Computational Geosciences (2021) 25:1579–1590
https://doi.org/10.1007/s10596-021-10051-4

ORIGINAL PAPER

Leveraging HPC accelerator architectures with modern techniques — hydrologic modeling on GPUs with ParFlow

Jaro Hokkanen[1] · Stefan Kollet[1] · Jiri Kraus[2] · Andreas Herten[3] · Markus Hrywniak[2] · Dirk Pleiter[3]

Received: 15 December 2020 / Accepted: 18 March 2021 / Published online: 1 May 2021

Abstract
Rapidly changing heterogeneous supercomputer architectures pose a great challenge to many scientific communities trying to leverage the latest technology in high-performance computing. Many existing projects with a long development history have resulted in a large amount of code that is not directly compatible with the latest accelerator architectures. Furthermore, due to limited resources of scientific institutions, developing and maintaining architecture-specific ports is generally unsustainable. In order to adapt to modern accelerator architectures, many projects rely on directive-based programming models or build the codebase tightly around a third-party domain-specific language or library. This introduces external dependencies out of control of the project. The presented paper tackles the issue by proposing a lightweight application-side adaptor layer for compute kernels and memory management resulting in a versatile and inexpensive adaptation of new accelerator architectures with little draw backs. A widely used hydrologic model demonstrates that such an approach pursued more than 20 years ago is still paying off with modern accelerator architectures as demonstrated by a very significant performance gain from NVIDIA A100 GPUs, high developer productivity, and minimally invasive implementation; all while the codebase is kept well maintainable in the long-term.

Keywords High-performance computing (HPC) · GPU computing · Distributed memory parallelism · Accelerator architecture · Domain-specific language (DSL)

Mathematics Subject Classification (2010) 65Y05 · 68–04 · 68N19

1 Introduction

High-performance computing plays an important role in many computation-intensive research areas, including physics, earth sciences, national security, biology, engineering, climate modeling, aerospace, and energy [1]. However, rapidly developing accelerator architectures pose a significant challenge to scientific communities trying to keep up with the technological change [2]. In the past, many projects required little changes to scientific codebases due to architectural changes. A typical scientific program relying on MPI [3] for parallelism was able to easily leverage the new hardware providing a larger number of available CPU cores with higher clock speeds. More recently, the expected ending of Moore's law and the growing diversity of hardware accelerators are making software performance engineering increasingly important [4].

In addition to housing a large number of CPUs, modern heterogeneous supercomputer setups already often consist of other processing architectures such as GPUs or (less often) FPGAs. Benefiting from these accelerator architectures requires not only exposing sufficient parallelism but often also changes to data structures in order to efficiently exploit the memory subsystem of the devices in question. It is not uncommon that the codebases have been developed for decades and consist of hundreds of thousands or even millions of lines of code [2]. The fundamental assumptions behind the scientific models change at a much slower pace compared to the employed hardware architectures, and thus, in the absence of hardware changes, many of these implementations could possibly be used for decades to come. Therefore, it is important to

✉ Jaro Hokkanen
j.hokkanen@fz-juelich.de

[1] Agrosphere (IBG-3), Forschungszentrum Jülich GmbH, Jülich, Germany

[2] NVIDIA GmbH, Würselen, Germany

[3] Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich GmbH, Jülich, Germany

find a way to efficiently leverage the new hardware developments with the resources available at the scientific institutions.

The adoption of accelerator architectures is still at an early phase for many projects, and in contrast to message passing and MPI, no single overarching standard or programming interface exists. Therefore, it is difficult to predict how to future-proof the codebase such that the significant investment spent into adapting the code projects for the current-generation acceleration architectures could also be leveraged with the future architectures as far as possible.

This paper demonstrates how the proposed forward-looking practices followed in ParFlow development more than 20 years ago keep paying dividends even with the most recent accelerator architectures. It is further argued that it is not too late even for established projects to follow such a versatile approach where the dependency on the programming models used with the current accelerator architectures is minimized, and the possibility to extensively leverage the investment with future architectures is maximized. Furthermore, a reference implementation is provided wherein GPU capabilities are added into the ParFlow hydrologic model using CUDA as the backend option of choice. Thus, the presented study takes up the challenges and suggestions posed by Lawrence et al. [2] and shows a practical path forward using the example of a real-world, large-scale scientific codebase.

The text is structured as follows. Section 2 discusses the common programming models used to add support for accelerator architectures. Section 3 briefly presents the ParFlow embedded domain-specific language, which plays an important role in the reference implementation presented in Section 4. Section 5 evaluates the performance implications of running ParFlow with GPU acceleration. Finally, conclusions are given in Section 6.

2 Programming models for accelerator architectures

2.1 Established approaches

In order to leverage accelerator architectures, many scientific software projects decide to use parallel programming models such as CUDA, HIP, OpenACC, OpenCL, or OpenMP directly in the source files dealing with the underlying scientific problem (i.e. the scientific code). Other projects decide to rely on domain-specific third-party frameworks (e.g., Firedrake [5], GridTools [6], and PSyclone [7]) or more general lower-level libraries (e.g., Alpaka [8], Kokkos [9], and RAJA [10]), which is commonly referred to as separation of concerns, because the scientific concerns are separated from concerns related to the (heterogeneous) massively parallel hardware.

The first approach, herein referred to as the *direct approach*, generally allows incremental development and can provide good productivity and performance. However, the direct approach introduces external dependencies to the respective native software stack and usually leads to a significant amount of code changes. This adds complexity to the user-facing scientific code which often becomes tightly integrated with the chosen accelerator programming model.

On the other hand, when relying on a third-party framework (separation of concerns), the short-term productivity highly depends on the intricacies of the chosen framework. In many cases the short-term productivity may be reduced due to insufficient flexibility; either too much code rewriting, too many algorithmic or data structure changes, or inadequate support for incremental adoption [10]. In contrast to the direct approach, the separation of the scientific code from the accelerator architectures is improved such that benefiting from new hardware developments, ideally, does not require any changes to the scientific code after the third-party framework has been successfully implemented. Furthermore, the codebase is well maintainable and additional support for future architectures is outsourced to a third party, which ideally minimizes the long-term development effort. However, building a large scientific codebase around a single third-party project also constitutes a risk; as many third-party options have emerged, it is difficult to predict which projects will be sustainable in the long-term and continue to add support for all desired architectures.

