ExaHyPE's OpenMP GPGPU Port—Lessons Learned

Tobias Weinzierl *

July 2020

This experience report summarises impressions from work conducted under the umbrella of EPSRC's Excalibur programme under grant number EP/V00154X/1 (ExaClaw) and EP/V001523/1 (Massively Parallel SPH for Engineering and Astrophysics at the Exa-Scale).

Development Team
Jonathan Frawley (a), Gonzalo Brito Gadeschi (b), Peter Noble (c), Philipp Samfass (d), Daniele Tartarini (e), Karakasis Vasileios (f), Tobias Weinzierl (c)
(a) Advanced Research Computing (ARC), Durham University
(b) NVIDIA
(c) Computer Science, Durham University
(d) Informatics, Technische Universität München
(e) Department of Computer Science, University of Sheffield
(f) Swiss National Supercomputing Centre

All development has been done with GCC 10.1 built with NVIDIA GPU offloading support, with CLANG 10.0.0 and with XLC++ 17.1.1. The latter was used in the GNU reusage mode (CLANG front-end). Most tools have been used within experimental testbeds (some of them not even accepted), i.e. some claims on weaknesses might be due to immature installations or a lack of experience on our side. We also acknowledge that newer tool generations might already eliminate some of the problems we did encounter. Two GCC bugs (96833 and 96835) have been filed due to this report.

1 Introduction and background

ExaHyPE is an Exascale Hyperbolic PDE Engine, i.e. an open source code base that allows users to simulate wave phenomena described by first-order hyperbolic partial differential equations (PDEs) [2]. Different to other frameworks or solvers from the field, it is application-generic yet numerics-specific: The engine prescribes the numerical scheme, the spatial discretisation, and parallelisation. The user buys into this set of ingredients and tailors the solver towards her particular application by injecting domain functions. These are flux functions, eigenvalue evaluations or non-conservative products, e.g. [6].

ExaHyPE is based upon Peano which is a C++ code to administer and traverse adaptive Cartesian meshes [9]. The C++ code is instructed via ExaHyPE's Python interface what kind of data structures are hosted within the computational mesh spanning the computational domain. Per time-step, this C++ core runs through the data and invokes application-generic

*This document is part of a series of workshop reports that I host at `http://www.peano-framework.org/index.php/workshops`.

compute kernels from the ExaHyPE core. These compute kernels again call the user's functions. The user's main loop configures all three layers (user code, ExaHyPE core, Peano) and then hands over the responsibility to run a time step to Peano. Peano calls back ExaHyPE for numerics-specific routines (compute kernels), and ExaHyPE calls back the user layer for PDE-specific terms. This callback pattern is described in detail in [8].

ExaHyPE has originally been written by a consortium sponsored by a FET HPC H2020 grant using Peano's third code generation plus several different software packages and languages. This first version of ExaHyPE has focused mainly on Intel architectures with a strong investment into vectorised kernels to facilitate the efficient usage of Intel's manycore (KNL) platform. Today, we are in the process of writing a second generation of ExaHyPE which teams up with a complete rewrite of the Peano core. We will end up with a tandem of ExaHyPE 2 plus Peano 4. This endeavour is sponsored—among other funding streams—by EPSRC's ExCALIBUR programme under the ExaClaw project. One of the promises behind the code rewrite is to bring in GPU support without harming the ExaHyPE philosophy to hide numerics, meshing and optimisation details wherever possible, i.e. without bothering the user (too much) with the technical details behind GPUs. ExaHyPE's vision is to allow users to focus on domain challenges (PDE terms) only even though the code features very advanced numerics and massive upscaling.

While research software engineers at Durham had started to port ExaHyPE 2 application prototypes to GPUs early throughout the summer 2020, the port to GPUs really took off as a few core developers joined an 2020 NVIDIA hackathon in Sheffield. Throughout this hackathon, the whole engine has been made GPU-ready. Despite the fact that a detailed performance analysis and the application of developed techniques to real-world PDEs has not been achieved and remains outstanding due to the ambitious time constraints, this document can already summarise some lessons learned that are tied to "how to port our engine".

