# Octo-Tiger's New Hydro Module and Performance Using HPX+CUDA on ORNL's Summit

Patrick Diehl*, Gregor Daiß†, Dominic Marcello*, Kevin Huck‡, Sagiv Shiber§, Hartmut Kaiser*, Juhan Frank§, Geoffrey C. Clayton§, and Dirk Pflüger†
*LSU Center for Computation & Technology, Louisiana State University, Baton Rouge, LA, 70803 U.S.A
Email: patrickdiehl@lsu.edu
† IPVS, University of Stuttgart, Stuttgart, 70174 Stuttgart, Germany
‡ *OACISS*, University of Oregon, Eugene, OR, U.S.A.
§ Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA, 70803 U.S.A.

***Abstract*—Octo-Tiger is a code for modeling three-dimensional self-gravitating astrophysical fluids. It was particularly designed for the study of dynamical mass transfer between interacting binary stars. Octo-Tiger is parallelized for distributed systems using the asynchronous many-task runtime system, the C++ standard library for parallelism and concurrency (HPX) and utilizes CUDA for its gravity solver. Recently, we have remodeled Octo-Tiger's hydro solver to use a three-dimensional reconstruction scheme. In addition, we have ported the hydro solver to GPU using CUDA kernels. We present scaling results for the new hydro kernels on ORNL's Summit machine using a Sedov-Taylor blast wave problem. We also compare Octo-Tiger's new hydro scheme with its old hydro scheme, using a rotating star as a test problem.**

## I. Introduction

Octo-Tiger is an astrophysics finite volume hydrodynamic code for simulating the evolution of stellar systems [1]. Octo-Tiger consists of several modules, *e.g.* hydro, gravity, and radiation. The gravity is solved based on the fast multipole method using adaptive octrees. The hydro module solves the mass, momentum and energy equations of an inviscid fluid in a rotating frame of reference, which reduces numerical viscosity effects. Recently, we improved the accuracy of the hydro module by including a full three-dimensional reconstruction technique (see a thorough introduction of this technique in [1]). With the fully three-dimensional reconstruction, the hydro module became the hotspot of the application. Here, we present and test its initial GPU implementation. Our radiation module, still in the testing phase, uses an explicit transport scheme with the reduced speed of light approximation, coupled to an implicit scheme for the radiation-hydro coupling terms, in a manner similar to Skinner et al. [2].

To validate the theoretical claim that the full three-dimensional reconstruction technique results in more accuracy, a rotating star simulation using the old and new hydro modules with the same gravity module were executed. The error and convergence of both methods is compared to validate the theoretical claim with numerical results, see Section VI. However, this paper focuses on the task-based execution using adaptive mesh refinement, resulting in some irregular parallelism. The task-based approach helps us with properly parallelizing the tree-traversals. As we strive for the lowest time per timestep possible, this in turn means we have to process millions of cells in sub-second runtimes. This means we have a task-graph of extremely short running compute kernels mixed with the communication and data transfers.

We are revisiting the performance of the gravity module and studying the performance of the new hydro module on ORNL's Summit. Octo-Tiger's scaling capabilities have been previously shown: NERSC's Cori [3] and on CSCS Piz Daint [4], however, in these measurements an older version of the hydro module was used. We have experience running Octo-Tiger and the C++ standard library for parallelism and concurrency (HPX) [5] on x86 systems and CRAY based systems, but not much previous experience with distributed runs on IBM® Power9™ systems.

First, the hydro module for the Sedov-Taylor blast wave is studied. Second, a rotating star for the combination of the hydro and gravity module is simulated. For both problems, we show the node level scaling for CPU and CPU+GPU runs on a single node. Note that due to the different implementations of the hydro kernels, especially the more computationally intense reconstruction of the fluxes in the new implementation, we can not directly compare the scaling results.

In addition, analyzing such large task graphs can be rather challenging, see Figure 1. This is the first time we employ APEX with CUDA support to get combined profiling of the CPU and GPU tasks. CPU-only profiling with APEX has been shown in [6].

The paper is structured as follows: Section II covers the related work. Section III sketches the software framework. Section IV introduces Octo-Tiger's new hydro module and its GPU acceleration. Section V shows the node level and distributed scaling of Octo-Tiger on Summit. Section VI compares the accuracy of the new three-dimensional full reconstruct of the hydro kernel with the previous kernel. Finally, Section VII concludes the paper.

## II. Related work

There are many astrophysics codes which combine hydrodynamic and gravity solvers for the simulation of astrophysical fluids. Here, however, we are focusing on those which have two additional properties that Octo-Tiger has: *1)* They are

2168-9253/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00059

accelerated by an asynchronous many-task system (AMT) and *2)* They use adaptive grid refinement. ChaNGa (Charm N-body Gravity solver) [7] performs collisionless N-body simulations for cosmological simulations or simulations of isolated stellar systems. A moving-mesh hydrodynamic solver was added to ChaNGa [8] together with the implementation of multiple time-steps techniques to form the code MANGA [9], suitable for simulating interacting binary stars. Enzo-E / Cello (formerly Enzo-P) [10], which is currently under active development, is designed for astrophysics simulations, including star formation and cosmology applications. Cello provides the AMR feature within Enzo-E. Both of these codes use the AMT Charm++ [11]. Another AMR-based code is Castro [12], part of the AMReX Astrophysics suite utilizing the more traditional MPI+X approach. The Athena++ code, a C++ rewrite of the magneto-hydrodynamic code Athena C, implements an adaptive mesh refinement and uses MPI+OpenMP for its parallelization [13]. A GPU-accelerated version of Athena++, K-Athena, was refactored using Kokkos to achieve better performance and portability [14]. All these codes attempt to exploit high abstraction programming for the parallelization of their code to display scaling on exascale supercomputers. For example, Charm++ and the AMT used by Octo-Tiger, HPX, have very similar programming models. From an application developer perspective, HPX can be seen as an abstraction to C++ and Charm++ more as a standalone library [15]. According to this survey [16] HPX has the highest technical readiness. Two of the codes, K-Athena and Castro, have recently reported their scaling and performance on OLCF's Summit [14], [17]. We aim to report Octo-Tiger's performance on Summit as well, in particular after upgrading the hydro solver and porting it to GPU CUDA kernels. Since two of the main functionalities of the code, the gravity and hydro solvers, can be executed on GPUs, it is interesting to study the scaling on numerous GPUs. Although a direct comparison between the performance of codes is not trivial, a simple basic measurement of interest is the number of cells (zones) updated per second (or per microseconds). Castro reported a value of 130 zones/$\mu$seconds on one Summit node [17], while K-Athena reported a peak value of $> 100$ zones/$\mu$seconds [14].

## III. Software framework

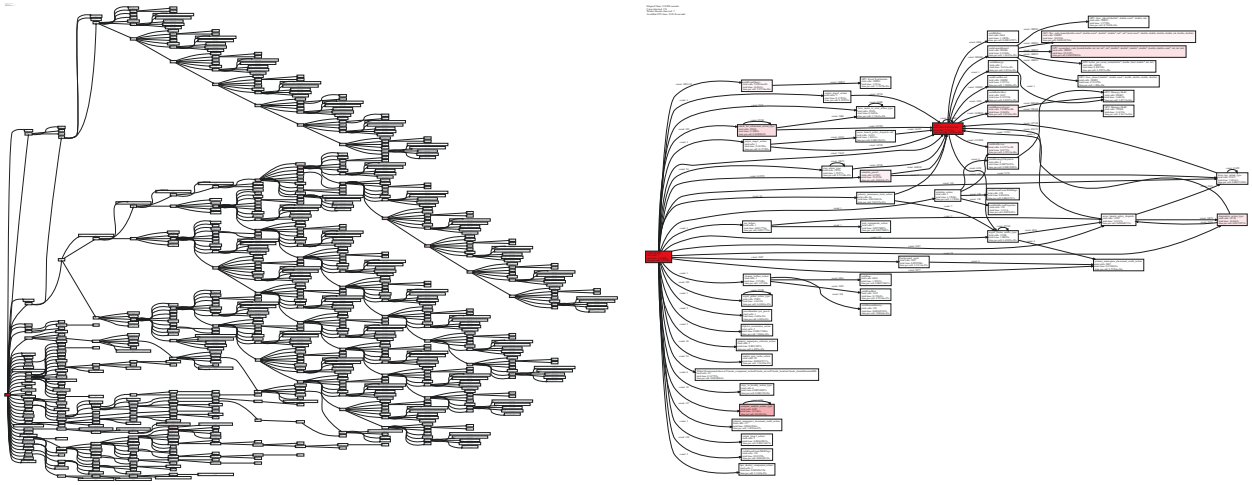
### A. C++ standard library for parallelism and concurrency

