Efficient GPU Offloading with OpenMP for a Hyperbolic Finite Volume Solver on Dynamically Adaptive Meshes

Mario Wille[1(✉)], Tobias Weinzierl[2], Gonzalo Brito Gadeschi[3], and Michael Bader[1(✉)]

[1] TUM School of Computation, Information and Technology, Technical University of Munich, Garching, Germany
`{mario.wille,michael.bader}@tum.de`

[2] Department of Computer Science, Institute for Data Science—Large-scale Computing, Durham University, Durham, UK
`tobias.weinzierl@durham.ac.uk`

[3] NVIDIA, Munich, Germany
`gonzalob@nvidia.com`

Abstract. We identify and show how to overcome an OpenMP bottleneck in the administration of GPU memory. It arises for a wave equation solver on dynamically adaptive block-structured Cartesian meshes, which keeps all CPU threads busy and allows all of them to offload sets of patches to the GPU. Our studies show that multithreaded, concurrent, non-deterministic access to the GPU leads to performance breakdowns, since the GPU memory bookkeeping as offered through OpenMP's `map` clause, i.e., the allocation and freeing, becomes another runtime challenge besides expensive data transfer and actual computation. We, therefore, propose to retain the memory management responsibility on the host: A caching mechanism acquires memory on the accelerator for all CPU threads, keeps hold of this memory and hands it out to the offloading threads upon demand. We show that this user-managed, CPU-based memory administration helps us to overcome the GPU memory bookkeeping bottleneck and speeds up the time-to-solution of Finite Volume kernels by more than an order of magnitude.

Keywords: GPU offloading · Multithreading · OpenMP · Dynamically adaptive mesh refinement

This research has been supported by EPSRC's ExCALIBUR programme (projects EX20-9, PAX–HPC and MGHyPE), by the German Ministry of Education and Research (BMBF, project targetDART) and by Intel's Academic Centre of Excellence at Durham University. Supercomputing resources and support was provided by the ARCHER2 UK National Supercomputing Service, the Erlangen National High Performance Computing Center, Jülich Supercomputing Center and CINECA. See the Acknowledgements section for details.

1 Introduction

GPUs are the workhorses of exascale: In exascale machines, the biggest performance share is offered by multiple GPUs per node, being orchestrated by modestly powerful CPUs. However, exascale simulation software also needs to run efficiently on smaller machines, where more CPU-centric nodes are accompanied by few GPUs as accelerators for specific tasks. It is also not yet clear if future top-end systems will continue to be GPU-centric.

Various orthogonal concepts to operate GPUs exist: (i) GPUs can be the sole data owners [19], they can own parts of the data and be compute cores on equal footing with the host cores [13,14], or they can be treated as offloading devices providing services to the host for compute-heavy phases. (ii) GPUs can be interpreted as a few devices with large internal concurrency, or they can be read as compute units accepting many tasks concurrently. In this case, we either split up the GPU internally or toggle between tasks. (iii) Finally, GPUs can be associated statically to particular cores, or we can allow multiple cores to share them. Simulation software is expected to navigate flexibly within these dimensions, as it is not yet clear which paradigm will dominate the future. This challenges the software's design and parallelisation concept.

We study ExaHyPE [15], a wave equation solver that employs explicit time stepping and dynamically adaptive mesh refinement (AMR) with Peano [20]. It uses patches of cells constructed through octree-type AMR (cf. [6] and references therein), and implements a Finite Volume scheme over these patches[1].

ExaHyPE and Peano follow the MPI+X paradigm, i.e., we have many ranks, each hosting several threads. Each thread can take Finite Volume patches and offload them to the GPU. We work in a coupled multiscale and multiphysics environment. As the physics, control, and algorithmic logic reside on the CPU cores which in turn potentially needs access to all simulation data, the GPU is used in offloading mode for the "number crunching". This comes at the cost of additional data movement between GPU and CPU [19].

To construct GPU compute kernels with a high computational load, ExaHyPE provides the opportunity for the host to gather multiple of our patches into a batch of patch update tasks which are handled in one rush on the accelerator via one kernel call [22]. This way, one offload action potentially can occupy the whole GPU. Yet, as we work with dynamically adaptive meshes, we do not constrain at any point which core can access a GPU, and multiple cores potentially may hit the accelerator simultaneously.

ExaHyPE's reference implementation realises the offloading per thread via OpenMP's `target` constructs (cf. [10,18]). Our measurements suggest that the major OpenMP implementations avoid race conditions on the GPU by locking the GPU per data transfer: Whenever the runtime encounters a `target map` clause or implicit data offloading, the target GPU is halted. This is a reasonable design pattern to realise remote memory access in any system. Unfortunately, it

[1] Peano and ExaHyPE are available under a modified BSD license at https://gitlab.lrz.de/hpcsoftware/Peano.

introduces significant overhead and synchronisation. Developers face a triad of challenges: To design compute kernels of sufficient load, to overlap computation and data transfer, and to avoid that data is allocated or freed while computations run. Our work presents data for the NVIDIA ecosystem, but we have observed qualitatively the same challenging behaviour for core LLVM and AMD's offloading. Similar results have been reported for pure CUDA [14].

We study two approaches tackling the latter two challenges: The first approach reserves memory on the GPU upon demand, yet does not free it anymore. Instead, it hands out pre-reserved memory to threads whenever they decide to offload. As the memory ownership resides on the host, most synchronisation and coordination can be handled there. The GPU is only interrupted whenever we have to grow the pre-allocated memory. Approach number two relies on virtual shared memory between the GPU and the host. Pre-allocated shared memory regions are held on the host. Logical data transfers to the GPU become plain memory copies on the CPU into the pre-allocated shared memory regions, while the actual data transfer then is realised via the GPU's page fault mechanisms. The allocations on the host come along with overheads—offloading-ready data for example, has to be aligned properly and requires the operating system to physically allocate memory immediately—yet do not interrupt the accelerator.

Our studies suggest that it is reasonable to withdraw memory management from the accelerator where possible and to assign it to the host [14]. Through a host-centric realisation, we speed up some calculations by an order of magnitude, without imposing a static offloading pattern of patches, huge patches, fixed subtimestepping, or a distributed task/patch management [14,19]. Though motivated by a real-world science case, we deliberately work with a worst-case scenario—small kernels, a memory-bound numerical scheme, and an offloading-only approach—to spotlight the challenges. Yet, we think that our techniques are of relevance for a broad range of applications that require flexible GPU usage.

In Sects. 2, we sketch our software architecture and the science cases. Measurements for a straightforward realisation with OpenMP's `map` (Sect. 3) suggest that we have to avoid the allocation and deallocation on the GPU. We introduce a realisation of this approach in Sect. 4 before we provide experimental evidence of the payoff (Sect. 5). A longer discussion of our approach in the light of existing implementations vs. fundamental challenges as well as some generic lessons learned (Sect. 6) lead into an outlook closing the discussion (Sect. 7).