The document is a strongly biased summary and by no means a fair, comprehensive evaluation. It also exclusively focuses on challenges around ExaHyPE. However, it identifies key challenges many other projects might encounter, too, and it documents some state-of-the-art hiccups that we encountered within the development environment that is available at the moment—a situation that might change quickly. Finally, the experience report focuses on OpenMP GPU programming only—for reasons we both discuss and question.

The document is structured as follows: We first sketch the algorithmic landscape we operate in (Section 2) and identify kernels that fit to accelerators as well as their context, i.e. further operations that run concurrently. In Section 3, we describe the arising software from a user perspective and consequently elaborate why OpenMP seemed to be an interesting pathway towards GPU support to investigate. The major part of this document discusses various technical lessons learned. Both contextual sections before that can be skipped for readers interested in technical specifics, but they help to understand why certain options have been tried out and others haven't even been considered. We eventually use Section 5 to revisit the code architecture philosophy and decisions we followed throughout our porting.

2 The algorithmic landscape

Algorithmic mindset Peano's computational domains are discretised by adaptive Cartesian meshes as they result from spacetrees [8]. While the code base supports multiscale meshes, i.e. multiple resolutions overlapping each other, we use a plain view. The ExaHyPE project orbits around high order schemes—in particular ADER-DG—which are combined with standard Finite Volume (FV) discretisations. The latter serve as a posteriori limiter and are used only in subregions of the computational domain where stability constraints require it. In the present study, we focus on this Finite Volume scheme and use patch-based SAMR realisation [5]: Each cell of the Cartesian mesh hosts a $n \times n \times n$ patch of Finite Volumes.

To keep things simple, we merely study the compressible Euler equations with a plain Rusanov solver on these patches. Previous studies have shown that the Finite Volume limiter quickly becomes the most time-consuming step within a typical ExaHyPE run [3]. It is thus convenient to focus on this numerical building block.

Peano relies on a two-layer parallelisation: First, the computational domain is split up into chunks. We employ space-filling curve (SFC) cuts [1] to the cells and obtain a non-overlapping domain decomposition. These are standard techniques. The phrase "standard" circumscribes the following properties: The domain is cut into partitions, and each partition is surrounded by a (tiny) halo layer. As we work with Finite Volume patches with Rusanov, only one layer of Finite Volumes is required. Due to this layer, each partition can be processed totally independently. After a time step, the halo layers are exchanged between all partitions. The data flow scheme is similar to a Jacobi iterative solver.

On the second level, each partition is split further: We call the cells which are adjacent to a partition boundary or resolution change *skeleton cells*. All other cells are *enclave cells* [4]. As each cell hosts a patch, we obtain skeleton patches and enclave patches. A partition traversal, i.e. time step, corresponds to a traversal of all of these cells. As each cell's patch is surrounded by its own small FV halo layer—we actually surround the patches with a halo layer and thus automatically get the aforementioned halo layer around a partition—there's no particular constraint on the traversal order. We can process all cell patches concurrently. However, skeleton cells feed into resolution mapping routines or partition boundary data exchange activities. We therefore first process all skeleton cells non-blockingly, then trigger all data exchanges, and finally expect all enclave cells' operations to terminate.

Host realisation The partitioning through SFC cuts is mapped onto large tasks. Each rank hosts several SFC partitions (subdomains) and thus hosts a set of traversal tasks. The count is a fraction of the actual core count, i.e. we may decide to hold around $1 \leq k \leq c$ local partitions per rank. c is the number of available physical cores per rank. The traversal tasks run concurrently through their partition as they are mapped onto plain OpenMP tasks, i.e. each rank issues k OpenMP tasks within a task group. A traversal task runs through its partition and calls the compute kernel per skeleton cell directly. No further tasking is used here. Once the task group hosting the subdomain traversals has terminated, we exchange all data between partitions serially—exchanges are plain data copies for partitions which reside on one rank, otherwise we need MPI—before we wait for the enclaves to terminate:

Whenever a traversal task hits an enclave cell, this cell's compute kernel is not launched. Instead, the traversal task enqueues a request to process the cell into a rank-global job queue. Parallel to the traversal task, we issue $\hat{k}$ so-called *consumer tasks*. They are plain OpenMP tasks as well (though not tied to the aforementioned task groups) which listen to the queue of enclave tasks, pick the jobs from there and issue the kernels parallel to the actual traversal. These consumers run with reduced OpenMP task priority. We summarise that both enclave and skeleton cells define tasks. However, skeleton tasks are not spawned as tasks but processed serially (though by mesh traversals running concurrently with other traversals), while enclave tasks are modelled as tasks yet pushed into a queue from where they are eventually consumed.