2.2 Proposed approach

Herein a third approach inspired by ParFlow is proposed which aims to combine the best properties of both aforementioned approaches; a lightweight adaptor layer that provides a domain-specific interface for the memory management and compute kernels for the underlying application (Fig. 1).

The intention is to leverage the lightweight adaptor layer to enable easy accommodation of not only one or more programming models, such as CUDA, HIP, OpenACC, OpenCL, or OpenMP, but also sufficiently flexible third-party libraries such as Alpaka, Kokkos, or RAJA. Thus, insourcing development to support all required accelerator architectures on the local backend can be avoided, and the interface for memory management and looping is independent of the used accelerator programming model allowing full customization in the underlying scientific domain. Furthermore, the cost of choosing a wrong accelerator programming model is minimized and adding support for new programming models including libraries is straightforward. A number of pros and cons of the proposed approach are listed below:

+ Separation of concerns
+ Incremental adoption
+ Flexibility with algorithms and data structures
+ Fully customizable interface for compute kernels and memory management

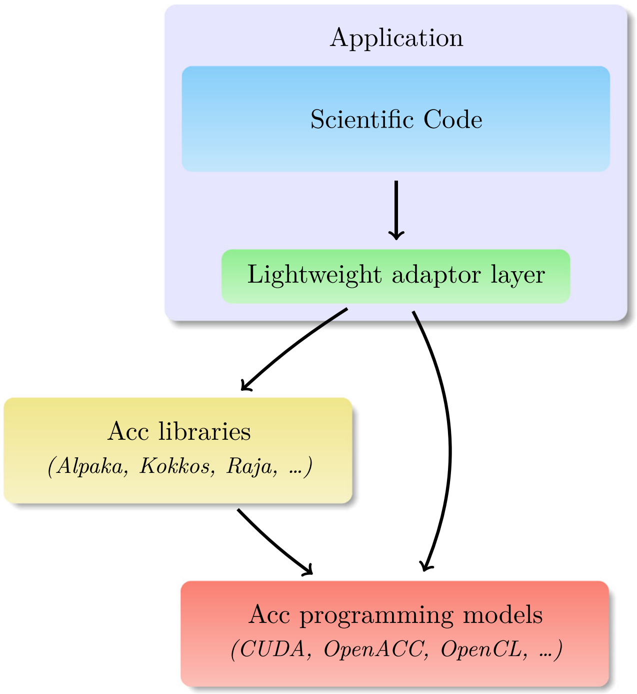
Application
Scientific Code
Lightweight adaptor layer
Acc libraries
(Alpaka, Kokkos, Raja, ...)
Acc programming models
(CUDA, OpenACC, OpenCL, ...)

Fig. 1 A lightweight adaptor layer for accelerator programming models and libraries

+ Codebase remains well maintainable
+ Easy adoption of one or more accelerator programming models or libraries
+ Cost of choosing a "wrong" programming model or library is minimized
– Development and implementation of a lightweight adaptor layer
– Compatibility of the adaptor layer with all future backends is not guaranteed

3 ParFlow embedded domain-specific language

ParFlow has a long development history dating back to the 1990s [11]. Hydrologic problems such as groundwater and overland flow are modeled using finite difference and finite volume schemes on a regular Cartesian grid [12–14]. The time integration is based on implicit methods, which requires finding solutions for large systems of equations. In order to solve the discretized problem in parallel on a distributed memory system such as a modern supercomputer, the computational domain is decomposed into multiple subdomains. However, the computations associated with each subdomain are not independent of each other due to the stencil operations of the flow problem which require each cell in the grid to communicate with its neighbors.

If the neighboring cell is found from another subdomain, cross-subdomain communication is required. This communication pattern is generally referred to as the halo exchange and relies on MPI such that each subdomain is associated with an MPI process. In addition to distributed memory parallelism (multiple nodes), ParFlow uses MPI also in place of shared memory parallelism (multiple cores on a single node).

The domain decomposition and message passing requirements pose significant hardware architecture complexities in the implementation. For a scientific programmer developing numerical methods this may be cumbersome and error-prone. In ParFlow, the scientific code relies on a high-level API provided by the ParFlow embedded domain-specific language (ParFlow eDSL) which abstracts away the architecturally imposed complexities. Many operations such as allocations and initializations (Listing 1), data accesses (Listing 2), message passing (Listing 3), and compute kernels (Listing 4) are accessed through the eDSL API which is mostly comprised of C preprocessor macros.

Listing 1: Allocation and initialization.

```
KW = NewVectorType(grid2d, 1, 1, cell_centered);
InitVector(KW, 0.0);
```

Listing 2: Data access.

```
ix = SubgridIX(subgrid);
iy = SubgridIY(subgrid);
iz = SubgridIZ(subgrid);
```

Listing 3: Message passing.

```
vector_update_handle
  = InitVectorUpdate(pressure, VectorUpdateAll);
FinalizeVectorUpdate(vector_update_handle);
```

Listing 4: Loops.

```
GrGeomInLoop(i, j, k, gr, r, ix, iy, iz, nx, ny, nz,
{
  int ips = SubvectorEltIndex(ps_sub, i, j, k);
  data[ips] = value;
});
```

Similarly to the domain decomposition and message passing, adding support for modern accelerator architectures further complicates the implementation. The currently available accelerator devices are typically designed for parallel computations using an internal device memory. Therefore, memory management and parallelizable compute kernels are of particular concern regarding the implementation. Fortunately for ParFlow, memory management and compute kernels are accessed through the ParFlow eDSL, thus, constituting an ideal layer for adding the support for the accelerator architectures.

4 Real-world example: Adding GPU-support into ParFlow eDSL

This section provides a detailed description of the work performed to add GPU support into the ParFlow eDSL using CUDA. This may serve as a blueprint for other projects developing a strategy to add support for new architectures into their codebases. Even projects which do not already have a suitable adaptor layer in place may benefit from the proposed approach, because it may be relatively easy to separate the code dealing with GPUs.