2 Science Case and Code Architecture

ExaHyPE's [15] finite volume solver, which is now in its second generation, accepts hyperbolic partial differential equations (PDEs) in first-order formulation

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) + \sum_{i=1}^{d} B_i(Q)\frac{\partial Q}{\partial_{x_i}} = S(Q) \qquad \text{with } Q : \mathbb{R}^{3+1} \mapsto \mathbb{R}^N, \qquad (1)$$

describing time-dependent wave equations. ExaHyPE offers a suite of explicit time-stepping schemes for these equations: Finite Volumes (FVs), Runge-Kutta Discontinuous Galerkin (DG) and Arbitrary high order using Derivatives

(ADER)-DG (see [15,21]). Users are furthermore offered a set of solver ingredients from which they can pick to assemble their solver, while they can decide which terms of Eq. (1) to employ within the numerical scheme of choice.

Science Case. Our ambition is to study gravity and non-standard gravity models subject to strong solution gradients and solution localisation such as neutron stars or black holes. Two particular flavours of Eq. (1) demand our attention:

The Euler equations yield a system of $N = d + 2$ non-linear PDEs which describe the evolution of the scalar density, the scalar energy and the d-dimensional velocity on a cosmological scale. We employ the textbook Euler fluxes F in Eq. (1), while gravity enters the equations as source term $S(Q)$ with Q determined by the previous time step. $B_i = 0$, i.e., there are no non-conservative terms. Even though the governing PDE is non-linear, the arithmetic intensity of the arising functions is low.

With Euler, small inhomogeneities in the initial mass density distribution lead to a spherical accretion scenario: Gravity pulls more and more matter into a few overdensity centres, such that the Hubble expansion is locally compensated and we observe matter concentration instead of spreading out. Around the accretion centre, the density eventually exceeds a critical threshold and we obtain a shock which again pushes material outwards. It is an open question to which degree the temporal and spatial shape of the arising expansion and contraction horizons are preserved under non-standard gravity models [3,22].

Our second setup of interest results from a first-order formulation of the conformal and covariant Z4 (CCZ4) equations [1,8]. They are available for $d = 3$ only and model the evolution of the space-time curvature as a constrained wave equation. Different to the Euler equations, gravity is not modelled via a (quasi-)elliptic, Poisson-type term impacting some governing equations. Instead, we evolve it explicitly. CCZ4 models gravitational waves as they arise from rotating binary neutron stars, but also describes the environment around static and rotating black holes, i.e., singularities of the density concentration.

As we work with a first-order rewrite of CCZ4 to fit into the scheme of Eq. (1), we have to evolve $N = 59$ equations. Common to all codes working with variations of these equations (cf. [5,7,9,11], e.g.) is the observation that the arithmetic intensity within the PDE evaluations is very high; leading even to register spilling on GPUs [9]. In our first-order formulation (Eq. 1), this high arithmetic intensity materialises in complex B_i and S terms, while $F(Q) = 0$.

Problem Statement 1. *Both equations of interest require dynamic AMR as they study strongly localised effects. Both have high computational demands, but their compute characteristics are completely different.*

Software Architecture. For the spatial discretisation of the computational domain, ExaHyPE employs dynamically adaptive Cartesian meshes. It relies on the PDE framework Peano [20] to realise them through a generalisation of the popular octree approach: We embed the computational domain into a cube, and then subdivide this cube recursively and locally. This yields, on the finest subdivision level, an adaptive Cartesian mesh of cubes. The code thus falls into the class of octree AMR [6,14,19]. The grid structure can change at every time step.

While the code base supports various numerical discretisations, we focus in this paper on its straightforward Finite Volume solver with a generic Rusanov Riemann solver: The code embeds $p \times p$ (2D) or $p \times p \times p$ (3D) regular Cartesian meshes which we call patches into each and every cube, i.e., we work with a block-structured adaptive Cartesian mesh. This mixture of tree code and patches is popular to obtain a reasonable arithmetic load relative to the mesh management overhead (cf. [5,6,14,15,22], e.g.). The code base traverses through the mesh once per time step and progresses each patch in time. For this *compute kernel*, the actual update due to Eq. (1) is determined by the source term plus the flow through the volume faces. These terms are injected by the user via a callback mechanism. All other program logic including mesh traversal order, data storage and parallelisation is hidden. Other codes have propagated such an "inject your domain knowledge" before under the term *Hollywood principle* [20].

We employ three layers of parallelism: The domain spanned by the spacetree is first decomposed into non-overlapping chunks with one chunk per MPI rank. We cut the domain along a Peano space-filling curve (SFC) and hence end up with connected subdomains with a good surface-to-volume ratio, i.e., limited communication compared to compute load [2]. Next, we cut each MPI partition again into chunks along the SFC and deploy the resulting subdomains to the CPU's threads. We obtain hierarchical MPI+X parallelism where the threads own subdomains. Bulk-synchronous processing (BSP) is the programming model for the traversals, as the individual subdomain traversals are triggered at the same time per time step. Realisation via MPI and OpenMP is straightforward. In the context of quickly varying AMR, we however found this MPI+X parallelisation algorithmically insufficient (similar to observations by Dubey et al. [6]), as the domain decomposition on the threading side struggles to load balance.

Each thread, therefore, identifies within its subdomain patches to be deployed as separate tasks: All the patches which do feed into MPI—these patches are time-critical on supercomputers and they have to feed into MPI in-order—or have to realise adaptive mesh refinement are directly executed throughout the mesh traversals. The remaining patches are deployed as separate tasks. This *enclave tasking* concept [4] allows us to balance out imbalances between threads, i.e., within the BSP sections [16].

GPU Offloading. Furthermore, we can pool the tasks in a separate queue: We wait until this queue contains $\|\mathbb{P}_{\text{GPU}}\|$ enclave tasks ($\|\mathbb{P}_{\text{GPU}}\|$ being a user-defined threshold), and then deploy all patches within the queue in one rush to the GPU. The arising compute kernels over batches or sets of patches make up our fourth level of parallelism. Fifth, we note that our kernel implementations rely heavily on data parallelism yielding vector concurrency.

We note that the pooling or batching of tasks allows us to write GPU compute kernels that have very high concurrency [12]. The individual tasks within a batch are, by definition, all *ready* tasks, i.e., can be processed concurrently, and all of them expose additional internal concurrency on top. Our concept stands in the tradition of the enclave concept by Sundar et al. [17], who deployed subdomains to Intel Xeon Phi coprocessors. However, we do not identify the enclaves geometrically ahead of a mesh traversal—the "enclave" tasks enqueue on the fly—but can fuse segments of enclaves on the fly whenever a task that enqueues GPU tasks finds that the queue size exceeds the GPU threshold and hence deploys a whole batch to the accelerator. This added flexibility allows us to obtain large GPU offloading tasks even though the code might encounter geometrically small enclaves scattered among the threads' subregions.

Due to the processing in batches of size $\|\mathbb{P}_{\text{GPU}}\|$, a proper choice of $\|\mathbb{P}_{\text{GPU}}\|$ should allow users to exploit all parallel potential of a GPU. Contrary to that, a large $\|\mathbb{P}_{\text{GPU}}\|$ might imply that only a few batches become ready per time step and can, potentially, overlap each other [14]. We thus aim for a small $\|\mathbb{P}_{\text{GPU}}\|$ which is just about large enough to utilise the GPU efficiently. Let N_{threads} traverse their subdomain per node and produce tasks. Hence, up to N_{threads} might concurrently decide that they each would like to deploy a batch of $\|\mathbb{P}_{\text{GPU}}\|$ patches to the GPU. Many threads offload to the GPU simultaneously. A GPU serves multiple cores.