We obtain a totally overlapping MPI+X parallelisation: Each rank hosts threads which run through their domain. While they do so and eventually exchange their boundary information, they spawn additional (background) tasks. These are processed with low priority and are supposed to backfill idle cores in algorithm phases with low compute workload or when a rank is busy with data exchanges.

GPU vision Enclave tasks have low logical priority as their algorithmic latency is not critical: We need the skeleton tasks to finish as soon as possible. This enables us to issue the data exchanges between partitions as soon as possible. Enclave tasks are issued together with the skeleton compute kernels, but they can run later.

Enclave tasks make up the bulk of compute work in ExaHyPE: We can roughly estimate that their number scales with $\mathcal{O}(N^d)$ for a d-dimensional setup with N^d cells. Their number grows volumetric. Skeleton tasks grow with roughly dN^{d-1} as they correlate to submanifolds within the domain.

Each enclave task is a well-defined unit of work: It is given a $(n+2) \times (n+2) \times (n+2)$ array of volume unknowns (or $(n+2) \times (n+2)$ for $d = 2$), and it yields a $(n+2)^d$ array of volumes in return. Each enclave task is conceptually a simple cascade of for-loops. All of these loops and data flow operations are application-generic, i.e. the same for any application. It is only within the very inner loop that the routines have to invoke domain-specific code.

Our vision is to offload the majority of enclave tasks to an accelerator. This is a light-weight, non-persistent, task-based offloading. It is light-weight, as we only offload atomic units (compute kernels) without cross-dependencies to the GPU. It is task-based as each offload corresponds to a logical task on the CPU for which we neither know whether it arises (grids may change any time and thus change the number of tasks) nor when it is spawned. Furthermore, several GPU offloads can be issued simultaneously. Our offloading is non-persistent as each enclave task is atomic, i.e. brings all of its data with it, and the computed data should eventually go back to the host where it is re-embedded into the overall computational domain.

3 Software architecture and user workflow

Peano is written with plain C++ using OpenMP threads. Back-ends for Intel's Threading Building Blocks and C++ threads exist, too. We do not use the latter two multithreading backends here; neither do we use the MPI extension.

Users configure their ExaHyPE run through a Python front-end. The front-end ensures that Peano sets up the right grid, runs through the grid appropriately, embeds the right data structures (patches) into the cells, and ties all tasks (both enclaves and skeletons) to the right compute kernels. These compute kernels are what ExaHyPE adds on top of Peano. They are "mere" lambda functions/functors given to generic Peano grid code. These lambdas realise the numerical scheme, i.e. the Finite Volume numerics in our case.

The generic ExaHyPE kernels have to know what the problem to be solved looks like. This works as follows:

- Users write a solver class in plain C++. The Python API is told the name of this solver class or the respective instantiation (object).
- Each solver class has to have a certain set of functions which are the actual domain knowledge.
- All grid setup and numerics are ran generically, but at certain points the generic kernels invoke the solvers' functions to compute user-specific outcomes.

We call this principle "Hollywood principle": Don't call us, we call you [8]. The user focuses on what PDE is solved, but the user has no say when and where the routines are actually invoked. The control logic stays within the engine.

For the design of how ExaHyPE supports GPUs, four guidelines shape our strategy:

1. As we know the pain of software rewrites—after all, this is the fourth generation of our meshing framework Peano already—the injection of GPU features should not add complexity to our core code components. That means in particular that no core code should depend on the GPU parts, these are completely optional, and the GPU code parts should not replicate core routines.