It is important to note that the ParFlow implementation to support GPUs heavily leverages *Unified Memory*. Unified Memory allocations are accessible by the CPU (host) and the GPU (device). This is achieved by automatically migrating data at the level of individual pages depending on where the data is accessed. It thus allows using the same pointer in host and device code paths. The benefits include a significant decrease in the development effort, and a less invasive and complex implementation.

The code dealing with compute kernels is built directly into the preprocessor macros of the ParFlow eDSL such that the macro definitions depend on the chosen architecture (or the chosen accelerator programming model). Most changes due to accelerator support are associated with the header files for macros and the message passing layer, and therefore, few changes to the scientific code are required. The following subsections provide a representative view of the GPU implementation and development effort using slightly simplified examples from the ParFlow codebase. The different steps encompass the compilation process, GPU affinity, memory management, loop parallelism, and GPU-GPU message passing. An additional subsection describes profiling and optimization procedures, which are essential components of the implementation. The full ParFlow codebase is available in a public repository at https://github.com/parflow (last access: 27th October 2020).

4.1 Compilation process

In order to build a project with CUDA, the first step is to set up the build configuration accordingly. In ParFlow, the compilation process is controlled by CMake. Compiling with GPU support is optional and can be enabled by passing an argument to CMake when configuring the project. If GPU acceleration is enabled, CMake performs the following additional tasks:

- Find CUDA installation
- Set CUDA host compiler
- Define CUDA compiler arguments
- Find CUDA-specific external libraries (CUB, RMM)
- Assign correct source files to the CUDA compiler

4.2 GPU affinity

Modern supercomputers typically consist of large numbers of nodes which may have multiple GPUs available. For a program such as ParFlow which originally relies solely on a message passing library for parallelism, the best option often is to launch the same number of processes as there are GPUs intended to be used. Each process then uses only one GPU and the communication between GPUs relies on a CUDA-aware MPI library that supports direct GPU-GPU data transfers. The correct GPU device for each process can then be determined using Listing 5

Listing 5: Determination of the correct GPU device. Further optimization of GPU placement for halo exchange would be possible but is not covered in this paper.

```
cudaSetDevice(node_local_rank % local_num_devices);
```

where node_local_rank and local_num_devices are the node-local rank of the process and the number of GPUs associated with the corresponding node, respectively. In the early phase of accelerating a legacy application often significant portions of the runtime are still spent in MPI parallel CPU only code paths. In this case, it is beneficial to exploit the MPI parallelism in these CPU code paths by launching more MPI processes per node as there are GPUs and share each GPU between multiple MPI ranks via CUDA Multi-Process Service (MPS).

4.3 Memory management

As the physical memory spaces between the host and the devices are separated, it is essential to make sure that the relevant stored data is accessible for each device. With the current CUDA version, this means replacing the standard host memory allocations and deallocations by the functions provided by the CUDA toolkit. While the CUDA API provides means to allocate memory on the host-side such that the device can access this data directly through the PCI Express bus or NVLink, storing the data in the device memory is usually more efficient. This is achieved by the CUDA functions *cudaMalloc* and *cudaMallocManaged* for device-pinned and Unified Memory allocations, respectively. Furthermore, *cudaFree* must be used for deallocations.

One drawback of simply replacing the required standard host memory allocations by a call to *cudaMallocManaged* is the significantly increased memory allocation overhead. This may cause a problem in case of recurring allocations and deallocations. For this reason, ParFlow supports using

Rapids Memory Manager (RMM) for Unified Memory allocations. Instead of calling *cudaMallocManaged* directly, a function *rmmAlloc* provided by the RMM API is called which then calls *cudaMallocManaged* internally. RMM provides a pool allocation mode in which the memory pool is prefetched to the device and the memory is never deallocated while the pool is in use and allowed to grow. A call to *rmmFree* makes room for new allocations without decreasing the pool size. This removes the overhead of recurring Unified Memory allocations without a considerable increase in peak memory usage (although the average memory consumption is increased). The performance impact is shown in Section 5.

In ParFlow, dynamic memory allocation is handled by the eDSL preprocessor macros *talloc* or *ctalloc* as shown in Listing 6 for a *Vector* type.

Listing 6: ParFlow dynamic memory allocation.

```
vector = talloc(Vector, 1);
```

For a normal host memory allocation, the preprocessor simply replaces *talloc* by a call to *malloc* (Listing 7). If ParFlow is compiled without GPU acceleration, this is always the case. In case of a Unified Memory allocation, a static inline function *_talloc_cuda* is called which then calls either *cudaMallocManaged* or *rmmAlloc* (depending on if ParFlow is compiled with the RMM library) as shown in Listing 8.

Listing 7: ParFlow eDSL host memory allocation.

```
#define talloc(type, count) \
  (type*)malloc(sizeof(type)
    * (unsigned int)(count))
```

Listing 8: ParFlow eDSL device memory allocation.

```
#define talloc(type, count) \
  (type*)_talloc_cuda(sizeof(type) \
    * (unsigned int)(count))

static inline void *_talloc_cuda(size_t size)
{
  void *ptr = NULL;
#ifdef PARFLOW_HAVE_RMM
  rmmAlloc(&ptr, size, 0, __FILE__, __LINE__);
#else
  cudaMallocManaged(&ptr, size);
#endif
  return ptr;
}
```

Similarly, the memory is deallocated using the preprocessor macro *tfree* as shown in Listing 9.

Listing 9: ParFlow dynamic memory deallocation.

```
tfree(vector);
```

Listings 10 and 11 show the corresponding macro definitions for host memory and Unified Memory deallocations. The memory allocated using *malloc*, *cudaMallocManaged*, or *rmmAlloc* is deallocated using *free*, *cudaFree*, or *rmmFree*, respectively.

