method and the problem by domain specific languages.

ACKNOWLEDGMENT

This work has been supported by the German Research Foundation (DFG) under grant KO 2252/3-2. Open access funding enabled and organized by Projekt DEAL.

ORCID

Tim Werner https://orcid.org/0000-0002-3450-8019

REFERENCES

1. Hairer E, Nørsett SP, Wanner G. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd ed. Berlin, Germany: Springer; 2000.
2. Korch M, Werner T. Accelerating explicit ODE methods by kernel fusion. *Concurr Comput Pract Exp*. 2018;30(18):e4470. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4470.
3. Korch M, Werner T. Exploiting limited access distance for kernel fusion across the stages of explicit one-step methods on GPUs. Paper presented at: Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France; 2018:148–157.
4. Korch M, Werner T. Multi-workgroup tiling to improve the locality of explicit one-step methods for ODE systems with limited access distance on GPUs. Paper presented at: Proceedings of the 13th International Conference on Parallel Processing and Applied Mathematics (PPAM), Białystok, Poland; 2019.
5. Korch M, Werner T. Improving locality of explicit one-step methods on GPUs by tiling across stages and time steps. *Future Generat Comput Syst*. 2020;102:889-901. https://doi.org/10.1016/j.future.2019.07.075.
6. Hennessy J. L., Patterson D. A.. *Computer Architecture: A Quantitative Approach*. Amsterdam, Netherlands: Morgan Kaufmann; 5th. 2011.
7. Wang G, Lin YS, Yi W. Kernel fusion: an effective method for better power efficiency on multithreaded GPU. Paper presented at: Proceedings of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom), IEEE/ACM International Conference on Cyber, Physical and Social Computing (CPSCom), Hangzhou, China; 2010:344-350.
8. Wahib M, Maruyama N. Automated GPU kernel transformations in large-scale production stencil applications. Paper presented at: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Portland Oregon USA; 2015:259-270.
9. Filipovič J, Madzin M, Fousek J, Matyska L. Optimizing CUDA code by kernel fusion: application on BLAS. *J Supercomput*. 2015;71(10):3934-3957. https://doi.org/10.1007/s11227-015-1483-z.
10. Xiao S, Aji AM, Feng W. On the robust mapping of dynamic programming onto a graphics processing unit. Paper presented at: Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China; 2009:26–33.
11. Xiao S, Feng W. Inter-block GPU communication via fast barrier synchronization. Paper presented at: Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS), Atlanta, Georgia, USA; 2010:1-12.
12. Grosser T, Cohen A, Kelly PHJ, Ramanujam J, Sadayappan P, Verdoolaege S. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. Paper presented at: Proceedings of the 6th Workshop on General Purpose Processing Using GPUs (GPGPU-6), Houston, Texas, USA; 2013:24-31.
13. Grosser T, Cohen A, Holewinski J, Sadayappan P, Verdoolaege S. Hybrid hexagonal/classical tiling for GPUs. Paper presented at: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Orlando, Florida, USA; 2014:66-75.
14. Malas T, Hager G, Ltaief H, Stengel H, Wellein G, Keyes D. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM J Sci Comput*. 2015;37(4):C439-C464. https://doi.org/10.1137/140991133.
15. Christen M, Schenk O, Burkhart H. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. Paper presented at: Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium; 2011:676-687.
16. Tang Y, Chowdhury RA, Kuszmaul BC., Luk CK, Leiserson CE. The Pochoir stencil compiler. Paper presented at: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11), San Jose, California, USA; 2011:117–128.
17. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F. Amarasinghe S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Paper presented at: Proceedings of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13), Seattle, Washington, USA; 2013:519-530.
18. Advanced Stencil-Code Engineering (ExaStencils); 2020http://www.exastencils.org/. Accessed 25th May 2020.
19. Levchenko V, Perepelkina A. Locally recursive non-locally asynchronous algorithms for stencil computation. *Lobachevskii J Math*. 2018;39:552-561. https://doi.org/10.1134/S1995080218040108.
20. Levchenko V, Perepelkina A, Zakirov A, Goryachev I, Savchenko V. Numerical 3D simulation of the light propagation in the optical WGM-microresonator by the FDTD method. Paper presented at: Proceedings of the 2017 IEEE International Conference on Computational Electromagnetics (ICCEM), Kumamoto, Japan; 2017:291-292.
21. Korneev B, Levchenko V. Runge-Kutta discontinuous galerkin method and diamondtorre GPGPU algorithm for effective simulation of large 3D multiphase fluid flows with shocks. Paper presented at: Proceedings of the 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), Nowosibirsk, Russia; 2019:0817-0822.
22. Korch M, Rauber T, Stachowski M, Werner T. Influence of locality on the scalability of method- and system-parallel explicit peer methods. Paper presented at: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS). Annals of Computer Science and Information Systems, PTI/IEEE, Gdansk, Poland; 2016:685-694.
23. Korch M. Locality improvement of data-parallel Adams–Bashforth methods through block-based pipelining of time steps. In Proceedings, since Euro-Par 2012 was a conference held in Rhodes Island, Greece; 2012:563-574.

How to cite this article: Korch M, Werner T. An in-depth introduction of multi-workgroup tiling for improving the locality of explicit one-step methods for ODE systems with limited access distance on GPUs. *Concurrency Computat Pract Exper*. 2020;e6016. https://doi.org/10.1002/cpe.6016