Problem Statement 2. *In ExaHyPE, multiple threads offload to the GPU simultaneously. Due to the dynamic AMR, the offloading pattern is not deterministic or known beforehand.*

3 A Realisation of GPU Offloads with `target map`

Let $\mathcal{K}_p^{\text{Euler,2D}}$, $\mathcal{K}_p^{\text{Euler,3D}}$ and $\mathcal{K}_p^{\text{CCZ4}}$ describe the compute kernels of interest. Each kernel takes the solution representation over a patch of $p \times p$ or $p \times p \times p$ finite volumes and returns the solution at the next timestep. When we benchmark the whole patch update cycle of such an update, we actually measure the cost including all data transfer, i.e., we measure

$$\big(\mathcal{R} \circ \mathcal{F} \circ \mathcal{K} \circ \mathcal{A} \circ \mathcal{P}\big) Q(t)$$

where the operator $\mathcal{P}$ takes the solution $Q(t)$ and transports it to the GPU, while $\mathcal{R}$ retrieves the solution and brings it back into the user memory. $\mathcal{A}$ allocates on the device all temporary variables required by $\mathcal{K}$, while $\mathcal{F}$ frees these memory blocks. ExaHyPE works with sets of patches and therefore processes sets

$$\Big\{\big(\mathcal{R} \circ \mathcal{F} \circ \mathcal{K} \circ \mathcal{A} \circ \mathcal{P}\big) Q_c(t)\Big\}_{c \in [1, \|\mathbb{P}_{\text{GPU}}\|]}. \tag{2}$$

Realisation. Our plain realisation of the GPU offloading through OpenMP implements Eq. (2) as follows (also, cf. Algorithm 1):

1. The data per patch are stored en bloc in one large array of structures (AoS) on the host, but the individual patches are scattered over the main memory as we invoke the batched GPU kernel (cf. Eq. 2). We thus deep copy a list of

Algorithm 1: OffloadMap($\|\mathbb{P}_{\text{GPU}}\|$):
Offloads $\|\mathbb{P}_{\text{GPU}}\|$ to the GPU using OpenMP's `map` clause. First, patch and temporary data are allocated on the host and the respective device pointers are constructed. After offloading to the GPU, results are copied back to the host and the data is freed.

```
1  Procedure offload_map($\|\mathbb{P}_{\text{GPU}}\|$, host_patch_data):
2  mapped_pointers ← allocate_host($\|\mathbb{P}_{\text{GPU}}\|$)
3  for $i \leftarrow 0$ to $\|\mathbb{P}_{GPU}\|$ do
4      patch_data ← host_patch_data[$i$]
5      #pragma omp target enter data map(to:patch_data)
6      mapped_pointers[i] ← omp_get_mapped_ptr(patch_data)
7  end
8  temporary_data ← allocate_host($\|\mathbb{P}_{\text{GPU}}\|$)
9  #pragma omp target teams distribute map(to:mapped_pointers)
    map(alloc:temporary_data)
10 for $i \leftarrow 0$ to $\|\mathbb{P}_{GPU}\|$ do
11     // Do computations on Finite Volumes
12 end
13 temporary_data ← free_host()
14 for $i \leftarrow 0$ to $\|\mathbb{P}_{GPU}\|$ do
15     patch_data ← host_patch_data[$i$]
16     #pragma omp target exit data map(from:patch_data)
17 end
18 mapped_pointers ← free_host()
```

pointers to patch data to the device: A for loop maps each patch's data onto the device trough `omp target enter data map(to:...)`. Due to the loop, the kernel can handle arbitrary $\|\mathbb{P}_{\text{GPU}}\|$. The copying per se is trivial, as the patch data is one large, continuous array of doubles. After that, we construct the list of pointers on the GPU and befill it with the device pointers. For this, OpenMP offers `declare mapper` constructs though we prefer to build up the list of device pointers via `omp_get_mapped_ptr`.

2. For all temporary data that the kernel requires to handle the $\|\mathbb{P}_{\text{GPU}}\|$ patches, we allocate one large block. There is only one $\mathcal{A}_{\|\mathbb{P}_{\text{GPU}}\|}$ allocation realised through a `map(alloc:...)`.
3. The actual kernel invocation is an `omp target` block supplemented with a `distribute` directive.
4. We free all temporary data in one rush.
5. With OpenMP's `map` clauses, copying the GPU outcomes back into the host memory is realised by a loop over the patches. We issue one `omp target exit data map(from:... )` call per patch and time step.

We conceptually end up with the following realisation of Eq. (2):

$$\left\{\mathcal{R}^{\text{map}}\right\}_{c\in[1,\|\mathbb{P}_{\text{GPU}}\|]} \circ \mathcal{F}_{\|\mathbb{P}_{\text{GPU}}\|} \circ \left\{\mathcal{K}\right\}_{c\in[1,\|\mathbb{P}_{\text{GPU}}\|]} \circ \mathcal{A}_{\|\mathbb{P}_{\text{GPU}}\|} \circ \left\{\mathcal{P}^{\text{map}}\right\}_{c\in[1,\|\mathbb{P}_{\text{GPU}}\|]}. \tag{3}$$

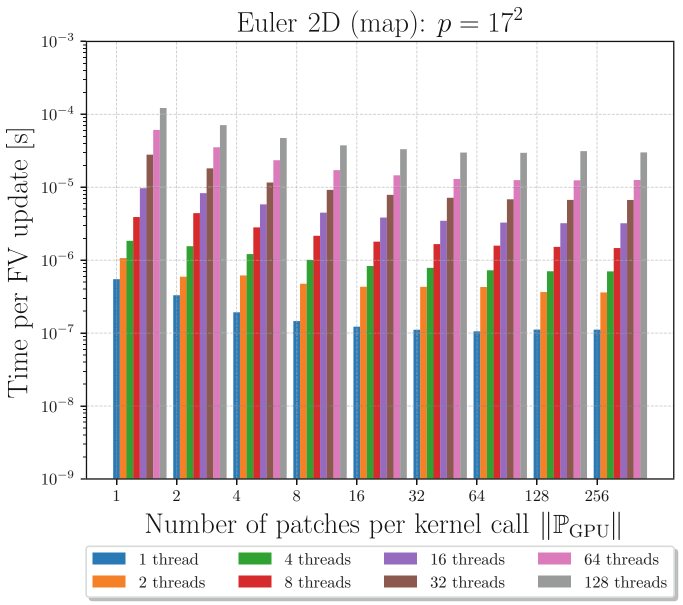

Fig. 1. Time per degree of freedom update for the 2D Euler equations with patch size 17×17. We benchmark the throughput for different numbers of patches $\|\mathbb{P}_{\text{GPU}}\|$ handled by each kernel invocation (x-axis). We also let different thread counts N_{threads} access the GPU at the same time (different bars).