HPX is the C++ standard library for parallelism and concurrency. It exposes an API that fully conforms to the recent C++ standards [18] on top of an asynchronous many-task runtime system (AMT). It has been described in detail in other publications, such as [5], [19]–[21]. In the context of this paper, HPX has been used for two purposes. a) to coordinate the asynchronous execution of a multitude of heterogeneous tasks (both on CPUs and GPUs), thus managing local and distributed parallelism while observing all necessary data dependencies, and b) as the parallelization infrastructure for executing CUDA-kernels on the CPUs via the asynchronous HPX backend.

### B. APEX

APEX [22] is a performance measurement library for distributed, asynchronous multitasking systems. It provides lightweight measurements without perturbing high concurrency through synchronous and asynchronous interfaces. To support performance measurement in systems that employ user-level threading, APEX uses a dependency chain in addition to the call stack to produce traces and task dependency graphs. The synchronous APEX instrumentation application programming interface (API) can be used to add instrumentation to a given run time and includes support for timers and counters. The NVIDIA CUDA Profiling Tools Interface [23] provides CUDA host callback and device activity measurements. Additionally, the hardware and operating system are monitored through an asynchronous measurement that involves the periodic or on-demand interrogation of the operating system, hardware states, or runtime states (e.g., CPU use, resident set size, memory "high water mark"). The NVIDIA Management Library interface [24] provides periodic CUDA device monitoring to APEX. In previous work [25], APEX was extended to capture additional timers and counters related to CUDA device-to-device memory transfers, as well as tracking memory consumption on both device and host when requested with the `cudaMalloc*` API calls.

Tracing measurement is typically used by application developers to understand timing and dependency relationships between different tasks within an application. When tracing to the Open Trace Format (OTF2) or Google Trace Events Format, each concurrent CUDA Stream is assigned three virtual "threads" to track kernel, memory and synchronization activity. This is necessary because these three classes of events are not perfectly nested timers – there is a potential for asynchronous overlap – which are a requirement for the OTF2 tracing library (Google Trace Events are more forgiving). However, each operation class within a Stream does have a guaranteed ordering, so this segregation of event types is sufficient to meet the requirements of the tracing libraries and formats. However, because the Octo-Tiger CUDA implementation uses up to 128 concurrent streams per process (along with the actual HPX worker and helper threads on the CPU), even a relatively small run with 6 ranks per node can result in over 2400 unique "threads" of execution, and a collection of trace files over 27GB in size from just 25 iterations. To work around this issue of scale, APEX was extended to support *task dependency trees* to complement the existing task dependency graph support. The tree is a summary representation of the task dependency relationships (*task types, not individual tasks*), revealing the full dependency chain and not just immediate parent/child relationships. While this can result in tree representations that are larger than the graph representation – due to expanded recursions and continuations, see Figure 1b for an example – the trees are still quite manageable and helpful in diagnosing problems in programming models like HPX that *do not* have a meaningful callstack context but *do* have a task dependency context, including tasks and other activity



(a) Task tree example.

(b) Task graph example.

Fig. 1. Task tree and task graph of Octo-Tiger as captured by APEX. Intensity of red color is correlated with the node's contribution to the overall runtime. The recursive structure of the octree is evident in the expanded tree. High resolution images are available here (https://doi.org/10.6084/m9.figshare.14666184.v1).

offloaded to GPU devices. To complement the taskgraph and tasktree data in the absence of a full trace, APEX also captures task and counter scatterplot data, indicating on the $x$ axis when the task started or the counter was captured, and the $y$ axis contains the duration of the task or the value of the counter. The tasks are sampled using a user-specified fraction (default $1\%$) whereas the counters are sampled at every value capture. This data collection allows the application developer to capture a time sequence of data without the filesystem overhead of a full event trace. Examples are shown in Section V.

## IV. Octo-Tiger

In this section, we briefly introduce the modules of Octo-Tiger studied in this paper, followed by details on how we integrate their GPU implementation with HPX. For a more general overview of the modules themselves, we refer to Octo-Tiger's method paper [1].

### A. Octo-Tiger's Gravity Solver

Octo-Tiger uses a fast multipole method (FMM) for solving the gravity [26]. This particular implementation of the FMM globally conserves both linear and angular momenta to machine precision, and, when coupled to the hydro-dynamics solver, also globally conserves energy to machine precision. The solver uses a third order multipole expansion. Its accuracy can be varied by adjusting the opening criterion, $\theta$. Lower values of the opening criterion lead to stricter multipole acceptance criteria, requiring that multipoles be further away to interact. This increases the solver's accuracy at the cost of more computation time.

### B. Octo-Tiger's Hydro Implementation

Octo-Tiger solves the equations of hydrodynamics using a finite volume method. It evolves the mass density, three linear momenta, and gas energy on a rotating adaptive mesh refinement (AMR) mesh. The AMR mesh is based on an octree structure, with each node of the octree being either not refined at all or fully refined with eight sub-grids of twice the resolution as their parent. By default, each of those sub-grids consists of $8^3$ cells, however, this is adjustable at compile-time to allow for more finely refined sub-grids with more cells (for instance $16^3$). The evolved variables reside on the leaf sub-grids of the octree. It additionally evolves an entropy tracer, using it to implement the dual energy formalism of Bryan et al. [27]. First, the evolution variables are reconstructed from cell averages at 26 quadrature points on the cell face: the centers of cell faces and cell edges and at cell vertices. This is accomplished by applying the piece-wise parabolic method (PPM) of Colella et al. [28]. This third order, five cell stencil is applied along the lines between cell centers that coincide with particular quadrature points, producing left and right values for each. Octo-Tiger optionally allows for the contact discontinuity detection available with PPM. Once the evolution values are reconstructed, the semi-discrete central-upwind scheme of Kurganov et al. [29] is applied to the reconstructed left and right variables at the quadrature points, producing fluxes. These fluxes are summed at quadrature points on a given cell face using Newtonian quadrature to obtain the final flux. Octo-Tiger's complete hydro scheme is described by Marcello, Shiber, et al. [1]. In this paper, we compare our new hydro module to the old hydro module. The old hydro module used the same reconstruction method, however, flux values were only computed at the centers of cell faces.

### C. Octo-Tiger's CUDA Implementation

To understand Octo-Tiger's GPU implementation of the hydro module, it is worth reintroducing the GPU implementation of the gravity module from prior work. While the gravity module uses entirely different compute methods (which we

will only briefly mention here), it uses the same mechanism for combining HPX and CUDA to facilitate concurrent GPU kernel execution. The following subsection offers details how (and why) we use this mechanism, followed by the details of the hydro GPU implementation in the subsequent subsections.