Listing 10: ParFlow eDSL host memory deallocation.

```
#define tfree(ptr) free(ptr)
```

Listing 11: ParFlow eDSL device memory deallocation.

```
#define tfree(ptr) _tfree_cuda(ptr)

static inline void _tfree_cuda(void *ptr)
{
#ifdef PARFLOW_HAVE_RMM
  rmmFree(ptr, 0, __FILE__, __LINE__);
#else
  cudaFree(ptr);
#endif
}
```

Due to performance reasons, Unified Memory allocations are only used in those compilation units where they are actually needed, thus the behavior of *talloc* may differ between the compilation units (the term *compilation unit*, also called translation unit, herein refers to the input for a compiler from which an object file is generated). An important consequence of this is that each memory allocation and the corresponding deallocation must be contained within compilation units using the same macro definitions (preferably the same compilation unit) such that each allocation call is eventually followed by the correct deallocation call.

4.4 Loops

In ParFlow, loops over the discretized domain are always accessed through the eDSL API. Similarly to memory management, the loop execution is defined by preprocessor macros. However, only a few general loop macros are provided for which the loop body is given as a macro argument. In ParFlow, the loop body is typically provided as the last argument to the macro, e.g., in Listing 12 the loop body refers to the contents within the curly brackets. This approach allows using the same loop macro for a large number of loops with different loop logic and a varying number of variables required by the loop. In fact, there are over one hundred loops with often very different loop bodies that use a single loop macro in ParFlow.

The definitions for the loop macros depend on whether they are executed on the host or the device. For example, the loop macro used in Listing 12 has been defined for a sequential execution on the host and for a parallel execution on the device in Listings 13 and 14, respectively. The sequential definition is straightforward as the loop body is just placed inside the innermost loop in the macro definition. However, in the parallel version, a GPU kernel must be launched for which the loop body macro argument cannot be directly passed. Instead, a relatively new CUDA feature known as extended host-device lambda is used to pass the loop body to the GPU kernel. The loop body macro argument is placed inside the lambda function such that the lambda function contains all required information about the loop logic; the variables found inside the loop body are captured by their value. Now all required information about the loop body can be passed to the GPU kernel as a single argument, i.e., the lambda function.

Listing 12: BoxLoopI0: a simple loop over the discretized domain.

```
double *fp;
double *pp;
double value;
Subvector *f_sub;

/* some code missing here */

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
{
  int ip = SubvectorEltIndex(f_sub, i, j, k);
  fp[ip] = pp[ip] - value;
});
```

Listing 13: BoxLoopI0 macro definition for sequential execution on the host.

```
#define BoxLoopI0(i, j, k, ix, iy, iz,                \
  nx, ny, nz, loop_body)                              \
{                                                     \
  for (k = iz; k < iz + nz; k++)                      \
    for (j = iy; j < iy + ny; j++)                    \
      for (i = ix; i < ix + nx; i++)                  \
      {                                               \
        loop_body;                                    \
      }                                               \
}
```

Listing 14: BoxLoopI0 macro definition for parallel execution on the device.

```
#define BoxLoopI0(i, j, k, ix, iy, iz,                \
  nx, ny, nz, loop_body)                              \
{                                                     \
  auto lambda_body = [=] __host__ __device__          \
    (const int i, const int j, const int k)           \
      loop_body;                                      \
                                                      \
  /* some code missing for grid & block sizes */      \
                                                      \
  BoxKernelI0<<<grid, block>>>(lambda_body,           \
    ix, iy, iz, nx, ny, nz);                          \
}
```

Listing 15 shows the general GPU kernel launched by the macro definition in Listing 14, which takes the aforementioned extended host-device lambda function as an argument. Apart from determining the correct thread indices *i*, *j*, and *k*, the kernel only calls the lambda function by passing the corresponding thread indices as arguments to the lambda function.

Listing 15: A general GPU kernel used by the BoxLoopI0 macro.

```
template <typename LambdaBody>
__global__ static void BoxKernelI0(
  LambdaBody loop_body,
  const int ix, const int iy, const int iz,
  const int nx, const int ny, const int nz)
{
  int i = ((blockIdx.x * blockDim.x) + threadIdx.x);
  int j = ((blockIdx.y * blockDim.y) + threadIdx.y);
  int k = ((blockIdx.z * blockDim.z) + threadIdx.z);

  if(i < nx && j < ny && k < nz)
  {
    i += ix;
    j += iy;
    k += iz;
    loop_body(i, j, k);
  }
}
```

The kernel in Listing 15 is used for the basic three-dimensional *parallel for* loop type in ParFlow. *Parallel reduction* loops use more complex (but still general) GPU kernel. The described approach allows incremental development and easy parallelization of a large number of compute kernels, while minimizing the amount of new code. However, it is important to note that the parallel loop macros pose some additional restrictions to the loop body; the most common restrictions are listed below:

- Host variables defined outside the loop body cannot be changed
- Pointers must point to Unified Memory allocations
- Functions called inside the loop body must have *__host__ __device__* identifier
- Operations causing race conditions (e.g. increment) must use atomic functions

4.5 GPU-GPU message passing

Most of the recurring intra-node and inter-node communications between the processes such as the halo exchange involve data that is stored on a GPU and needed by another GPU. Therefore, efficient data transfer between GPUs on a node and also across nodes is important. The data could be copied from a GPU to a staging buffer on the host, then transferred to the host staging buffer of another process using a message passing library, and finally copied back to a GPU device. In this case, the choice of the accelerator architecture would not pose any requirements for the message passing library, but the resulting performance would be bad due to many unnecessary operations that are not properly pipelined.

Better performance can be obtained by leveraging direct GPU-GPU communication such as NVIDIA GPUDirect or AMD DirectGMA. For example, GPUDirect Peer-to-Peer (P2P) and Remote Direct Memory Access (RDMA) enable direct data transfers between two GPUs (intra-node) and a GPU and a network adapter (inter-node), respectively. However, usage of these technologies requires additional support from the message passing library. For example, at the Jülich Supercomputing Centre, Germany, ParFlow is frequently run with MVAPICH2-GDR and Parastation MPI which both support GPUDirect P2P and RDMA, and are often referred to as CUDA-aware MPI libraries.