The scheme mirrors batched linear algebra, where a matrix is applied to multiple right-hand sides in one sweep. Each kernel $\mathcal{K}$ has some internal concurrency such as "loop over all faces" or "loop over all volumes". The batching pays off, as we obtain, on top of this, another outer loop over $[1, \|\mathbb{P}_{\text{GPU}}\|]$ which we annotate with OpenMP's `distribute`. We leave it to OpenMP to distribute the patches over the streaming multiprocessors (SMs) or to deploy multiple patches onto one SM via multiple warps, while the SM threads are used to process all finite volumes within a patch.

Experimental Setup. To benchmark the offloading including all data transfers, we disable all calculations on the host, we artificially ensure that all work between the CPU threads is perfectly balanced, we make all cores use one GPU only, and we disable MPI. Next, we vary the number of threads N_{threads} which offload to the GPU simultaneously and let each thread deploy 100 patches, grouped into sets of $\|\mathbb{P}_{\text{GPU}}\|$. In the big picture, it translates to a setup where each GPU of a compute node is governed by one MPI rank, in which multiple threads run in parallel, flooding the GPU with offloaded tasks. However, real simulations barely will encounter situations where all N_{threads} threads offload exactly at the same time. We focus on this worst-case constellation. It is a stress test.

All tests are run on the Alex cluster hosted by the Erlangen National High Performance Computing Center (NHR). Each node of our testbed is equipped with two AMD EPYC 7713 Milan processors (64 cores per chip) which accommodate eight NVIDIA A100 GPUs. Each GPU features 80 GB main memory. Our experiments use the NVIDIA HPC Software Development Kit (SDK) in the version 23.1 as well as the Compute Unified Device Architecture (CUDA) in the version 12.0 as the underlying software stack. NVIDIA SDK's collection of compilers, libraries, and software tools supports OpenMP GPU target offloading and OpenMP loop transformations as used by our compute kernels. However, some features have to be used with care[2].

Benchmark Results. For all different kernel variants as well as p choices, we get qualitatively similar results (Fig. 1):

Observation 1. *It is important to batch multiple patches into one GPU compute kernel to exploit the hardware concurrency of the accelerator.*

This is not a surprising observation once we take into account what hardware concurrency current GPUs offer. Our data however showcase that we quickly run into some saturation: Regardless of the number of threads used, the measured time per finite volume update decreases until it starts to saturate from around $\|\mathbb{P}_{\text{GPU}}\| = 16$ patches. It barely pays off to merge more than $\|\mathbb{P}_{\text{GPU}}\| = 16$ patches for the two-dimensional Euler. With a patch size of 17×17 and 128 threads, the GPU becomes saturated by keeping around $5.24 \cdot 10^5$ finite volume updates in flight at any point.

Observation 2. *If we launch multiple kernels from multiple threads, the performance of our straightforward implementation deteriorates.*

In theory, spawning kernels in parallel from multiple threads should pay off performance-wisely, as we can hide memory transfers of one kernel behind the computations of another kernel that has already started to work. Tutorials around the usage of streaming buffers or `nowait` (async) kernel launches exploit this. Even as the threads start to offload at the same time in a stress test, we should at least be able to scale up to the number of supported hardware streams. Our data however show that simultaneously firing kernels to the GPU reduces the throughput by at least two orders of magnitude. This is counterintuitive!

Observation 3. *The cheaper a compute kernel, the more severe the impact of concurrent data transfers.*

For the three-dimensional Euler, $\|\mathbb{P}_{\text{GPU}}\| = 8$ is reasonable, while CCZ4 has $\|\mathbb{P}_{\text{GPU}}\| = 4$ (not shown). As the saturation thresholds are lower, the penalties for concurrent kernel launches kick in stronger for lower thread counts.

Rationale. While modern GPUs can manage several compute kernels in flight, the maximum number of such kernels is relatively small, and each kernel launch

[2] See https://doi.org/10.5281/zenodo.7741217 for supplemental material.

introduces overhead. While this motivates why the code benefits from larger $\|\mathbb{P}_{\text{GPU}}\|$, it does not explain the penalties resulting from parallel kernel launches from multiple threads. It does not explain the performance degradation once we increase N_{threads}.

GPU offloading in OpenMP is realised through address mapping: The runtime manages a table per accelerator which stores which addresses from the CPU are mapped onto which GPU addresses including the corresponding memory sizes. From these data, the runtime can identify all reachable memory regions on the accelerator. An allocation of a new memory region on the GPU inserts a new entry into the device table. If an entry is removed, subsequent inserts will be able to use the "freed" memory regions again.

If any thread accesses the memory table, the runtime first has to avoid races with other threads. Secondly, the GPU itself might want to allocate GPU memory. Our kernels do not require dynamic memory, but it is not clear to what degree the compiler synthesises this knowledge from the source code. Thirdly, the GPU hardware can only read from page-locked host-pinned memory to copy data from the host to the device. In general, memory passed to `target map` is not page-locked. Therefore, additional `staging` is required. Our data suggest that this triad of challenges makes the memory manager suspend all running GPU kernels before it allocates or frees memory.

Observation 4. *Memory allocations on the GPU are expensive and potentially delay running kernels if multiple threads offload to the GPU concurrently.*

We consider this final Observation 4 to be a flaw in GPU offloading runtimes. To the best of our knowledge, it does not attract attention in current literature.

4 User-Managed Memory Management

To avoid memory allocations on the GPU, we propose to make the host the owner of the memory blocks on the GPU which are used for host-GPU data transfer. We propose to introduce a GPU memory manager [14], and we provide two realisations of such a manager.

Algorithmic Framework. Let each rank hold one instance of a GPU memory manager. Without loss of generality, we can assume that there is one manager per host CPU, i.e., for all N_{threads} threads sharing one GPU. If a code wants to deploy a memory chunk to the accelerator, it *allocates* memory through the GPU memory manager by passing the size of the memory chunk plus its address, as well as a device number if multiple GPUs are hosted on one node. The allocation routine returns a device pointer, i.e., an address that is valid on the respective device. The GPU memory manager guarantees that the resulting device pointer points to a valid device region that can be accessed consecutively from the calling code. The counterpart of the allocation is a *free* which releases the device memory. It is given another host address into which the GPU memory manager dumps the kernel results. Access to the GPU memory manager is made thread-safe through global semaphores—which is sufficient, as the map simply handles out pointers.

Internally, the GPU memory manager hosts a hash map of tuples of integers onto a sequence of tuples of device addresses plus a boolean marker:

$$M : \mathbb{N}^+ \times \mathbb{N}^+ \mapsto \left(\mathbb{A} \times \{\top, \bot\}\right)^+.$$

The key tuple represents the combination of the device number and memory block size (to be allocated). When the code requests (allocates) memory on a particular device of a particular size, we construct the key and study the image in M which is a sequence of addresses on the device. Each address either holds $\top$ which means that this address is currently in use. The manager may not hand out this address again. If it is labelled with $\bot$, then the address is not in use.

If the GPU memory manager can serve an allocation with an existing address with the label $\bot$, it toggles the flag to $\top$ and returns the corresponding address. If there is no address with $\bot$ available—and notably if a key tuple points to an empty list—it is the GPU memory manager's responsibility to acquire new GPU memory and then return the corresponding address. When memory is freed, the manager retrieves the result from the GPU into the user address space that is passed. After that, it sets the corresponding entry in M to $\bot$. As our compute kernels rely on the managed memory, they can use the manager's returned pointers within `target` compute kernels by labelling them as `is_device_ptr` and effectively avoiding the staging of host memory.