*1) Gravity Module GPU Implementation:* The gravity solver—more specifically the calculation of the same-level interactions in the second FMM step—was the original hot spot within Octo-Tiger [30], [31]. Here, we have to calculate the cell-to-cell interactions for each of the cells of a sub-grid. The exact number of interactions per cell depends on the parameter $\theta$. The actual hot spot consisted of different methods (henceforth called gravity kernels) that take care of the various types of cell-to-cell interactions. All kernels operate on one sub-grid at a time, calculating all interactions between the cells within that sub-grid in addition to their interactions with cells in the ghost layer. The interaction types and the gravity kernels themselves are detailed in prior work in more detail [30].

As a sub-grid only contains $512$ cells by default, a gravity kernel responsible for calculating the interactions of a single sub-grid does not cause enough work to saturate a GPU. There are two ways to address this. As mentioned previously, the number of cells per sub-grid can be increased, which in turn would provide more work for each GPU kernel. However, this would be an Octo-Tiger specific solution. Instead, we were previously able to overcome this limitation for the gravity-solver GPU kernels by using a more general approach: A HPX-CUDA integration.

This integration allows for the execution of CUDA kernels to be integrated with the HPX runtime system via HPX futures. Essentially, after launching a CUDA kernel, HPX offers the functionality to return a HPX future for it. The HPX scheduler will then continue to poll a CUDA event that will be set as soon as said CUDA kernel is done. Once the event is set, the HPX future will be set to ready, which in turn triggers all tasks that depend on it. This allows us to integrate CUDA kernels into the HPX task graph.

We can thus handle CUDA kernels (and CPU/GPU data transfers) the same way as any other HPX task, making it easily possible to chain them with other tasks, such as arbitrary CPU compute tasks, inter-node communication, or I/O. Crucially, this means that the execution of a CUDA kernel gets automatically overlapped with other tasks, which includes the execution of other CUDA kernels on separate CUDA streams. This leads to the concurrent execution of multiple CUDA kernels on separate sub-grids, preventing GPU starvation despite the small workload with just $512$ workitems per kernel invocation.

As we launch each CUDA kernel within a normal HPX task, we can easily suspend the task until the GPU kernel is done (as indicated by its HPX future) and have an HPX worker thread pick up the original task afterwards to process its results. This allows a single worker thread to easily handle multiple CUDA streams, switching between HPX tasks. In previous work, we achieved a high GPU utilization and performance using this approach within the gravity solver [4]. There, we used $12$ worker threads (one for each CPU core) and $128$ CUDA streams for one P100 GPU.

For this approach, however, we need to keep any GPU-wide synchronization to a minimum. This includes calls to `cudaMalloc()` and the creation of CUDA streams. To avoid creating more CUDA streams than necessary, we pre-allocate them at the start of the simulation. We usually use a pool of $128$ HPX CUDA executors per device, each handling one CUDA stream. We further employ a GPU-buffer manager to avoid on-the-fly allocation of buffers as much as possible. If available, the manager reuses previously allocated but currently unused device buffers from previous kernel invocations. Only if none is available a new buffer will be created.

Both the HPX-CUDA integration (exposed with HPX futures) and the buffer manager (exposed by a set of allocators within the library CPPuddle) can now be used independent of Octo-Tiger, to allow a similar scheme of easy, task-based, concurrent GPU kernel execution in other applications. This also means we can also easily re-use this technique to port more of Octo-Tiger's solvers to the GPU.

Furthermore, if needed, this CUDA-HPX integration approach can be combined with the other approach mentioned to increase GPU utilization: Increasing the size of the sub-grids. This allows us to approach the issue both on the tasking level using the integration and on the data-structure level by using sub-grids with more cells.

*2) Initial Hydro Module GPU Implementation:* Between the GPU implementation of the gravity module and the changes moving from the old hydro (where flux values are only computed at the centers of cell faces) to the new one as outlined in Section IV-B, the hydro module becomes the new application hot spot. Hence, we have ported the relevant methods of the hydro solver to CUDA for this work. The two major hot spots within the solver are the `reconstruct` method and the `compute_fluxes` method (henceforth called hydro kernels). The reconstruct method reconstructs the evolution variables using the PPM method as mentioned in Section IV-B. In turn, the flux method takes care of computing the fluxes and the Newtonian quadrature to obtain the final flux.

Just as the kernel of the gravity solver, each hydro kernel operates on one sub-grid in each invocation. Therefore, we are facing the same challenge as for the gravity solver: One kernel invocation on its own is insufficient to prevent GPU starvation. We have therefore ported the hydro solver's methods into CUDA kernels in two steps: First, we have optimized the kernels to run efficiently on a GPU. We have removed any excessive branching within the method (to avoid warp divergence), we have flattened all required data structures into one-dimensional arrays of continuous memory and removed any remaining, unnecessary memory in-directions of the initial CPU implementation. Second, we have integrated the kernels into the HPX task graph as we did with the gravity kernel to facilitate concurrent GPU kernel execution and the overlap of data transfers.

*3) Next steps for the Hydro GPU Implementation:* While porting the hydro solver to CUDA resolves a major bottleneck

TABLE I
TOOLCHAIN AND OCTO-TIGER'S DEPENDENCIES.

| | | | |
|---|---|---|---|
| gcc | 8.1.1/9.1.0 | hwloc | 1.11.12 |
| spectrum-mpi | 10.3.1 | boost | 1.70.0 |
| cuda | 11.2.0 | jemalloc | 5.1.0 |
| hpx | 1.6.0 | silo | 4.10.2 |
| hdf5 | 1.8.12 | cppuddle | `d32e50b` |

TABLE II
ORNL'S SUMMIT HARDWARE INFORMATION

| | | | |
|---|---|---|---|
| GPUs | 6 NVIDIA® Volta™ V100 | CPU | 2 IBM® POWER9™ |
| OS | RHEL 7.4 | Kernel | 4.14.0 |
| Interconnect | | Mellanox® EDR 100G InfiniBand | |

TABLE III
SIMULATION DETAILS OF THE SEDOV-TAYLOR BLAST WAVE. NOTE THAT EACH CONFIGURATION HAS $16,777,216$ CELLS TO BE PROCESSED.

| Sub-Grid Size | Sub-Grid Count | Refinement level |
|---|---|---|
| $8^3$ | 32768 | 5 |
| $16^3$ | 4096 | 4 |
| $32^3$ | 512 | 3 |

within Octo-Tiger, the kernels themselves are still an initial implementation and thus not yet tuned to the maximum extent: We first need to evaluate whether the concurrent execution of the multiple GPU hydro kernels with several CUDA streams and HPX futures is sufficient for GPU utilization. While we had achieved good results with this approach within the gravity solver [4], the hydro kernels are less compute-intensive than the gravity kernels. Thus, we might reach the limits of this approach.

If we do, there are multiple ways to address the issue: The easiest way is to simply increase the size of the sub-grids, providing more work per kernel invocation, increasing the number of blocks in the CUDA launch configuration. This makes it both easier to utilize the entire device and to increase the likelihood of having multiple resident blocks per SM which increases occupancy and thus hides latency. Of course, a higher sub-grid size comes with the trade-off of decreased scalability as (given the same overall grid size) we have less sub-grids to distribute to the different compute nodes. A more sustainable method would be to combine the kernels of multiple sub-grids into one kernel. However, this kind of work aggregation is more tedious to implement and comes with several implementation challenges of its own.

Thus, the current state of the CUDA implementation in this work provides a good starting point to evaluate the performance, before moving forward to fine-tuning the kernels themselves. We have therefore enabled Octo-Tiger to be configured with larger sub-grid sizes at compilation time, and we will study its performance and scalability impact in the following sections. A significant performance impact of larger sub-grid sizes in the hydro kernels would be a strong indication that we should focus on further work-aggregation before any fine-tuning of the compute kernels themselves.