The default message passing option in ParFlow relies on derived MPI datatypes and MPI library-side data packing and unpacking. When this message passing option is used with GPU acceleration, the pointers passed to the MPI library point to Unified Memory allocations. However, the authors found no CUDA-aware MPI library which would pack and unpack the data for the underlying MPI data type on the GPU, and leverage the fast GPUDirect data transfers. After implementing optimized GPU kernels for application-side data packing and unpacking, and using a simple MPI_BYTE data type, efficient GPUDirect data transfers were leverageable with both aforementioned CUDA-aware MPI libraries, MVAPICH2-GDR and Parastation MPI.

When using GPU acceleration in ParFlow, the application-side data packing and unpacking for each process is performed in multiple streams on a GPU using a pinned GPU staging buffer; a pointer to this staging buffer is then passed to the MPI library. Using pinned GPU memory instead of Unified Memory for the staging buffers typically results in better performance because the MPI library must internally use a pinned buffer anyway (GPUDirect data transfers do not support Unified Memory).

The changes required to leverage GPU-GPU message passing have been implemented into the message passing layer of ParFlow which is not solely based on preprocessor macros but is instead compiled as a separate library.

4.6 Profiling & Optimization

In the implementation of the GPU support in ParFlow, the GPU utilization during runtime was monitored with various NVIDIA profiling tools such as Visual Profiler, Nsight Systems, and Nsight Compute. In ParFlow, a large number of compute kernels is found, none of which clearly dominate the wall time. The parallelization of these compute kernels was performed with the help of profiler output one compilation unit at a time. No architecture-specific optimizations were introduced into the loop body macro arguments (cf. Section 4.4) which are shared between the host and device compilation paths.

Little performance improvement was realized until most of the frequently executed loops were offloaded to the GPUs. This is explained by page faults and recurring data migrations between the host and device along the PCI Express bus or NVLink. When a virtual page is accessed that is not mapped to a physical page on the memory of the underlying processing unit, a page fault is generated. The issue is resolved during the runtime by locating and copying the data and remapping the virtual page to the corresponding physical page such that it is now accessible to the processing unit in question. This is referred to as on-demand paging and is supported on the GPU-side since the NVIDIA Pascal architecture; for older NVIDIA architectures, all Unified Memory is always migrated to the device memory prior to launching a GPU kernel.

The single most important part of the optimization was to minimize the page faults and avoid recurring memory transfers between host and device. This was mostly achieved by offloading all loops accessing the same data to the GPUs therefore minimizing the need to migrate pages to the host memory.

CUDA also offers functions *cudaMemAdvise* and *cudaMemPrefetchAsync*, which can be used to provide information about the memory usage, or to prefetch data to the desired location, respectively. Unfortunately, the employed adaptor layer is not aware of the individual pointers captured by the extended host-device lambda functions. Therefore, it is difficult to embed these into the ParFlow eDSL without modifying the scientific code, and thus memory advise or prefetching are not extensively used in the implementation. However, this is not a significant drawback, because the corresponding data is mostly accessed from the GPUs in the first place.

Due to the incremental approach for development, all compute kernels were initially parallelized using a simple parallel for construct and race conditions were handled using atomic functions. However, this is a very ineffective strategy for certain types of compute kernels such as reduction kernels where the execution becomes highly sequential. Efficient parallel reduction kernels that leverage the CUB (CUDA Unbound) header-only library were, thus, added to the ParFlow eDSL.

ParFlow is memory parsimonious, thus, memory is frequently allocated and deallocated in loops. However, allocating and deallocating Unified Memory leads to significantly higher overhead than the standard host memory allocation. Therefore, a pool allocator (Rapids Memory Manager) is supported as described in Section 4.3, resulting in considerably better performance (see Section 5). The trade-off is increased average memory consumption without a considerable increase in peak memory usage.

Another high priority memory optimization is coalescing the global device memory accesses. Considering the architectures starting from Pascal, the global memory is accessed in transactions of 32 bytes in size. If a warp of 32 threads executing the same instruction accesses a 32-byte aligned array of 128 bytes, only a maximum of 4 global memory transactions are required. This is the case for example when k-th thread within a warp accesses the k-th 4-byte integer of a 32-byte aligned array. In case the array is not aligned, 5 transactions are required. On the other hand, for a strided array with the stride size larger than 32 bytes, each thread requires a separate transaction resulting in a total of 32 transactions. In ParFlow, the data along the x-dimension of the domain is typically stored in consecutive memory locations. Therefore, mapping this dimension to the x-dimension of a three-dimensional grid of a GPU kernel results in an ideal memory access pattern for non-strided arrays.

With CUDA, multiple GPU kernels can run concurrently while the CPU is performing other tasks. Therefore, better performance is achieved when the number of synchronizations between host and device are minimized. Generally, the CPU does not need to wait for a GPU kernel to finish immediately after launching a kernel and can instead continue the program execution also queuing more GPU kernels for execution. However, a synchronization is required before the CPU accesses data that is relevant for the device kernels. Unfortunately, in ParFlow, the adaptor layer is not aware of the program control flow of the scientific code and does not know when synchronizations are needed. Therefore, as a default option, *cudaStreamSynchronize* is called after each kernel launch guaranteeing that the CPU does not continue before the kernel is finished. However, an option to prevent synchronization after a kernel launch is provided by the API for advanced users, and is used for the most benefiting code regions.

The last optimization considered herein is the kernel launch configuration. For optimal computing efficiency and memory coalescing, the number of threads per block should be a multiple of warp size (32 threads for the currently available NVIDIA architectures). In ParFlow, the block size for the x-dimension is currently set to 32 for best memory coalescing, while the block sizes for y- and z-dimensions are dynamically adjusted based on the problem size. However, optimizing the block sizes did not have a significant impact on the performance.

The discussed optimizations are listed below in descending order according to their impact on the whole-program performance. It is also indicated whether the impact was small or large.