The algorithmic framework sketches a code utility that allocates memory and hands it out upon demand. As it does not free memory but re-uses memory blocks which are not in use anymore, we avoid repeated memory allocations. Notably, we share pre-allocated data between different threads. The exact allocation mechanism is subject to two different realisation flavours.

4.1 Data Pre-allocation on the GPU

A GPU-centric variant of the GPU memory manager acquires all memory requested via `omp_target_alloc` directly on the GPU: If no free memory blocks are held within M, we reserve GPU memory and store the GPU memory's address within the hash map. Whenever we identify a fitting pre-allocated memory region (or have literally just acquired memory), the manager transfers the user data to the allocated memory via an `omp_target_memcpy`. Bringing data back is another explicit `omp_target_memcpy` call (cf. Algorithm 2).

Employing the OpenMP API routines mirrors the behaviour behind the `map(alloc)` and `map(to)` pragma clauses. Internally, the compiler breaks down an `omp target enter data map(to: ...)` statement into $\left(\mathcal{P} \circ \mathcal{A}\right)$, where the $\mathcal{A}$ operator denotes the explicit memory allocation on the GPU via `omp_target_alloc` which is followed by the actual data transfer.

Algorithm 2: OffloadManaged($\|\mathbb{P}_{\text{GPU}}\|$):
Offloads $\|\mathbb{P}_{\text{GPU}}\|$ to the GPU using the managed memory approach. We allocate patch and temporary data through the GPU memory manager. After offloading to the GPU, results are copied back to the host and the data handles are freed for re-use.

```
Procedure offload_managed($\|\mathbb{P}_{\text{GPU}}\|$, host_patch_data):
patch_data ← GPUMemoryManager→allocate_device($\|\mathbb{P}_{\text{GPU}}\|$)
patch_data ← omp_target_memcpy(host_patch_data, $\|\mathbb{P}_{\text{GPU}}\|$)
temporary_data ← GPUMemoryManager→allocate_device($\|\mathbb{P}_{\text{GPU}}\|$)
#pragma omp target teams distribute is_device_ptr(patch_data,
  temporary_data)
for $i \leftarrow 0$ to $\|\mathbb{P}_{GPU}\|$ do
    // Do computations on Finite Volumes
end
temporary_data ← GPUMemoryManager→free()
host_patch_data ← omp_target_memcpy(patch_data, $\|\mathbb{P}_{\text{GPU}}\|$)
patch_data ← GPUMemoryManager→free()
```

Compared to Eq. (3), the present GPU memory manager variant eliminates, in most cases, the allocations, while we omit the frees. In all cases where our pre-allocated memory regions can serve the user code requests, we reduce the actual kernel invocation cost to

$$\left\{\mathcal{R}^{\text{copy}}\right\}_{c\in[0,|\mathbb{C}|-1]} \circ \left\{\mathcal{K}\right\}_{c\in[0,|\mathbb{C}|-1]} \circ \left\{\mathcal{P}^{\text{copy}} \circ \hat{\mathcal{A}}\right\}_{c\in[0,|\mathbb{C}|-1]}, \tag{4}$$

where $\hat{\mathcal{A}}$ is a no-operation. Only in the cases where we cannot serve a memory request with pre-allocated memory blocks, $\hat{\mathcal{A}}$ becomes an actual $\mathcal{A}$. As the $\mathcal{F}$ and $\mathcal{A}$ operations halt the GPU temporarily, we obtain a fast code stripped of these stops. Solely the orchestration overhead to launch the batched compute kernel for $\left\{\mathcal{K}\right\}_{c\in[0,|\mathbb{C}|-1]}$ remains.

4.2 Pre-allocation on the CPU with Unified Memory

Our second approach works on GPUs which offer unified memory. In this case, we exploit that the GPU has full access to the host address space and that the hardware can migrate pages from the main memory via page faults to the CPU upon demand. GPU and CPU form one NUMA domain.

On such systems, it is possible to replace all memory allocations with a shared allocation, i.e., to enable the system to migrate any data automatically between host and accelerator. However, such allocations differ from "normal" allocations in that they induce particular memory layouts and arrangements. Even though the NVIDIA software stack allows developers to enable this *CUDA Unified Memory* globally at compile time through a flag, we refrain from using it globally, as it is not compatible with static memory regions employed for global constants, e.g. Instead, we distinguish the allocation on the GPUs $\mathcal{A}$ from an allocation $\mathcal{A}^{\text{shared}}$ on the CPU which allocates memory that can be transferred to the GPU. For the latter, we introduce the memory copy operators $\mathcal{C}^{\text{host}\mapsto\text{shared}}$ and $\mathcal{C}^{\text{shared}\mapsto\text{host}}$. While $\mathcal{P}$ transfers data directly to the GPU, both $\mathcal{C}$ operators copy data on the host from a "normal" memory region into a region that can be migrated to the GPU upon demand. They are plain CPU memory copies between two memory regions on the host.

Whenever the GPU kernel accesses a unified memory region that resides on the host the access might page fault and trigger a page migration to the GPU which we denote as $\mathcal{P}^{\text{pf}}$ with pf for page fault. Moving data back would be $\mathcal{R}^{\text{pf}}$. With this formalism, our kernel launch becomes

$$\left\{\mathcal{C}^{\text{shared}\mapsto\text{host}} \circ \mathcal{R}^{\text{pf}}\right\}_{c\in[0,|\mathbb{C}|-1]} \circ \left\{\mathcal{K} \circ \mathcal{P}^{\text{pf}}\right\}_{c\in[0,|\mathbb{C}|-1]} \circ \left\{\mathcal{C}^{\text{host}\mapsto\text{shared}}\right\}_{c\in[0,|\mathbb{C}|-1]}. \tag{5}$$

Initially, we simply copy our data on the host. This is cheap compared to the data transfer to the GPU. Notably, nothing stops multiple threads to copy their data in parallel, once the GPU memory manager has identified or created well-suited shared memory blocks. Immediately after that, we launch the kernel. In RAM, the algorithmic latency caused by memory transfers is significantly lower than in our previous managed approach, while the bandwidth is usually higher. While the kernel launches immediately, it has to retrieve data from the host via page fault $\mathcal{P}^{\text{pf}}$. The actual data transfer is prolonged yet has the potential to delay the kernel execution. However, it is realised in hardware, and the GPU is good at orchestrating such data transfer. Getting data back is again a relatively cheap host-internal memory transfer which might however trigger data transfers $\mathcal{R}^{\text{pf}}$ back from the GPU. There might be additional latency here, though the GPU hardware might also trigger the corresponding page faults ahead of time.

Technically, the second variant is close to trivial: We take the first managed approach and replace the OpenMP memory allocations with NVIDIA's CUDA allocations. The OpenMP data copies become plain C++ memory copies.