## V. PERFORMANCE MEASUREMENTS

In this section, we examine the scaling of Octo-Tiger on ORNL's Summit. Table I shows the toolchain that compiled Octo-Tiger. Table II lists the hardware information of ORNL's Summit. Note that we used 128 streams per V100. Disclaimer: Due to a testbed allocation on Summit, we had limited node hours, which limited the possible performance measurements. In addition, for jobs with more than 128 nodes we experienced some error from the IBM® Spectrum MPI on Summit that we send too many messages and a network device crashed, see IBM® ticket TS005902510. We therefore cannot show scaling results beyond 128 nodes. Strong scaling was used for all runs.

### A. Sedov-Taylor Blast Wave (Pure Hydro)

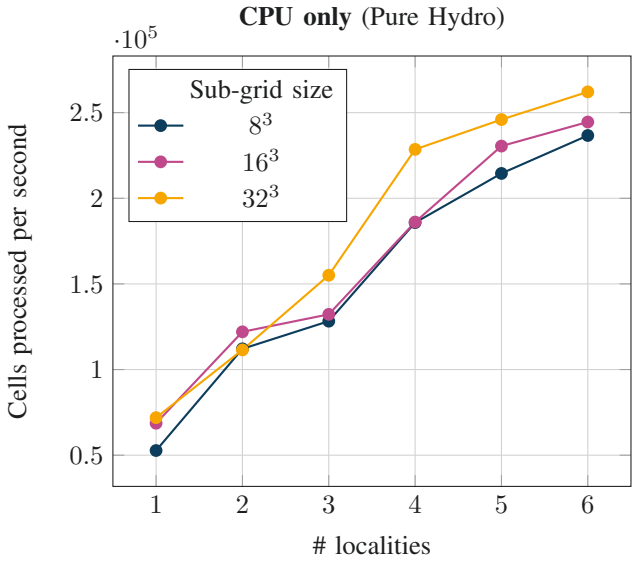
To benchmark the new hydro kernels, the Sedov-Taylor blast wave is used. Table III shows the details of each level of refinement.

*1) Node level scaling:* The scaling on one Summit node is presented in this section. Each configuration with an increasing sub-grid size, see Table III, is executed on a single node using CPUs and CPUs + GPUs. We start with one HPX locality, which is equivalent to one MPI process. Thus, using six HPX localities, we run six MPI processes on Summit. We chose this setup to enable easy multi-GPU usage, at the expense of more inter-process communication. For each HPX locality, we assigned seven CPU cores and none of the six GPUs. Figure 2a shows the scaling with the increasing number of localities.

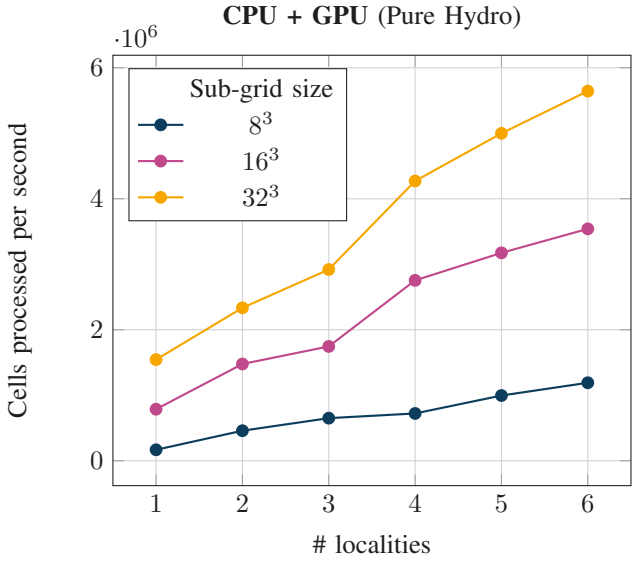
The CPU-only scaling for the sub-grid sizes of $8^3$ and $16^3$ behaves similarly, and the sub-grid size of $32^3$ performs better for three and more localities.

For the next run, one locality was assigned to seven CPU cores and one NVIDIA® V100 GPU. With six localities, all available CPU cores and GPUs on a single node are utilized. We assigned 128 CUDA streams to each locality. Note that for the sub-grid size of $32^3$ we had to decrease the number of streams for the run with one locality, since queuing too many large kernels caused the device to hit its memory limit.

Figure 2b shows the number of processed sub-grids per second. With increasing sub-grid size, the number of cells processed per second improves notably, even though the overall grid size stays the same (albeit consisting of fewer sub-grids). As mentioned in Section IV-C2, the hydro GPU kernels might not offer enough work to prevent GPU starvation, even with running multiple kernels (on separate sub-grids using separate CUDA streams) concurrently on the GPU. Increasing the sub-grid size increases the amount of work per kernel accordingly, making it easier to scale up to an entire GPU simply by having more blocks of work items available. Of course, it also increases the chance of having multiple blocks resident on one SM (we ensure that register usage is low enough for multiple blocks to be resident on one SM during the compilation time), increasing occupancy and thus hiding latencies more efficiently. The average runtime per `reconstruct` kernel is just 258 microseconds, or 108 microseconds for the `flux`




(a)




(b)

Fig. 2. Cells processed per second for the node level scaling. For one up to 6 localities on one Summit node. One locality was assigned to seven CPUs and one NVIDIA® V100 GPU.

kernel when using a sub-grid size of $8^3$, further highlighting this point. In the short term, we can offset this problem by using a larger sub-grid size. However, an explicit work aggregation scheme combining multiple sub-grids might be preferable as a long-term solution.

Overall, we get a reasonable speedup for using the GPUs given the initial state of our hydro implementation. For all sub-grid sizes, the processed number of sub-grids was one order of magnitude higher.

*2) Distributed scaling:* The scaling up to 128 Summit nodes using 768 NVIDIA® V100 GPUS and 5376 CPU cores is studied. Here, we use 6 localities with one GPU and 7 CPU cores per node. Figure 3a shows the number of sub-grids processed per second. Here, the sub-grid size of $16^3$ performs slightly better than the sub-grid size of $8^3$. For up to 8 nodes the sub-grid size of $32^3$ performs best, but later not enough work is available, and the scaling flattens out. Figure 3b shows the speedup with respect to a single node. For up to 8 nodes all sub-grid sizes perform similarly and the largest sub-grid size flattens out again. Up to 16 nodes the lower two sub-grid sized perform similar and later the smallest sub-grid size performs best.

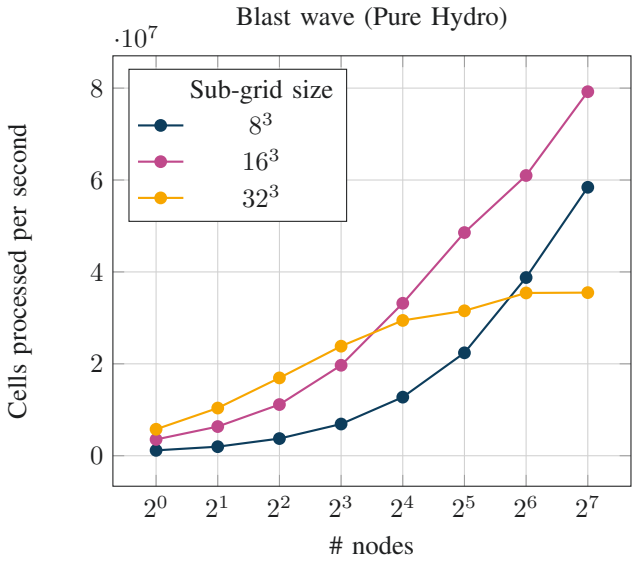
We need at least 7 sub-grids per locality (42 per node), as otherwise the 7 CPU cores are underutilized. While the majority of the work is done by the GPUs, there are pre-processing steps and the procedure of sending the data to the GPU and launching the kernels that are done purely by the CPU. Ideally, we have more sub-grids per locality, to truly benefit from the overlapping of computation, inter-locality communication and CPU/GPU data transfers that we gain by using the task-based functionality offered by HPX. Indeed, we can observe good scaling as long as we have about 21 sub-grids per locality, as we both have enough work for all cores and the GPU and benefit from the overlapping. The parallel efficiency degrades visibly when going below that threshold. First, we start losing the benefits of overlapping. Later on, we simply cannot use all CPU cores of a locality to do the pre-processing, kernel launches and communication tasks (as one core always works on one sub-grid). Lastly, we hit the point where we only have one sub-grid per locality. Here, we naturally do not benefit at all by adding more nodes.