2. We want to be generic. Even though NVIDIA likely offers the most mature software and hardware ecosystem, ExaHyPE shall run on Intel and AMD platforms, too.
3. All GPUisation should be hidden from the ExaHyPE user. Our target community originates from Physics, Earth Sciences, Mathematics, and so forth. They should not be bothered with GPU specifics. However, they shall be able to benefit from their potential through ExaHyPE.
4. We try to keep the number of external dependencies small. Compared to the first generation of ExaHyPE, we have successfully kicked out Java, e.g., and reducing further dependencies where possible is definitely a commitment.

Item 2 and 3, in our opinion, almost rule out the usage of CUDA. Item 1 and item 4 rule out the adoption of high-level, machine-agnostic frameworks such as Kokkos. As such, we decided to strive for a pragma-based approach. As our code is heavily taskified and as we commit to item 4, we decided, for this hackathon, to try out OpenMP's GPU pragmas.

All of these strategic commitments can be challenged and we will revisit them eventually. This revisit also is necessary, as we have not been able to meet all of our guidelines 100%.

4 Technical lessons learned

On the following pages, I sketch stumbling blocks that we encountered while adding GPGPU support to our code. I also sketch how we tackled/resolved them, before I discuss my impressions and opinions about the solution.

4.1 Non-canonical loops

In several places of the code, we use `std::vector` to build up dynamic arrays. The container is very convenient: Different to arrays, we can pass it around with a length operation (so we don't have to hand over a pointer plus the array size), the class takes care of proper deallocation, and yet the underlying data structure is an array. The overhead is small as long as the vector does not grow (very) dynamically.

Issue 1. *Older GCC variants refuse to translate*

```
#pragma omp parallel for
for (auto p: myVector) {
```

Solution 1. *We replaced all "auto-loops" which shall run parallel with a standard C++ for-loop.*

C++ is not a compact language. A lot of features require quite some syntactic overhead. The for-loop with `auto` is a great example how we can reduce syntactic overhead and make code simpler. It is clear that such a container traversal might effectively be in non-canonical form, but it is not clear why the compilers can't figure that out. So we end up with a situation where modern C++ would help us to write more compact code, but the current-state OpenMP implementations undo these achievements.

The present case is a minor item, but a representative of an ongoing motif. Meanwhile, we did doublecheck this issue again and have not been able to reproduce it with GCC 9.3.1 and newer. CLANG seems to be fine, too. Similar issues however seem to persist around functors, e.g.

Follow-up question 1. *When we use an annotation system, we have to evaluate beforehand whether the newest C++ language constructs are supported, too. The compiler/language exensions tend to lack behind.*

4.2 Memory management

Our code realises a very "dynamic" algorithm: The mesh changes all the time, the numerical model is swapped in and out per cell—data structures thus have to be mapped onto each other—tasks are migrated from one node to the other [7], and so forth. There is neither a static scheduling pattern nor a homogeneous or invariant data access pattern. In a multitasking environment, this implies that tasks temporarily have to allocate buffers (also to move data from/to the GPU). Our code baseline therefore uses a lot of arrays on the stack:

Issue 2. *The non-GNU compilers do not allow us to use "dynamic" arrays on the callstack, i.e. they do not accept code fragments alike*

```
void foo(int N, ...) {
 double x[N];
```

Solution 2. *We replace such code fragments with explicit memory allocations on the heap (and hope not to forget the free):*

```
void foo(int N, ...) {
 double* x = new double[N];
 ...
 delete[] x;
```

We are aware that our baseline code is not C++ standard-conform but rather expects the compiler to support a C99 feature. We may not expect a compiler to support arrays on the stack if their size is not known at compile time. However, having arrays on the callstack is convenient. Even the OpenMP Technical Report 5 (Memory Management Support for OpenMP 5.0) uses this technique[1], too. It simplifies the code (not at least as we don't have to explicitly free data), and there's a speed argument to be made as we discuss below.