5 Results

We first benchmark the three-dimensional Euler equations and the CCZ4 setup where the offloading's data transfer is organised via a plain `map`. A fixed total number of patches is split into chunks (batches) of $\|\mathbb{P}_{\text{GPU}}\|$ and offloaded concurrently by the threads to the GPU. We vary the thread count N_{threads}. This is a classic strong scaling setup that again simplifies real-world simulation runs where the $N_{\text{threads}} \cdot \|\mathbb{P}_{\text{GPU}}\|$ patches never become available in one rush. The total number of patches used is empirically chosen such that the average runtime per patch becomes invariant, i.e., the experimental setup avoids burn-in effects.

Our data (Fig. 2) confirm that both bigger patch sizes p and higher number of patches per batch $\|\mathbb{P}_{\text{GPU}}\|$ pay off performance-wisely. However, the size of the individual patches is the more decisive performance lever: We can distribute a batch of patches over the GPU streaming multiprocessors but if the individual patch is very small, the batching is not able to close the performance gap to

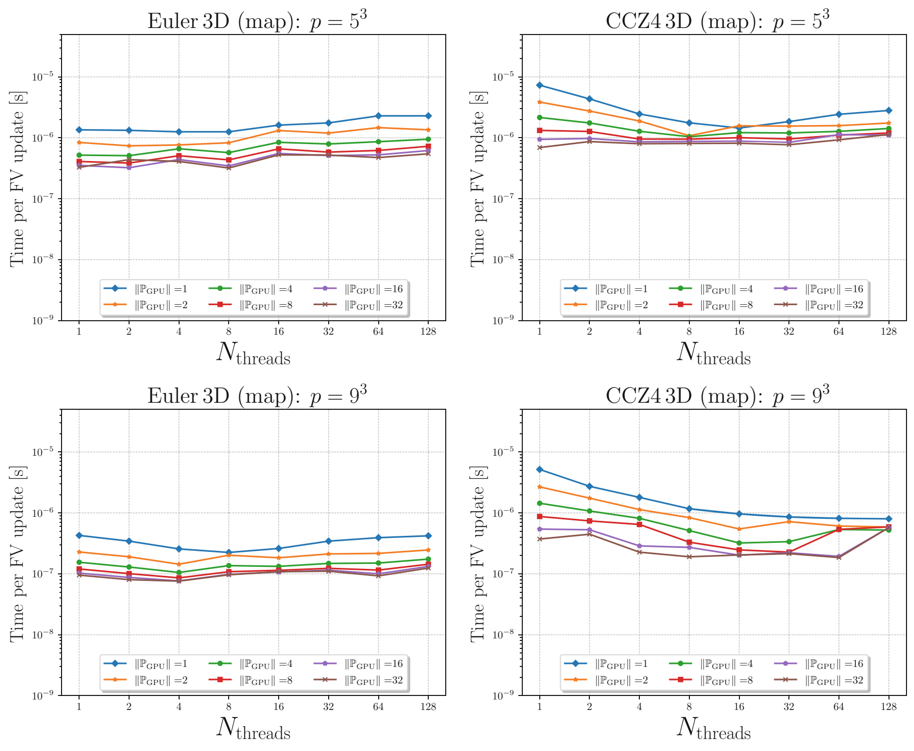

Fig. 2. Time per FV degree of freedom (volume) update for three-dimensional Euler (left) and CCZ4 (right). Lower is better. Each patch either hosts 5^3 (top) or 9^3 (bottom) Finite Volumes along each Cartesian coordinate axis. All data transfer is realised through OpenMP's `map` clauses (cf. Sect. 3).

a run employing big expensive patches right from the start. In any setup, the N_{threads} hold $N_{\text{threads}} \cdot \|\mathbb{P}_{\text{GPU}}\| \cdot p^3$ finite volumes in flight on the GPU. While this corresponds to a reasonable memory footprint for CCZ4 with its 59 unknowns per volume—we also have to allocate temporary data for the non-conservative fluxes in all three directions plus the source term—the saturation for $\|\mathbb{P}_{\text{GPU}}\| \approx 8$ is reached early.

All measurements confirm that offloading to the GPU simultaneously from many threads comes along with a significant performance penalty. If we split up the total work equally among the threads and deploy the batches concurrently, the throughput relative to the threads plateaus quickly and eventually rises again. Indeed, we see a speedup if and only if the number of offloading threads is small, and if we offload only a few patches per batch.

When we rerun the experiments with our GPU memory manager, we dramatically improve the robustness of the concurrent offloading (Fig. 3). Batching, i.e., large $\|\mathbb{P}_{\text{GPU}}\|$, and reasonably large individual patches aka p remain key performance ingredients, but concurrent offloading can help robustly to improve

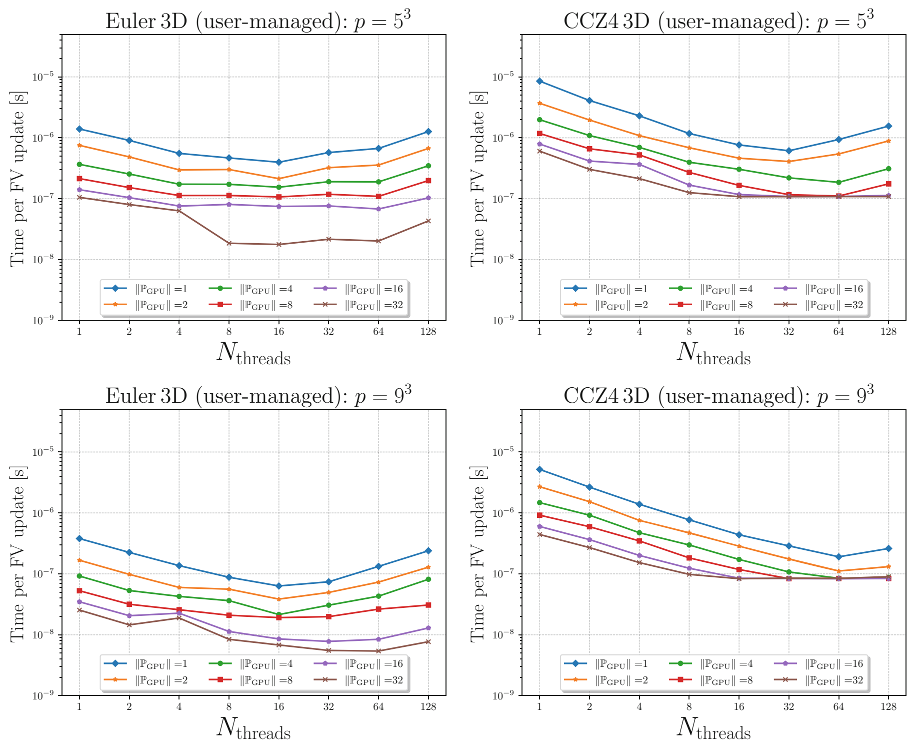

Fig. 3. Experiments from Fig. 2 with our user-managed memory (cf. Sect. 4.1).

performance. Robust here means that the gain through multi-threaded offloading might eventually plateau, yet, we do not pay a performance penalty. Given enough threads that access the GPU at the same time, setups employing smaller $\|\mathbb{P}_{\text{GPU}}\|$ match the throughput of setups with large $\|\mathbb{P}_{\text{GPU}}\|$. We also see that the best-case throughput becomes independent of the p-choice.