We can see this in the runs with sub-grid size $32^3$. Here we go below 21 sub-grids per locality in-between 4 and 8 nodes (as we use 6 localities per node), afterwards we go below 7 sub-grids at 16 nodes. Lastly, we hit 1 sub-grid per locality at 64 nodes, so further increasing the node count to 128 makes no difference.

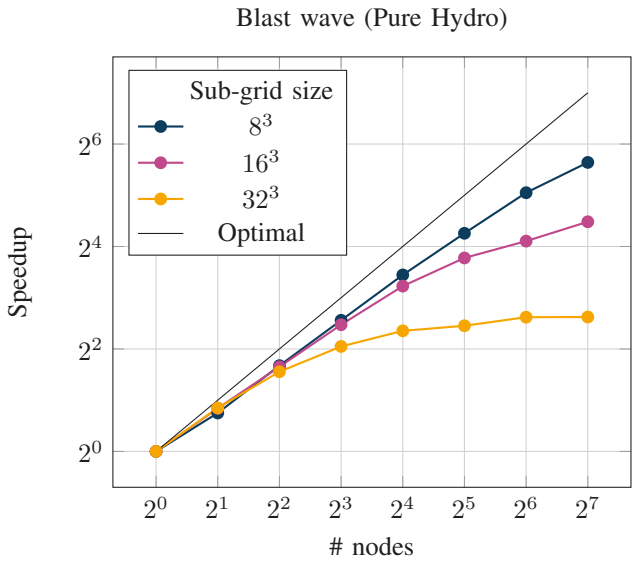
It is worth noting that the largest run with sub-grid size $8^3$ and 128 nodes results in a runtime per timestep of just 286ms, while with a sub-grid size of $16^3$ we get a runtime per timestep of 211ms. Considering each timestep consists of three consecutive iterations of the hydro solver (due to Octo-Tiger's use of a third-order Runge Kutta time integration scheme) this highlights that even small inefficiencies and barriers could cause significant slowdowns, simply due to the short runtimes involved.

### *B. Rotating star (Hydro and gravity)*

For the second example, the rotating star problem is studied, where the gravity solver is added to the hydro solver. Table IV shows the details for each level. Here, we use the default $\theta$ value (0.5) for the rotating star problem, which leads to fewer cell-to-cell interactions than we encounter with production run simulations. This makes the gravity solver less compute-intensive than it would typically be. Furthermore, we have redesigned the gravity GPU kernels to allow different (larger) stencil sizes, making them currently less finely tuned than they previously were, as the shared-memory implementation in the monopole-monopole gravity kernel assumed a fixed stencil




(a)




(b)

Fig. 3. Cells processed per second for the distributed scaling from one Summit node up to 128 Summit nodes. Note that all six NVIDIA® V100 GPUs per node were used.

TABLE IV
SIMULATION DETAILS OF THE ROTATING STAR. NOTE THAT EACH CONFIGURATION HAS 16, 777, 216 CELLS.

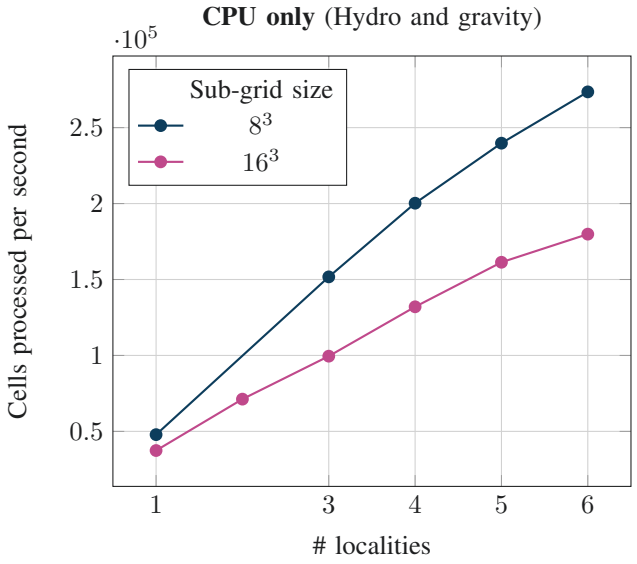
| Sub-Grid Size | Sub-Grid Count | AMR boundaries | Refinement level |
|---|---|---|---|
| $8^3$ | 44472 | 3800 | 8 |
| $16^3$ | 5944 | 3800 | 7 |

size. Still, the rotating star scenario presents a good benchmark as it allows us to test the hydro- and gravity solver together in a simple scenario.

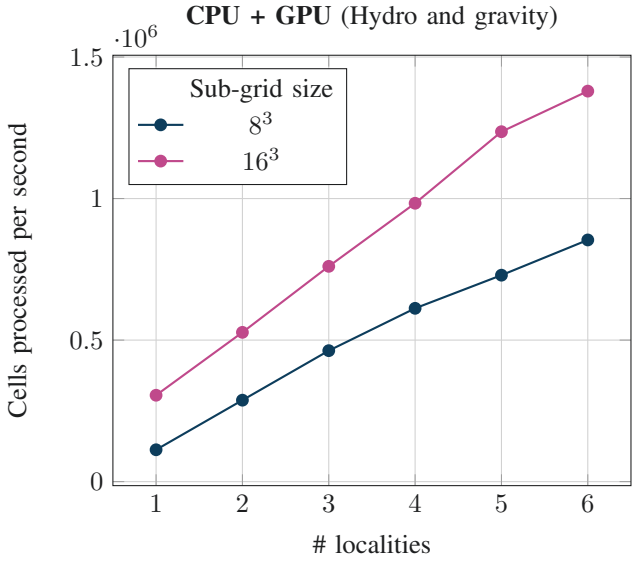
*1) Node level scaling:* The scaling on one Summit node is presented in this section. Each configuration with an increasing sub-grid size, see Table IV, is executed on a single node using CPUs and CPUs + GPUs. We start with one HPX locality, which is equivalent to one MPI process. Therefore, using six HPX localities, we run six MPI processes on Summit. For each HPX locality, we assigned seven CPU cores and none of the six GPUs. Figure 4a shows the node level scaling from one up to 6 localities for CPUs only. The smaller sub-grid sizes perform better using the CPUs only. We suspect that this is due to the gravity solver's handling of the root sub-grid within the octree: We have to process all same-level interactions within the sub-grid (as there is no higher level available that would take care of those interactions within the FMM algorithm). The runtime of calculating these interactions is $\mathcal{O}(N^2)$ with $N$ as the number of cells in the root sub-grid. In a CPU-only run, the root node is processed like any other sub-grid, meaning the same-level interactions are calculated within one HPX task; thus, only one CPU core is working on it, while all other cores take care of other tasks. This increases the runtime substantially while increasing the size of the sub-grids in particular, since the entire next top-down tree-traversals within the FMM algorithm depend on the results of the root sub-grid. With an increasing number of CPU cores, more of them will simply be idle whilst waiting on these results. When increasing the number of localities, the ratio of the root sub-grid's work to the work of the remaining sub-grids on the root locality increases, resulting in a higher load imbalance.

Figure 4b shows the node level scaling adding one GPU to each locality. In that case, the GPU kernels benefit of the larger sub-grid size and larger sub-grid sizes performs better. The issue with the root sub-grid is less severe here as the interactions are not being calculated by one CPU core alone, but instead by a GPU kernel. Between this improvement, and the general better runtime behavior of the hydro kernels when dealing with larger sub-grids, the performance improves when switching to a sub-grid size of $16^3$. However, the speedup is less severe than with the Sedov-Taylor blast wave scenario as the gravity GPU kernels do not seem to benefit from larger sub-grid sizes (even with the improved GPU kernel for the root sub-grid). Again, the processed sub-grids per second are one order of magnitude higher adding the GPUs.