- Minimizing data transfers between host and device (large impact)
- Adding efficient parallel reduction kernels (large impact)
- Using a pool allocator for Unified Memory (large impact)
- Coalescing global device memory accesses (large impact)
- Avoiding unnecessary synchronizations (small impact)
- Tweaking with kernel launch configurations (small impact)

5 Performance

If the computational efficiency is the only concern, the best results are usually achieved by rewriting and optimizing a significant portion of the codebase for each desired architecture from scratch. However, this approach often leads to multiple ports and is too expensive and time consuming for most projects. The proposed approach of using an adaptor layer that provides an API for generalized loop types overcomes this problem at the cost of slightly reducing the achievable level of optimization. Nevertheless, the developed GPU support for ParFlow that is discussed in Section 4 shows good performance gains from using GPU accelerators. A representative benchmark problem was run on the booster module[1] of the JUWELS supercomputer where each utilized node is equipped with dual AMD EPYC Rome 7402 processors (2×24 cores @ 2.8 GHz) and 4 NVIDIA A100 40 GB GPUs. The nodes are connected through 4 HDR200-InfiniBand devices. It is expected that more HPC systems in the near future are adopting a design similar to that of JUWELS Booster (around 50 CPU cores and 4 GPU devices per node).

The benchmark consists of a variably saturated infiltration problem into a homogeneous soil with a fixed water table at a depth of 6 m, and a constant infiltration rate of 8×10^{-4} m / hour. The vertical and lateral spatial discretization was 0.025 and 1 m, respectively. The time step size was 1 h. The profile was initialized with a hydrostatic profile based on a matric potential of −9 m at the top resulting in a considerable initial hydrodynamic disequilibrium with respect to the water table at the bottom boundary. The number of grid cells in the lateral directions was varied to change the total number of degrees of freedom in performance testing. The system of equations formed from the nonlinear problem is solved for each iteration at each time step using the GMRES method along with the ParFlow internal multigrid preconditioner. The input file for the benchmark problem is available in the ParFlow repository.[2]

The reference results were obtained without accelerator devices by launching an MPI process for each CPU core. In

[1] JUWELS Booster is based on the Atos XH2000 technology [15].

[2] https://github.com/parflow/parflow/blob/974c7bb98061fbf52bfcce3e0518d48854b28293/test/tcl/clayL.tcl

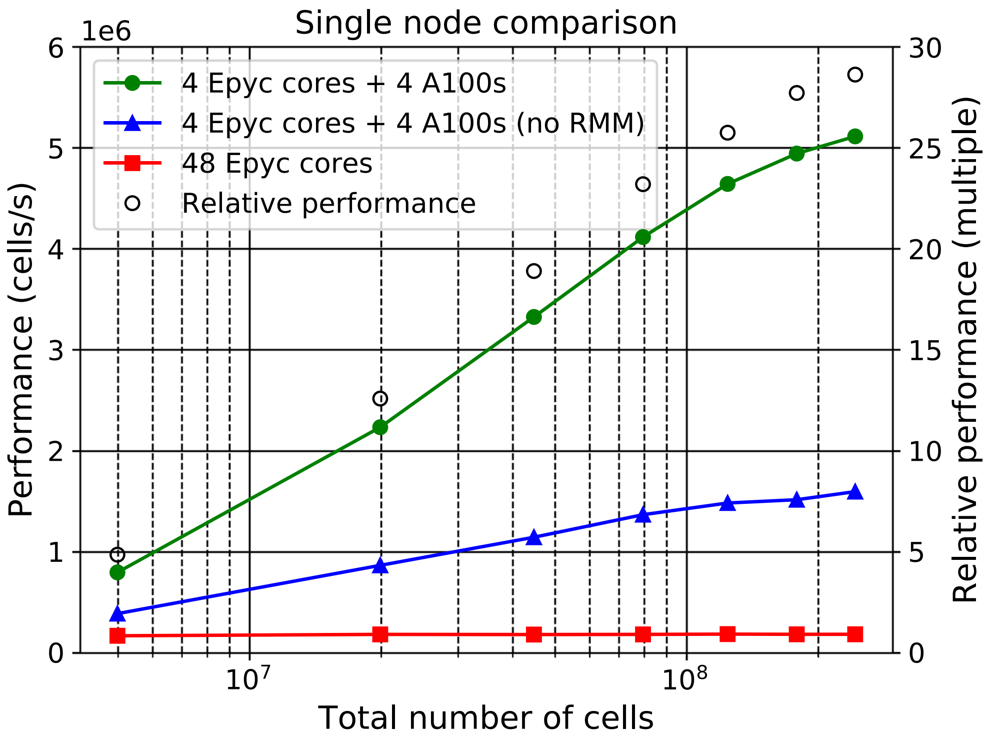

Fig. 2 Single node performance comparison

the case of the accelerated runs, 4 MPI processes per node were launched such that each process uses one GPU and the halo exchange leverages GPU-based application-side data packing (unpacking) before (after) the MPI communication takes place. The simulation results between accelerated and non-accelerated simulations are not bit-identical due to floating-point operations which are conducted in a different order. However, the difference is negligible not only for the presented benchmark problem and several other real-world applications but also for more than one hundred automated tests cases that are run frequently to validate the implementation.

The first test evaluates the impact of using GPU accelerators considering only a single node (Fig. 2). The problem size was varied between $144^2 \times 240$ and $1008^2 \times 240$ cells (horizontal axis). The upper value for the number of cells was limited by the available GPU memory (4×40 GB). The left vertical axis in Fig. 2 represents a performance metric (marked by lines) simply referred to as *performance* which is obtained by dividing the number of cells by a representative timing index. The performance is plotted for GPU accelerated runs with and without the pool allocator (RMM library) for Unified Memory (see Section 4.3). The impact of using the pool allocator increases with the increasing number of cells and more than triples the performance for the largest problem sizes. However, the non-accelerated simulation could also benefit from a pool allocator (although to a much lesser extent), but testing was not possible due to non-existent implementation. The right vertical axis represents the relative performance (marked by circles), i.e., the ratio between the performance metrics of the non-accelerated and accelerated simulations with values >1 indicating faster execution of the accelerated simulation. With the pool allocator, the relative performance increases with the growing number of cells from ~5 (for $144^2 \times 240$ cells) to ~29 (for $1008^2 \times 240$ cells). The plot further suggests that even better performance could be achieved with GPUs providing more memory capacity, although with diminishing returns. Furthermore, the accelerated simulation with $1008^2 \times 240$ cells corresponds to a non-accelerated simulation with less than $196^2 \times 240$ cells when requiring a similar progression speed (i.e. matching the timing indices). This suggests that per-node problem sizes even on the order of 10^8 cells become suitable for real-world use cases when leveraging the latest GPUs with sufficient memory capacity.