The really inconvenient experience however is that some compilers accept the on-stack allocation, while others accept it as long as no GPUs are used. We got the impression that the introduction of accelerator support is used as a vehicle to enforce a more rigorous interpretation of the C++ standard on the compiler side. This is unfortunate, as it makes the conversion of existing code more laborious. A softer approach (through deprecated warnings as we know if from the Java world, e.g.) would have saved us several man hours.

Issue 3. *With a lot of dynamic memory allocations (`new`s) within the threads, the performance went down.*

Solution 3. *We wrap the memory allocations in ifdefs:*

```
void foo(int N, ...) {
 #if defined(RelaxedCompiler)
 double x[N];
 #else
 #pragma omp alloc
 double* x = new double[N];
 #endif

 ...
 #if !defined(RelaxedCompiler)
 delete[] x;
 #endif
```

[1] cmp. `https://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf`, page 62, Section 4.1.

with the plan to annotate the dynamic heap allocation later such that OpenMP's memory allocators are used.

This solution is pragmatic: We keep a variant that is fast, yet prepare to migrate towards a standard-conform realisation. So far, we haven't experimented with OpenMP's allocators, as we have not seen a need for it (consult discussion below on managed memory). However, this seems to be an interesting aspect here. The rationale behind keeping the original variant is simple: It remains faster in many cases with an on-stack allocation.

Follow-up question 2. *It it not clear to us why arrays on the stack (C99 style) are not supported for the HPC community (for security-aware code, e.g., it obviously makes sense). For our code, we have to add* `-Wvla` *to identify all invalid uses, and eventually use preallocated memory through* `std::pmr::unsynchronized_pool_resource`*. The latter option implies that the code requires a reasonably new STL.*

Follow-up question 3. *Furthermore, it is clear in a task context, i.e. for such dynamic requests within an OpenMP task, that such allocations should be scalable and in low-latency memory. One may therefore ask why an explicit annotation will be required in the future.*

The ifdefs become painful in the context of our GPU port. In line with most other projects, we observe that data transfer to and from the accelerator become a bottleneck in our code. Therefore, we use managed memory plus explicit prefetches. Unfortunately, managed memory is not yet covered by OpenMP—maybe we missed the detail how to do it. We therefore ended up with three implementations:

Issue 4. *If data should be managed, we have to replace the allocation calls with a call to* `cudaMallocManaged`*. The free has to be diverted accordingly.*

Solution 4. *For our core kernels, we offer three implementation flavours: Allocation on the stack, standard C++ allocation (later to be augmented with OpenMP annotations) and CUDA managed memory allocation. Each allocation is paired up with the respective free mechanism. As we use our kernels both within (host) tasks and GPU offloading, the switch which allocation flavour to use has to be made via a function argument.*

This context-dependent switch is not an ideal solution: A simple array allocation unfolds into a multi-line code fragment with a case distinction. For the de-allocation, a similar pattern arises.

Follow-up question 4. *We would hope that future OpenMP generations allow us to skip the case distinctions and offers context-awareness: If a kernel is called within an OpenMP target, it needs a different treatment than a kernel called from within a standard task. At the moment, we have to pass this context through our callstack hierarchy explicitly. Furthermore, we hope that the deallocation becomes simpler. In an ideal case, a compiler identifies the "right" deallocation through code analysis.*

4.3 Lambdas and generic classes

All three previous generations of Peano have heavily relied on C++ templates: We parameterise over the dimensions of the solvers, over data types (which data records are associated with which grid entities, e.g.), and the actual user functions to be invoked by the generic compute kernels. As a consequence, a lot of code ends up in header files. This implies that the compile times are high, the executables grow large, and the error messages are difficult to understand.

With the fourth generation of Peano and the aligned ExaHyPE rewrite, we have been able to eliminate a lot of the templates through two refactoring/re-organisation techniques: On the

one hand, we decompose the engine into a set of static libraries and translate these libraries for 2d and 3d[2]. On the other hand, we replace templates where possible with functors fed by lambda expressions.