*2) Distributed scaling:* We now study scaling on up to 128 Summit nodes using 768 NVIDIA® V100 GPUS and 5376 CPU cores. Here, we use 6 localities with one GPU and 7 CPU cores per node. Figure 5a shows the processed sub-grids per second for increasing number of nodes. Again, for the combined hydro and gravity simulation, the larger sub-grid sizes results in slightly better performance. Larger sub-grid sizes have less effect on the gravity solver and predominantly accelerate the hydro solver. Therefore, we observe a similar picture as for the hydro-only scenario. It is worth noting that the runtime per time step on 128 nodes for the sub-grid size $8^3$ is $\approx 0.48$ seconds, and for sub-grid size $16^3$ it is $0.45$ seconds. Note that for each time step, Octo-Tiger solves 3 hydro steps and 6 FMM steps (the gravitational potential as well as its time derivative appear in the source equations for the hydrodynamics). Here, the same argument is valid that we have good scaling as long as we have 21 sub-grids per

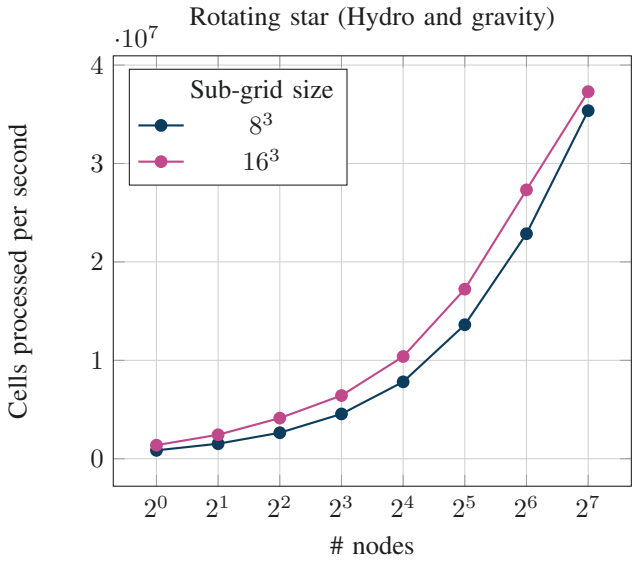



(a)

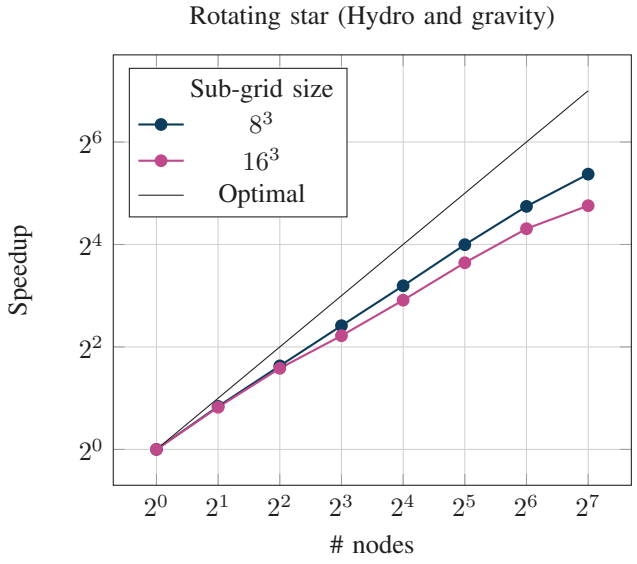



(b)

Fig. 4. Cells processed per second for the node level scaling. For one up to 6 localities on one Summit node. One locality was assigned to seven CPUs and one NVIDIA® V100 GPU.




(a)




(b)

Fig. 5. Cells processed per second for the distributed scaling from one Summit node up to 128 Summit nodes. Note that all six NVIDIA ® V100 GPUs per node were used.

locality. This indicates that approximately 16 million cells are not enough work for 768 GPUs.

### C. APEX + CUDA

The introduced overhead for the APEX CUDA measurements was about 30 seconds for the run on a full single node which is $\approx 8.5\%$ of the total execution time. This is slightly more than using APEX without the CUDA counters where the overhead was around one percent [6]. This overhead is likely caused by excessive callback processing for some frequently called but short-lived CUDA functions. In fact, because the algorithms support the ability for each locality to schedule work on more than one GPU, the profiling showed that the function `cudaSetDevice` is called over $4,322,208$ times during a 332 second run. In addition, HPX uses polling to detect GPU activity completion instead of callbacks — polling provides faster throughput — and performing the polling requires $3,056,145$ calls to `cudaEventQuery`. These frequent, short calls are fine on their own, but there is an observed overhead in measuring them.

Figure 6 shows the time spent in the sampled tasks during a short execution of the rotating star problem. The gravity (monopole/multipole interactions) and hydro (flux_cuda_kernel, reconstruct_cuda_kernel) kernels execute on the GPU, whereas other actions are executed on the CPU. The validation routine (compare_analytic_action_type)