In the second test, the weak scaling behavior for 1, 4, 16, 64, and 256 nodes with a fixed number of cells ($1008^2 \times 240$) per node is studied resulting in more than 62 billion cells for 256 nodes (Fig. 3). Again, the left vertical axis denotes the performance (marked by lines), and the right vertical axis the relative performance (marked by circles). The relative performance shows a sharp initial drop when increasing the number of nodes from one to four. The accelerated version splits the domain into four subdomains per node whereas the non-accelerated version splits into 48. In both cases, the domain is split only along the horizontal dimensions of the three-dimensional domain. Therefore, when running on a single node, the halo exchange involves at most two faces of each subdomain for the accelerated simulation as there is no halo exchange along the outer boundary. With the non-accelerated simulation, the domain is split into 48 subdomains and for the interior subdomains that do not

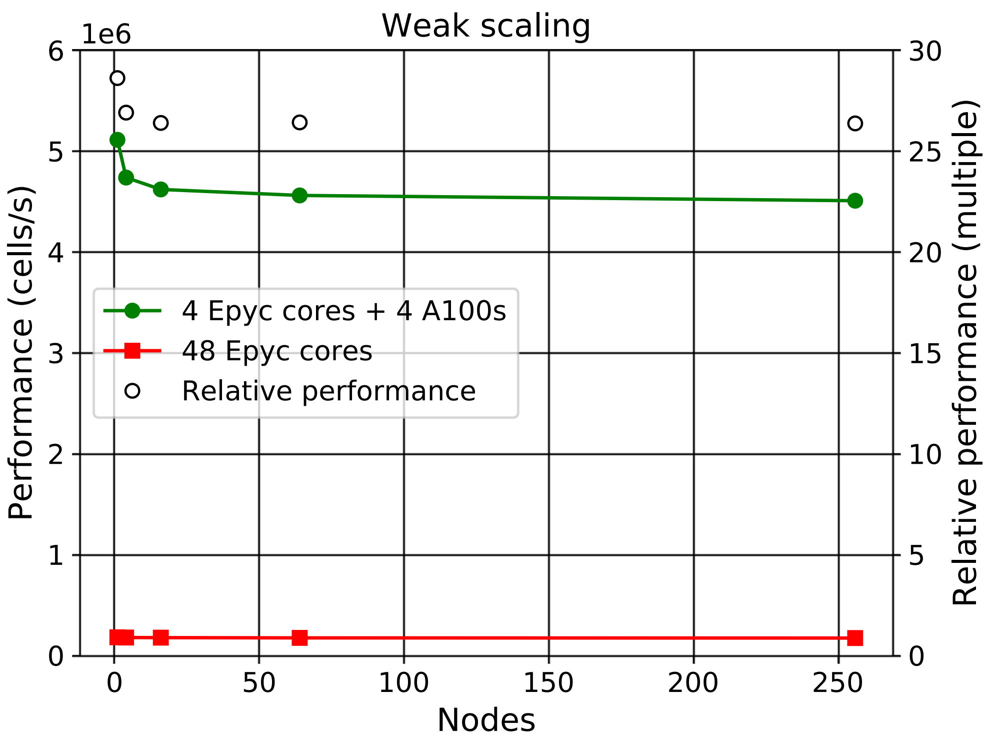

Fig. 3 Weak scaling comparison

contribute to the outer boundary, the halo exchange involves all four faces resulting in proportionally higher communication overhead. However, the GPU accelerated simulation disproportionately benefits from this only when running on a single node. According to Fig. 3, the relative performance is roughly constant between 16 and 256 nodes at ~26 suggesting very good weak scaling performance.

Finally, it is noted that the relative performance of ~26 achieved in the weak scaling study on multiple nodes may represent a more meaningful metric of the performance gain from using GPUs, because the proportionally higher communication overhead in the non-accelerated single-node simulation vanishes when the number of nodes is increased. It is also emphasized that the achievable speedup is highly dependent on the underlying problem, numerical methods, and implementation. Therefore, it is difficult to make accurate generally applicable predictions based on the results from a single project.

6 Conclusions and recommendations

In order for scientific codebases to benefit from the rapidly developing accelerator architectures, an application-side adaptor layer is proposed for accommodating the accelerator programming models or libraries instead of using the third-party interfaces directly throughout the codebase. For example, in a C project this adaptor layer could be based on preprocessor macros, whereas in a project based on C++ the interface could directly leverage more advanced features such as templates, function objects, and lambda functions. As a result, the scientific code can be easily separated from the architecture-dependent code achieving the desired attributes for separation of concerns.

The ParFlow reference implementation employs an adaptor layer which consists of architecture-dependent preprocessor macros for memory management and compute kernels (see Appendix A for a complete Hello World example program with macro definitions for sequential execution on the CPU and parallel execution on the GPU). During the compilation process, the correct set of macros is chosen based on the target architecture. These preprocessor macros form a part of ParFlow embedded domain-specific language (ParFlow eDSL) which is used for many operations such as allocations and initializations, message passing, and looping. However, having a comprehensive domain-specific language already in place is by no means a prerequisite; only a minimal interface for memory management and the relevant compute kernels is often enough to benefit from this approach. Using preprocessor macros results in a good performance and is compatible with lower-level languages such as C, but can obfuscate code for debuggers and compiler diagnostics.