Once we replace our manual copies with pre-allocated GPU memory using CUDA unified memory, concurrent offloading to the GPU yields no performance improvement anymore and overall runtime suffers (Fig. 4). With CUDA unified memory, we postpone the data movement penalty to the point when the data is actually required. As the data access is spread out temporarily, bandwidth demands are spread out, too. This however does not manifest in better performance. Notably, it stops us from profiting from multiple threads which offload at the same time—we assume that the interconnect is kept busy by a single kernel already and multiple kernel launches interfere and compete with each other.

We provide full details on how to reproduce the results presented in this paper on https://doi.org/10.5281/zenodo.7741217.

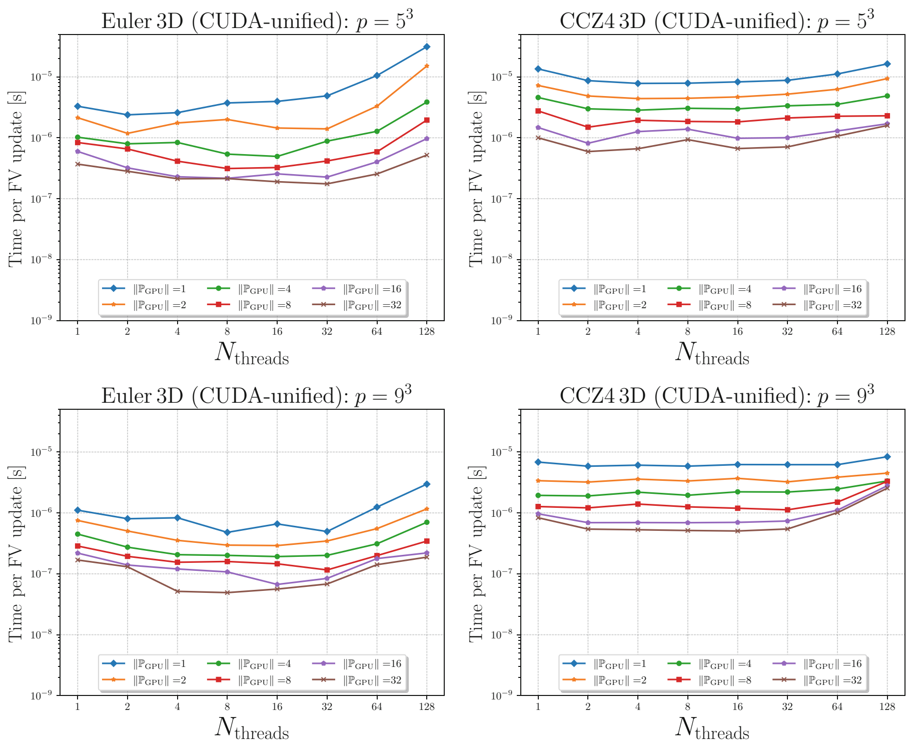

Fig. 4. Experiments from Fig. 2, using CUDA-unified memory (cf. Sect. 4.2): The GPU memory manager accepts data that has to be offloaded and packs it into unified memory on the host. It is then the responsibility of the CUDA runtime to bring the data from shared managed memory regions into the GPU.

6 Discussion and Conclusions

Our observations, proposed solutions, and runtime measurements allow us to draw conclusions for our algorithms as well as the used runtime:

Tasks of high computational load are important to exploit the concurrency on modern GPUs, and batching is one technique to construct such tasks while we stick to small patches (cf. Observation 1). With the concurrent offloading onto the GPU through multiple threads, we eventually manage to make the best-case throughput of this combination independent of p, and we are able to reduce the minimal batch size $\|\mathbb{P}_{\text{GPU}}\|$. Yet, a reasonable value continues to depend on the algorithmic fingerprint of the underlying PDE. We assume that other discretisations such as DG have a major impact here, too.

The insight contradicts the rule of thumb knowledge which suggests that an efficient GPU utilisation becomes impossible in the presence of a totally adaptive AMR with tiny patches. If we employ tiny patches where the meshing has to track data flow and dependencies between small Cartesian meshes, the streaming compute properties per patch are not sufficient to keep a GPU busy [14]. However, small patches, i.e., small p-values in our case, are key to efficient AMR in an algorithmic sense: Large patches constrain the AMR, as we cannot represent rapid resolution changes accurately. They hence reduce the algorithmic efficiency, i.e., invested cost per numerical accuracy.

Conclusion 1. *The combination of parallel offloading, user-managed GPU memory, and batching yields a fast GPU code that works with relatively small patches.*

Indeed, our approach abandons the concept of a geometric "streamability" and instead translates this idea into the data space: Patches are batched into sets that can be processed in a streaming fashion, even though there might be no geometric correlation between those patches.

Our data suggest that OpenMP GPU offloading is vulnerable to concurrent access by multiple threads. Our GPU memory manager mitigates this shortcoming and renders Observations 2 and 3 invalid. It is not clear if the need for it (cf. Observation 4) is a shortcoming of the employed GPU runtime or an intrinsic property of any GPU runtime, as we obtained qualitatively comparable data for LLVM and AMD's runtime, too. We hypothesise that, as long as a GPU kernel is allowed to make dynamic allocations, any GPU allocation has to be thread-safe and hence introduces some synchronisation: To be thread-safe, any data transfer to the GPU has to stop all running kernels to prevent them to make allocations; unless the memory region for data exchange and the local heap is strictly separated, or a compiler derives a priori if a kernel does not require dynamic memory allocation and hence does not need to be stopped.

Conclusion 2. *Multithreaded access to GPU offloading in combination with dynamic memory allocation on the device requires special care on the programmer's side and eventually benefits from a deployment of the GPU's memory management onto the CPU.*

This argument gains importance for software which—in line with ExaHyPE—deploys algorithmically irregular and unstructured operations such as AMR administration to the CPU, yet keeps other data and work persistently on the GPU [19]. It might notably gain weight in the context of local time stepping, where patch interaction patterns quickly become challenging.

Conclusion 3. *Our data do not support the idea that managed memory is a competitive replacement for well-designed manual data migration of dynamically allocated memory regions.*

Our data yields "disappointing" results for managed memory, much in line with disappointing data of cache architectures compared to algorithms which explicitly exploit write-through or streaming capabilities. However, we have exclusively studied an offloading approach which requires dynamic allocations within the managed memory, and we have used a code base which is likely PCIe latency-bound. In this context, we assume that managed memory in combination with CUDA prefetching allows for significantly more elegant and faster code.

7 Summary and Outlook

With the advent of more and more cores on the host and with more GPUs being added to each node, in-depth analysis, and discussion around multi-threaded accelerator usage is imminent. Our work orbits around flaws that we document for the multithreaded usage of GPUs. Future versions of the employed OpenMP runtimes might fix those flaws and supersede our user-defined memory management. Even so, any future runtime development has to be contextualised in which way software operates GPUs:

Keeping data permanently on the accelerator [19] is beyond the scope of the present studies, as we let our ExaHyPE solver construct worst-case stress tests where each and every patch is offloaded to the GPU and eventually brought back. In contrast, many simulation codes try to hold data on the GPU as long as possible, i.e., let the GPU own the data, as the fastest data transfer is avoided data transfer. Therefore, it remains relevant to assess to which degree data transfer has to interrupt running GPU kernels. For our GPU memory manager, a fix could imply that parts of the memory administration are deployed to the GPU, i.e., that the GPU memory manager is distributed, too. We furthermore hypothesise that kernels could continue to run despite threads offloading to the GPU as long as the compiler is aware that no dynamic memory allocation is required for these kernels, and as long as the compiler can derive the maximum call stack size. No interaction with any dynamic memory management should be required.