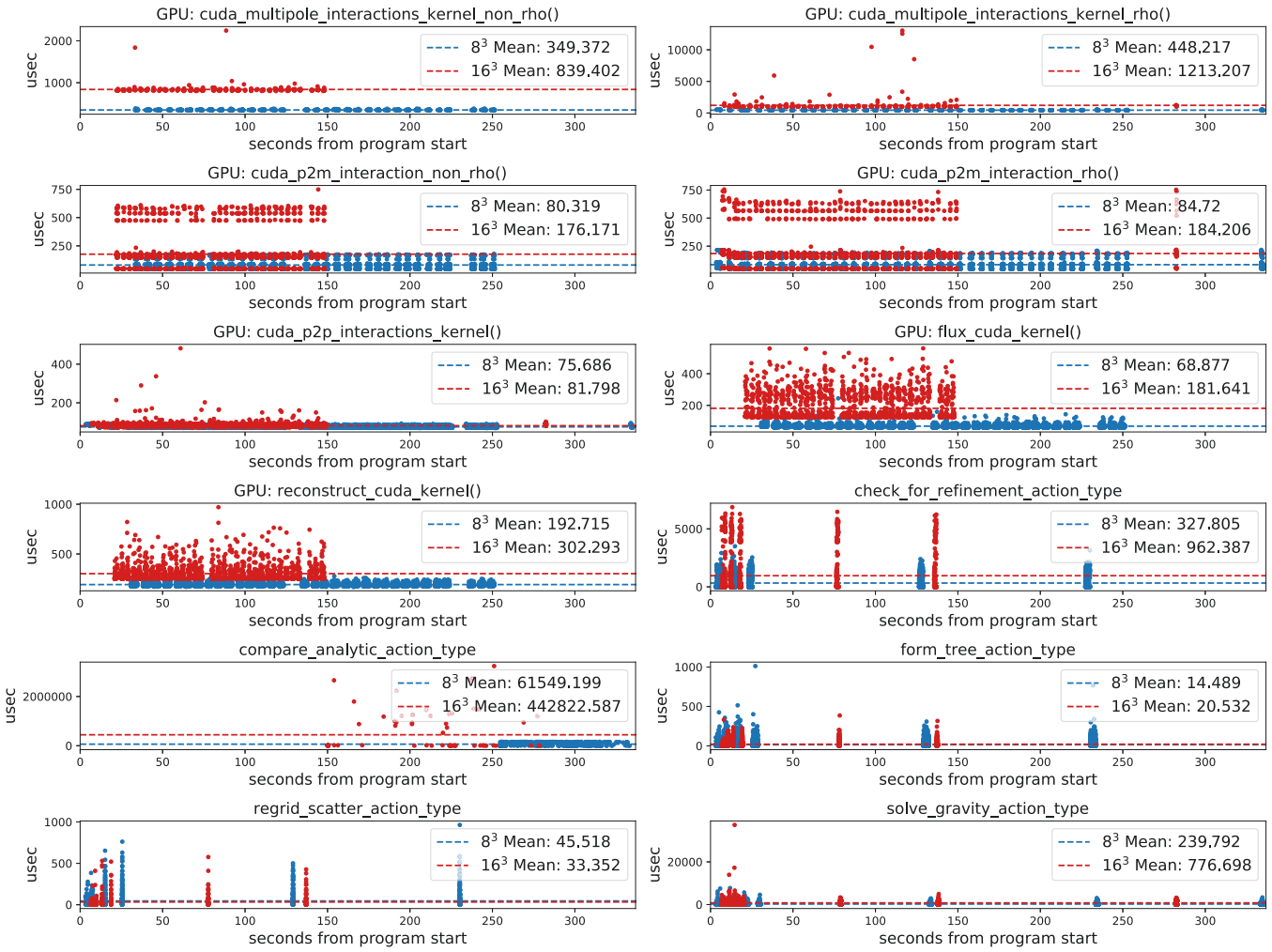



Fig. 6. GPU kernel activity and CPU task actions for the gravity and hydro tasks when executing the $8^3$ and $16^3$ rotating star test on 6 localities. The $16^3$ decomposition leads to longer-running tasks and kernels, but a shorter overall execution time because there are significantly fewer of them.

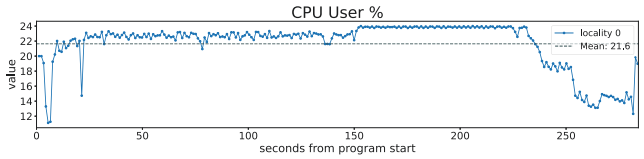
is executed on the CPU only. As this routine is only used for validating the results, it is unlikely to be ported to the GPU.

Figure 7 shows three counters captured during the rotating star run that indicate utilization of the allocated hardware. The CPU user-space utilization in Figure 7a is captured by monitoring the `/proc/stat` virtual file. Although HPX has launched 1 worker thread per physical core, the operating system detects 4 hardware threads per core. Therefore, the maximum utilization possible in this configuration is 25%. During the CPU-intensive validation at the end of execution, these threads are fully utilized, and during most of the execution the threads are well utilized. Time spent processing system calls (not shown) peaks at 3% during initialization and finalization and otherwise averages 0.66%. The GPU utilization data is captured by periodically capturing the available NVML data for device 0. Finally, Figure 7c shows the total memory allocated on the device through `cudaMalloc*()` calls, which peaks out at less than 11% of available memory. The GPU utilization and memory usage show that there is plenty of resources available to increase the amount of work per kernel and retain more data on the GPU.

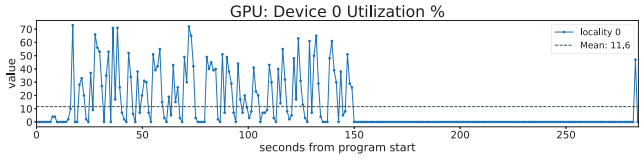
## VI. Astrophysical Test Results

To verify that Octo-Tiger's new hydro module delivers better results for an equilibrium configuration, we ran a rotating star test problem. This star was constructed using a polytropic structural equation of state with the self-consistent field method (SCF) [32]. It is uniformly rotating about its $z$-axis at a rate sufficient to produce a star whose minor axis is $3/4$ the length of its major axis. We ran this problem for ten dynamical times. Since the star begins in equilibrium, we expect it to stay in equilibrium. We used two resolutions and for each resolution, two choices for the opening criterion, $\theta$. (Lower $\theta$'s result in a larger multi-pole interaction stencil for the gravity solver and hence better results). Here we define the density error as

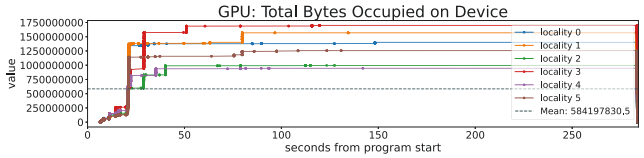
$$\rho_{L1} := \frac{\sum_{\Omega}(\rho_{IC} - \rho)\Delta^3}{V}, \tag{1}$$




(a) CPU Utilization of the 168 total available hardware threads. Data captured by locality 0 represents the aggregate utilization of all processes on the node.




(b) GPU Utilization for Device 0 used by locality 0.




(c) Total memory occupied through explicit allocations on each GPU.

Fig. 7. APEX Performance counter metrics from the $16^3$ rotating star test case run on 6 localities.

TABLE V
THE AVERAGE ERROR IN THE DENSITY FIELD FOR THE ROTATING STAR TEST USING THE OLD AND NEW HYDRO MODULES. IN THESE UNITS, THE CENTRAL DENSITY OF THE STAR IS 1.

| Refinement Level | Opening Criterion | Old | New |
|---|---|---|---|
| 6 | 0.5 | $2.41 \times 10^{-3}$ | $1.45 \times 10^{-3}$ |
| 6 | 0.35 | $5.22 \times 10^{-4}$ | $3.59 \times 10^{-4}$ |
| 7 | 0.5 | $2.52 \times 10^{-3}$ | $1.51 \times 10^{-3}$ |
| 7 | 0.35 | $4.49 \times 10^{-4}$ | $2.78 \times 10^{-4}$ |

where $\rho$ is the numerical mass density, $\rho_{IC}$ is the mass density from the initial conditions, $\Delta$ is a cell width, $V$ is the initial volume of the star, and the summation is over the entire domain $\Omega$. As shown in Table V, in all cases the new hydro module delivers a lower error.

## VII. CONCLUSION

This paper showed the following aspects in evaluating Octo-Tiger's performance on Summit. First, from the astrophysical aspect, the new implementation of the hydro kernel using a fully three-dimensional reconstruction of the fluxes is more computationally expensive than the old kernel. However, the new hydro kernel evolves an equilibrium rotating star with greater accuracy than the old kernel.

Second, the scaling on Summit showed the following two things. First, on a single node, the usage of the GPUs improved the cells processed per second by an order of magnitude. Thus, Octo-Tiger benefits from the usage of GPUs for the hydro, and combined hydro and gravity simulations. Second, the distributed scaling up to 128 nodes using 768 NVIDIA® V100 GPUS and 5376 CPU cores was presented. Both test problems scaled up to 128 nodes for the two lower sub-grid sizes. However, we have seen that a problem containing $16,777,216$ cells starts to flatten out up to 128 nodes and indicates that even larger problems are necessary to provide enough work for the additional GPUs. With our testbed allocation on Summit, we could only show preliminary scaling results; however, we will continue to work to get the larger node counts running.

Third, the variation of sub-grid sizes was added to Octo-Tiger and this work studied the performance impact for the first time. For the hydro module on a single node, the sub-grid size of $32^3$ showed the best performance for the combined CPU and GPU runs, since with the larger sub-grid size more work was available for a single kernel run. However, for the distributed runs, only up to 8 nodes the largest sub-grid size gave the best performance. For the combined hydro and gravity simulation, the sub-grid size of $16^3$ gives slightly better performance. This indicates that this sub-grid size will be the default for production runs.

Finally, the APEX CUDA profiling provides combined task trees and task graphs for the work on the GPU and CPU. Previously, Octo-Tiger was run first to profile the CPU usage with APEX and a second time with NVIDIA®profiler. The new plots provide some insights into the asynchronicity of HPX and the dependency of tasks. The scatter plots showed that the memory usage on the GPU was small, since only the data to be computed are kept in the device memory. In addition, we could show a good utilization of the CUDA devices on a single node. These plots provide a good base to analyze the combined asynchronous tasks on the CPU and GPU and support our efforts to optimize the concurrent CPU and GPU tasks.