Issue 5. *We have not managed to declare routines that accept functors as arguments as GPU kernels.*

Solution 5. *We replaced functions alike*

```
void foo( std::function<void()> x ) {
```

with

```
template <typename F>
void foo( F x ) {
```

Once again, this is a pragmatic, mechanical rewrite of code parts and the rewrite is hidden from our users. They do not care if "their" routines implementing the physics are passed in via functor or template argument. However, it is a step back in our code evolution, as we return to templates, pack more and more code into header files, end up with longer compilation times plus more cryptic error messages. The proposed rewrite also compromises the idea that the generic numerical kernels should be organised within libraries; though we did not further investigate to which degree this librarisation did stop us from sticking to a functor-based software architecture.

We want to emphasise that we struggled to pass our template rewrites through the IBM compiler straightforwardly. This compiler seems to expect all declarations used within a template prior to the actual template instantiation. As a result, we ended up with major code reorganisation tasks. Once completed, the implicit template inlining within GPU offloading code resulted in out-of-memory errors and we had to manually adopt maximum memory sizes for the compile.

Follow-up question 5. *We have to continue to investigate to which degree compilers allow us to offer functions with functor arguments within (static) libraries which eventually are offloaded. It is our understanding that most GPU offloading relies on intense interprocedural "optimisation" (IPO) prior to the linking step when all object files are readily available. It is thus not clear why our original software architecture fails to compile with GPU offloading. The IPO should be able to remove the function calls and inline everything*[3]*.*

With a rigorous template mechanism, we have been able to use our generic kernels within OpenMP GPU kernels (targets). Not successful however has been the attempt to port our linear algebra utilities straightforwardly. This is a small set of operations orbiting around a vector class for tiny vectors plus the associated matrix operations.

To increase performance, Peano's tiny linear algebra classes and routines have always been templates anyway; though the compute kernels accept particular specialisations, i.e. instantiations with data types (double or float) plus sizes. The augmentation of the template with a `declare target` is conceptually straightforward, yet fails:

Issue 6. *We declare our (linear algebra) classes as target but the GNU compiler has not been able to handle overloaded constructors.*

Solution 6. *We rewrite our GPU kernels such that they do not accept an offloadable class. Instead, we manually extract the scalar fields from the object, and then puzzle the object together on the GPU again using these scalars; without using the vector constructors.*

[2]Peano supports up to $d = 7$, but this feature is not used in the present context.

[3]With the Intel compiler, a `#pragma forceinline recursive` plus IPO does work along the argument.

This is a very cumbersome fix.

```
tarch::la::Vector<3,double> myVector;
#pragma omp target to(myVector)
```

becomes

```
tarch::la::Vector<3,double> myVector;
double x0 = myVector(0);
double x1 = myVector(1);
double x2 = myVector(2);
#pragma omp target to(myVector) // this myVector thing does not work
{
  tarch::la::Vector<3,double> myVector;
  myVector(0) = x(0);
  myVector(1) = x(1);
  myVector(2) = x(2);

  tarch::la::Vector<3,double> myOtherVector(myVector); // rewrite code
   // such that constructor invocations are avoided (use standard
   // constructor only)
```

Our linear algebra classes are convenient as they offer many constructors: We can construct a vector from a sequence of scalars, we can copy vectors, or we can construct a vector from a plain array. None of these constructors is available within a GPU code.

Eventually, we decided to replicate our core compute kernels. The vanilla version relies on lambdas and copy constructors and is easy to understand (and, hence, to alter by numerics specialists). It also is written in a dimension-generic way. The GPU port manually maps the copy constructors onto a linearisation using scalars and (as discussed before) relies on templates.

Follow-up question 6. *It is not clear to us why overloading poses such a big problem for current OpenMP GPU offloading.*

Finally, our generic numerical kernels call back user functions. These are functions from PDE solver classes that the user has to provide. They implement the actual solver terms, i.e. PDE equation parts. Our original software design reads as follows: The user specifies on the Python API level what type of solvers she wants (this includes which terms are actually found within the PDE, whether they are non-linear, whether there are stiff source terms, and so forth) and how many of these solvers are used at the same time. In theory, ExaHyPE allows users to run multiple solvers with different parameter sets concurrently; a feature attractive for uncertainty quantification, e.g. The Python API then puzzles the solvers together via a C++ class structure: An abstract superclass specifies which functions have to be provided by the user as C-code according to the spec (via an implementation of the abstract superclass), while a repository of classes holds the instances of the solver. The generic compute kernels then invoke methods from the user's class implementations.