Our work confirms that a GPU performs best if we deploy kernels with a huge concurrency level. We achieve this through batching. As the strict rule of lock-stepping comes to an end on the hardware side, we assume that smaller and smaller batch sizes become feasible. In this context, it remains to be seen if fewer restrictions on the lock-stepping side go hand in hand with the support of more active kernels and how this affects the concurrent offloading to the GPU from many threads.

Our GPU memory manager is basic and can be improved in many ways. A canonical extension is garbage collection, e.g., [14]. Fundamentally new challenges arise from switching to a multi-process view: Once multiple ranks are deployed per CPU—to accommodate multiple NUMA domains, e.g.,—our GPU memory manager becomes a distributed memory allocator which requires cross-process synchronisation and the coordination of multiple ranks requesting access to the GPU's pre-allocated memory at the same time. It remains open to which degree future OpenMP runtimes can and will accommodate the requirement to support multi-rank setups, and in which way they support a dynamic association of GPU compute resources to these ranks.

Acknowledgements. This research has been supported by EPSRC's ExCALIBUR programme through its cross-cutting project EX20-9 *Exposing Parallelism: Task Parallelism* (Grant ESA 10 CDEL). It uses the code base Peano as supported through ExCALIBUR's project *PAX–HPC—Particles At eXascale on High Performance Computers* (EP/W026775/1) and *MGHyPE—An ExCALIBUR Multigrid Solver Toolbox for ExaHyPE* (EP/X019497/1). We appreciate the support by the German Ministry of Education and Research (BMBF) via the project targetDART (16ME0634K), and by Intel's Academic Centre of Excellence at Durham University which allowed the team to investigate into OpenMP GPU offloading as facilitated through OpenMP. We also gratefully acknowledge support through the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk) under grant no ARCHER2-eCSE04-2, and the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nurnberg (FAU) under the NHR project PeanoMP. NHR funding is provided by federal and Bavarian state authorities. NHR@FAU hardware is partially funded by the German Research Foundation (DFG)—440719683. The authors acknowledge Jülich Supercomputing Center for providing access to the JURECA DC Evaluation Platform. This work was completed in part at the CINECA GPU Hackathon, part of the Open Hackathons program. Finally, the authors would like to acknowledge OpenACC-Standard.org for their support.

References

1. Alic, D., Bona-Casas, C., Bona, C., Rezzolla, L., Palenzuela, C.: Conformal and covariant formulation of the Z4 system with constraint-violation damping. Phys. Rev. D **85**(6), 064040 (2012)
2. Bader, M.: Space-Filling Curves–An Introduction with Applications in Scientific Computing. Texts in Computational Science and Engineering, vol. 9. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-31046-1
3. Bertschinger, E.: Self-similar secondary infall and accretion in an Einstein-de Sitter universe. Astrophys. J. Suppl. Ser. **58**, 39–65 (1985)
4. Charrier, D., Hazelwood, B., Weinzierl, T.: Enclave tasking for DG methods on dynamically adaptive meshes. SIAM J. Sci. Comput. **42**(3), C69–C96 (2020)
5. Daszuta, B., Zappa, F., Cook, W., Radice, D., Bernuzzi, S., Morozova, V.: GR-Athena++: puncture evolutions on vertex-centered oct-tree adaptive mesh refinement. Astrophys. J. Suppl. Ser. **257**(2), 25 (2021)
6. Dubey, A., Berzins, M., Burstedde, C., Norman, M.L., Unat, D., Wahib, M.: Structured adaptive mesh refinement adaptations to retain performance portability with increasing heterogeneity. Comput. Sci. Eng. **23**(05), 62–66 (2021)
7. Dumbser, M., Fambri, F., Tavelli, M., Bader, M., Weinzierl, T.: Efficient implementation of ADER discontinuous Galerkin schemes for a scalable hyperbolic PDE engine. Axioms **7**(3), 63 (2018)
8. Dumbser, M., Guercilena, F., Köppel, S., Rezzolla, L., Zanotti, O.: Conformal and covariant Z4 formulation of the Einstein equations: strongly hyperbolic first-order reduction and solution with discontinuous Galerkin schemes. Phys. Rev. D **97**, 084053 (2018)
9. Fernando, M., et al.: A GPU-accelerated AMR solver for gravitational wave propagation. In: 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1078–1092. IEEE Computer Society (2022)
10. Huber, J., et al.: Efficient execution of OpenMP on GPUs. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 41–52 (2022)
11. Kidder, L., et al.: SpECTRE: a task-based discontinuous Galerkin code for relativistic astrophysics. J. Comput. Phys. **335**, 84–114 (2017)

12. Li, B., Schulz, H., Weinzierl, T., Zhang, H.: Dynamic task fusion for a block-structured finite volume solver over a dynamically adaptive mesh with local time stepping. In: Varbanescu, A.L., Bhatele, A., Luszczek, P., Marc, B. (eds.) ISC High Performance 2022. LNCS, vol. 13289, pp. 153–173. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07312-0_8
13. Peterson, B., et al.: Automatic halo management for the Uintah GPU-heterogeneous asynchronous many-task runtime. Int. J. Parallel Programm. **47**(5–6), 1086–1116 (2018). https://doi.org/10.1007/s10766-018-0619-1
14. Qin, X., LeVeque, R., Motley, M.: Accelerating an adaptive mesh refinement code for depth-averaged flows using GPUs. J. Adv. Model. Earth Syst. **11**(8), 2606–2628 (2019)
15. Reinarz, A., et al.: ExaHyPE: an engine for parallel dynamically adaptive simulations of wave problems. Comput. Phys. Commun. **254**, 107251 (2020)
16. Schulz, H., Gadeschi, G.B., Rudyy, O., Weinzierl, T.: Task inefficiency patterns for a wave equation solver. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 111–124. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_8
17. Sundar, H., Ghattas, O.: A nested partitioning algorithm for adaptive meshes on heterogeneous clusters. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 319–328 (2015)
18. 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
19. Wahib, M., Maruyama, N., Aoki, T.: Daino: a high-level framework for parallel and efficient AMR on GPUs. In: SC 2016: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 621–632 (2016)
20. Weinzierl, T.: The Peano software–parallel, automaton-based, dynamically adaptive grid traversals. ACM Trans. Math. Softw. **45**(2), 14 (2019)
21. Zanotti, O., Fambri, F., Dumbser, M., Hidalgo, A.: Space-time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting. Comput. Fluids **118**, 204–224 (2015)
22. Zhang, H., Weinzierl, T., Schulz, H., Li, B.: Spherical accretion of collisional gas in modified gravity I: self-similar solutions and a new cosmological hydrodynamical code. Mon. Not. Roy. Astron. Soc. **515**(2), 2464–2482 (2022)