### A. Future Work

The results of this work motivate further improvements of the hydro solver's GPU implementation. We plan to investigate on-the-fly work aggregation across sub-grids to combine the benefits of larger GPU kernels to saturate GPUs with the increased scalability that smaller sub-grids offer.

Furthermore, after recent promising results using HPX and Kokkos together within the gravity solver, we plan to port the current hydro CUDA implementation to Kokkos [33] as well. The HPX Kokkos integration works similarly as the CUDA one, and transforming the hydro CPU methods into GPU Kokkos kernels would have required the same changes to the methods themselves as outlined in Section IV-C2. Hence, as of the current state, we have already completed the first important steps.

Using Kokkos rather than pure CUDA provides us with two advantages: We can easily target GPUs of other vendors, such as AMD GPUs (and with the recently introduced Kokkos SYCL execution space, also Intel GPUs). Furthermore, Kokkos provides the means of using explicit SIMD vectorization [34] to run GPU-capable kernels efficiently on the CPU as well. Currently, we have to maintain a second set of CPU kernels using Vc for SIMD vectorization, which would be replaced by the Kokkos kernels. With a portable Kokkos implementation, there would be no need to maintain

two specialized CPU and GPU kernels to cover all platforms anymore. APEX already supports Kokkos profiling.

Furthermore, we plan to optimize the hydro kernels for shared memory usage as soon as they have been ported to Kokkos. With respect to HPX, more debugging is needed for jobs with larger node counts ($\geq$ 128 nodes): We have experienced stalls for higher node counts due to an error from the IBM® Spectrum MPI on Summit possibly caused by sending too many messages which result in a network device crash, see IBM® ticket TS005902510.

From the application perspective, the authors would like to compare the performance of the rotating star with the Castro code to gain insight into whether the more accurate hydro module results in more stable shapes of the star. However, a comparison of the scaling is not trivial since different algorithms and solvers are used in both codes. In addition, Octo-Tiger utilizes asynchronous computation with HPX, which CASTRO does not, as it uses MPI+X. Next, these scaling results are the preparation for large production runs on GPU accelerated supercomputers.

## ACKNOWLEDGMENT

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Diehl and Marcello thank the LSU Center of Computation & Technology for supporting this work. The APEX work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) under contract DE-SC0021299.

## SUPPLEMENTARY MATERIALS

The scripts to compile Octo-Tiger are available on GitHub and the script to run the jobs and the input files on Zenodo, respectively. CPPuddle is available here.

## REFERENCES

[1] D. C. Marcello *et al.*, "Octo-tiger: a new, 3d hydrodynamic code for stellar mergers that uses hpx parallelisation," *Monthly Notices of the Royal Astronomical Society*, 2021.

[2] M. A. Skinner *et al.*, "A Two-moment Radiation Hydrodynamics Module in Athena Using a Time-explicit Godunov Method," ApJS, vol. 206, no. 2, p. 21, Jun. 2013.

[3] T. Heller *et al.*, "Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 699–715, 2019.

[4] G. Daiß *et al.*, "From piz daint to the stars: Simulation of stellar mergers using high-level abstractions," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2019, pp. 1–37.

[5] H. Kaiser *et al.*, "HPX - The C++ Standard Library for Parallelism and Concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020.

[6] P. Diehl *et al.*, "Performance measurements within asynchronous task-based runtime systems: A double white dwarf merger as an application," *Computing in Science & Engineering*, 2021.

[7] P. Jetley *et al.*, "Massively parallel cosmological simulations with changa," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–12.

[8] P. Chang *et al.*, "A moving-mesh hydrodynamic solver for ChaNGa," *mnras*, vol. 471, no. 3, pp. 3577–3589, Nov. 2017.

[9] L. J. Prust *et al.*, "Common envelope evolution on a moving mesh," *mnras*, vol. 486, no. 4, pp. 5809–5818, Jul. 2019.

[10] J. Bordner *et al.*, "Enzo-p / cello: Scalable adaptive mesh refinement for astrophysics and cosmology," in *Proceedings of the Extreme Scaling Workshop*, ser. BW-XSEDE '12. USA: University of Illinois at Urbana-Champaign, 2012.

[11] L. V. Kale *et al.*, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 91–108.

[12] A. Almgren *et al.*, "Castro: A massively parallel compressible astrophysics simulation code," *Journal of Open Source Software*, vol. 5, no. 54, p. 2513, 2020.

[13] J. M. Stone *et al.*, "The Athena++ Adaptive Mesh Refinement Framework: Design and Magnetohydrodynamic Solvers," *apjs*, vol. 249, no. 1, p. 4, Jul. 2020.

[14] P. Grete *et al.*, "K-Athena: a performance portable structured grid finite volume magnetohydrodynamics code," *arXiv e-prints*, p. arXiv:1905.04341, May 2019.

[15] I. P. Demeshko *et al.*, "TBAA20: Task-Based Algorithms and Applications," Los Alamos National Laboratory, Tech. Rep. LA-UR-21-20928, 2021.

[16] P. Thoman *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018.

[17] M. P. Katz *et al.*, "Preparing Nuclear Astrophysics for Exascale," *arXiv e-prints*, p. arXiv:2007.05218, Jul. 2020.

[18] The C++ Standards Committee, "ISO International Standard ISO/IEC 14882:2020, Programming Language C++," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2020.

[19] H. Kaiser *et al.*, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. ACM, 2014, pp. 6:1–6:11.

[20] ——, "Higher-level parallelization for local and distributed asynchronous task-based programming," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM '15. New York, NY, USA: ACM, 2015, pp. 29–37.

[21] T. Heller *et al.*, "Closing the Performance Gap with Modern C++," in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Taufer *et al.*, Eds., vol. 9945. Springer International Publishing, 2016, pp. 18–31.

[22] K. A. Huck *et al.*, "An autonomic performance environment for exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.

[23] NVIDIA, "Cuda profiling tools interface," 2020, https://docs.nvidia.com/cuda/cupti/index.html.

[24] ——, "Nvidia management library (nvml)," 2020, https://developer.nvidia.com/nvidia-management-library-nvml.

[25] W. Wei *et al.*, "Memory reduction using a ring abstraction over gpu rdma for distributed quantum monte carlo solver," 2021.

[26] D. C. Marcello, "A Very Fast and Angular Momentum Conserving Tree Code," AJ, vol. 154, no. 3, p. 92, Sep. 2017.

[27] G. L. Bryan *et al.*, "A piecewise parabolic method for cosmological hydrodynamics," *Computer Physics Communications*, vol. 89, no. 1-3, pp. 149–168, Aug. 1995.

[28] P. Colella *et al.*, "The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations," *Journal of Computational Physics*, vol. 54, pp. 174–201, Sep. 1984.

[29] A. Kurganov *et al.*, "Semidiscrete central-upwind schemes for hyperbolic conservation laws and hamilton–jacobi equations," *J. Comput. Phys. SIAM J. Sci. Comput*, vol. 23, pp. 707–740, 01 2000.

[30] D. Pfander *et al.*, "Accelerating Octo-Tiger: Stellar mergers on Intel Knights Landing with HPX," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: ACM, 2018, pp. 19:1–19:8.

[31] G. Daiß, "Octo-tiger: Binary star systems with hpx on nvidia p100," Master thesis, Universität Stuttgart, May 2018.

[32] I. Hachisu, "A Versatile Method for Obtaining Structures of Rapidly Rotating Stars," ApJS, vol. 61, p. 479, Jul. 1986.

[33] H. C. Edwards *et al.*, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[34] D. Sahasrabudhe *et al.*, "A portable simd primitive using kokkos for heterogeneous architectures," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2019, pp. 140–163.