Issue 7. *Generic GPU kernels require state-less, side-effect free implementations of the PDE terms that do not use any virtual function tables, if we hold the objects on the CPU yet want to use some of its routines on the GPU, too.*

Solution 7. *As it is not possible to overload a method with a* `static` *alternative or to annotate methods to highlight that they are state-less, users have to provide these additional GPU routines manually. The additional routines then are annotated with a* `declare target` *pragma.*

This solution is not 100% what we had in mind—we wanted to come up with a solution that does not require any additional work from the user side—but it is the best solution we could come up with.

Follow-up question 7. *For OpenMP offloading, we would need a mechanism for users to annotate their C++ class members to highlight that a member function is side-effect free and stateless even though it is not declared as static to meet a superclass' signature needs. Such a function could internally be converted/replicated by a compiler into a static function and hence made offloadable. This would simplify our code yet is not possible at the moment.*

Further to that, we have not been able to puzzle out completely, why we have to declare our functions as offloadable at all. They are now invoked from templates, i.e. a code analysis could defer the need for a GPU version including all method properties.

4.4 General remarks

Besides the major code refactorings sketched, we had to make a few minor changes all over the place.

Issue 8. *GPU code does not support string operations.*

Solution 8. *Remove all routines alike* `toString` *(or* `to_string`*) from classes that have to be offloaded to the GPU.*

For the majority of our classes, we have made the string representation routines non-member functions declared as friends. They can then return a human-readable representation of the object, while the object remains suitable for GPU offloading. This refactoring is in line with the C++ definition of a `<<` operator.

As for many HPC codes, we usually do not use our string routines. They are solely there to generate error messages and, sometimes, some statistics. If we want to deploy routines to the GPU, we can, in principle, accept that we have to get rid of sanity checks and stats. This is a design decision. It is problematic however that we might have to replicate our functions (the prototype for the CPU including sanity checks, string representations and statistics) plus a speced-down version for the accelerator. This does not improve the quality of our code base.

Issue 9. *Our code relies—for the non-production runs—heavily on our bespoke assertions. As the C++ language assertions, they terminate a code immediately if they are harmed result in non-offloadable code. Beyond that, they also report on the context of the violation dumping some related variables, e.g.*

Solution 9. *We disable all assertions once we build a GPU version of our code base.*

This does not a major deal either, but it has severe implications: We heavily rely on assertions to ensure that our code remains correct. Whenever an assertion fails, we make it dump a lot of relevant context in a human-readable format such that bug hunting is as simple as possible. For our large code base, the absence of an assertion/termination mechanism for GPU code means that we have to rely on more and more unit tests to identify errors and that debugging does not become easier. An alternative solution in line with the string discussion would be to provide each compute kernel twice: with bespoke assertions and with pure C++ language assertions.

Follow-up question 8. *We will have to track carefully to which degree future OpenMP GPU codes offer mechanisms to realise assertions plus meaningful error messages.*

This stripping of feedback that we have realised due to our GPU offloading is in line with a general problem we faced while we ported the code:

Issue 10. *The error messages resulting from GPU offloading are hard to digest.*

The whole exercise reminded us heavily about our initial work with heavily templated code. Cryptic error messages late throughout an extensive compile process (maybe at link time) make the day-to-day development work hard.

5 Summary

Our team has succeeded with porting the ExaHyPE 2 engine onto GPUs. This is the starting point for our investigations into a scaling MPI+X+GPU code generation. While the functional requirements (it has to run on the accelerators) are met, we are not 100% happy about how we met objectives. We face a traditional conflict of functional vs. non-functional requirements. We wrap up this presentation by a reiteration of our non-functional objectives and discuss to which degree they have been met:

1. *GPU code parts should not replicate core routines and core routines should not depend on GPU code.* While the latter point is almost met (we have not introduced any GPU dependencies into the baseline code besides the more complex treatment of memory allocations), our GPU code replicates massive amounts of code from the baseline version. This would be acceptable if we had already performed GPU-specific optimisations—it is clear that changes in the data layout, loop traversals, ... requires code rewrites. However, our code redundancies right now solely result from technical constraints.

2. *We want to be platform-generic.* This criterion is not met in all places. Memory allocations for example require CUDA routines, and it is not clear whether these allocations could be transfered directly to AMD accelerators.

3. *GPUisation hidden from the engine user.* This criterion is not met fully, as users have to offer their core PDE routines as additional static variants, too: They have to implement a C++ class interface through a function with an `override` qualifier plus have to offer this function again as static routine if they want to use GPUs. In an ideal world, this manual re-implementation could be done via a simple annotation.

4. *Keep the number of dependencies small.* Despite sticking to an open standard with OpenMP, our software depends on the specific ecosystem around. The memory allocation is the prime example for this, but we also failed to port our code to the CLANG ecosystem while using `std::complex`. We hope this will homogenise/streamline soon.

The latter point has become worse as we tried to use C++ language features such as lambdas plus functors or C99 techniques such as stack allocation. To make these fit to (some) GPU compilers, we had to build them back into old-school templates with all of their drawbacks.

From hereon, we plan to invest into an optimisation of ExaHyPE's accelerator data flow and accelerator compute kernels. Throughout this process, we will re-examine facts that we consider to be weaknesses in the present report, and we will study new tool generations. We also assume that our own understanding of C++ and GPU offloading improves. Besides the continuing investment into OpenMP, we plan to repeat this exercise with SYCL soon. A lot of the difficulties result from our ambition to marry a complex object-oriented code with a pragma-based parallelisation approach which addresses hardware that is designed for massive data parallelism. It will be interesting to see whether the SYCL approach is inherently a better fit to our object-oriented mindset.

Acknowledgements

The work presented results from a workpackage in the project ExaClaw, funded via EPSRC's Excalibur programme under grant number EP/V00154X/1 (ExaClaw). It also aligns with research in ExCALIBUR's Massively Parallel SPH for Engineering and Astrophysics at the Exa-Scale (EP/V001523/1). The group has been supported by researchers funded through the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE), and has received funding from the European Union's Horizon 2020 research and innovation programme under the ChEESE project, grant agreement No. 823844. All outcomes became possible only due to the Sheffield GPU Hackathon, July 27th—31st, 2020, generously supported and delivered by NVIDIA. Additional support has been provided through Durham's Advanced Research Computing (ARC) and the team around the N8 supercomputer Bede.

References

[1] M. Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[2] M. Bader, M. Dumbser A.-A. Gabriel, L. Rezzolla, and T. Weinzierl. ExaHyPE—An Exascale Hyperbolic PDE Engine, 2020. www.peano-framework.org.

[3] D. E. Charrier, B. Hazelwood, E. Tutlyaeva, M. Bader, M. Dumbser, A. Kudryavtsev, A. Moskovsky, and T. Weinzierl. Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *Int. J. High Perf. Comp. Appl.*, 33(5), 2019.

[4] D. E. Charrier, B. Hazelwood, and T. Weinzierl. Enclave tasking for dg methods on dynamically adaptive meshes. *SIAM Journal on Scientific Computing*, 42(3):C69–C96, 2020.

[5] A. Dubey, A. S. Almgren, J. B. Bell, M. Berzins, S. R. Brandt, G. Bryan, P. Colella, D. T. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. van Straalen, and K. Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, 2016.

[6] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. ExaHyPE: an engine for parallel dynamically adaptive simulations of wave problems. *Comp. Phys. Comm.*, page 107251, 2020.

[7] P. Samfass, T. Weinzierl, D. E. Charrier, and M. Bader. Lightweight task offloading exploiting mpi wait times for parallel adaptive mesh refinement. *Concurrency and Computation: Practice and Experience*, page e5916, 2020.

[8] T. Weinzierl. The peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Trans. Math. Softw.*, 45(2):14:1–14:41, 2019.

[9] T. Weinzierl. Peano, 2020. www.peano-framework.org.