The proposed approach of using an adaptor layer that provides a domain-specific interface for generalized loop types allows less custom optimization for each compute kernel, but at the same time provides a versatile and inexpensive adaptation of new accelerator architectures. In contrast to more general third-party frameworks, a relatively simple scaled-down adaptor layer may suffice as only the features required by the underlying application must be supported. Nevertheless, a good starting point for an adaptor layer for a C++ project can be obtained by studying the interfaces of libraries such as Alpaka, Kokkos, and RAJA with well-proven and thought-out designs; these libraries also serve as ideal backend candidates to target multiple architectures at once.

It is emphasized that a large portion of the development effort is transferred into the development of the adaptor layer which is very likely utilizable with future architectures; this may not be the case with many other approaches if the codebase is built tightly around an accelerator programming model or library that becomes obsolete. The aforementioned claim about utilizability is strongly supported by the ParFlow eDSL of which development started more than 20 years ago, and which now serves as the adaptor layer for modern-day accelerators; the reference implementation discussed in Section 4 demonstrates that a significant performance gain, high developer productivity, and minimally invasive implementation are all achievable at the same time while keeping the codebase well maintainable in the long-term.

The performance evaluation in Section 5 demonstrates good scaling across a large number of nodes with ~26 times increase in the performance from using GPU accelerators. The development effort for the presented ParFlow reference implementation was several months for a single full-time developer with no prior experience of the underlying codebase. However, in a general case, the development time estimate is highly dependent on the underlying codebase, especially the implementation of the data structures and compute kernels. It is concluded on a positive note, and perhaps somewhat contrary to the message of Lawrence et al. [2] that also relatively small developer groups have a good chance of achieving performance portability with their codebase in a reasonable time frame.

Acknowledgements The work described in this paper has received funding from the Helmholtz Association (HGF) through the project "Advanced Earth System Modeling Capacity (ESM) and the Pilot Laboratory Exa-ESM. The authors gratefully acknowledge the computing time granted through the ESM test partition on the supercomputer JUWELS at the Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany. The authors also gratefully acknowledge support from the European Commission Horizon 2020 research and innovation program under Grant Agreement No. 824158 (EoCoE-II). Furthermore, NVIDIA Application Lab at the Jülich Supercomputing Centre is thanked for technical support regarding the CUDA implementation. Finally, the foundations for the ParFlow eDSL were laid by Steven Smith, Rob Falgout, and Chuck Baldwin, all from Lawrence Livermore National Laboratory, USA.

Code availability ParFlow source code is covered by the GNU Lesser General Public License and is available in a public repository at https://github.com/parflow (last access: 27th October 2020). The commit 974c7bb dated 21st October 2020 was used in this paper.

Author contributions Jaro Hokkanen performed the technical developments, analyses, and wrote the manuscript; Stefan Kollet advised on ParFlow technical issues, contributed to the analyses, and co-wrote the manuscript; Jiri Kraus, Andreas Herten, and Markus Hrywniak provided technical support regarding the implementation, optimization, and the HPC environment; Dirk Pleiter contributed to the analyses and the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare that there is no conflict of interest.

Appendix A

A Hello World example program

```
#include <stdio.h>

/* Kernel macro selection (CPU/GPU) */
// #define BoxLoop BoxLoopCPU
#define BoxLoop BoxLoopGPU

/* Compute kernel macro for CPU (API) */
#define BoxLoopCPU(i, nx, loop_body)                      \
{                                                         \
  for (i = 0; i < nx; i++)                                \
  {                                                       \
    loop_body;                                            \
  }                                                       \
}

/* Compute kernel macro for GPU (API) */
#define BoxLoopGPU(i, nx, loop_body)                      \
{                                                         \
  auto lambda = [=] __host__ __device__ (int i)           \
  {                                                       \
    loop_body;                                            \
  };                                                      \
                                                          \
  int blocksize = 1024;                                   \
  int gridsize = (nx - 1 + blocksize) / blocksize;        \
                                                          \
  _BoxKernel<<<gridsize, blocksize>>>(lambda, nx);        \
  cudaStreamSynchronize(0);                               \
  (void)i;                                                \
}

/* General GPU kernel */
template <typename LambdaBody> __global__ static
void _BoxKernel(LambdaBody lambda, const int nx)
{
  const int i =
    blockIdx.x * blockDim.x + threadIdx.x;
  if(i < nx)
  {
    lambda(i);
  }
}

/* Memory allocation macro (API) */
#define alloc_managed(type, count)                        \
  (type*)_alloc_managed(count * sizeof(type));

//* Function to allocate Unified Memory */
static inline void *_alloc_managed(size_t size)
{
  void *ptr = NULL;
  cudaMallocManaged((void**)&ptr, size);
  return ptr;
}

/* Memory deallocation macro (API) */
#define free_managed(ptr) _free_managed_cuda(ptr);

/* Function to deallocate Unified Memory */
static inline void _free_managed_cuda(void *ptr)
{
  cudaFree(ptr);
}
```

```
/* Driver function */
int main(int argc, char *argv [])
{
  int i, nx = 10;
  int* array = alloc_managed(int, nx);

  BoxLoop(i, nx,
  {
    array[i] = i;
  });

  BoxLoop(i, nx,
  {
     int thread = array[i];
     printf("Hello from GPU thread %d\n", thread);
  });

  free_managed(array);
}
```

References

1. PRACE (2018) The scientific case for computing in Europe 2018–2026. Tech. rep
2. Lawrence, B.N., Rezny, M., Budich, R., Bauer, P., Behrens, J., Carter, M., Deconinck, W., Ford, R., Maynard, C., Mullerworth, S., Osuna, C., Porter, A., Serradell, K., Valcke, S., Wedi, N., Wilson, S.: Crossing the chasm: how to develop weather and climate models for next generation computers? Geosci. Model Dev. **11**(5), 1799–1821 (2018). https://doi.org/10.5194/gmd-11-1799-2018, URL https://www.geosci-model-dev.net/11/1799/2018/
3. MPI Forum (1994) MPI: a message-passing interface standard. Tech. rep., University of Tennessee
4. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the top: what will drive computer performance after Moore's law? Science. **368**(6495), eaam9744 (2020). https://doi.org/10.1126/science.aam9744
5. Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., McRae, A.T., Bercea, G.T., Markall, G.R., Kelly, P.H.: Firedrake: automating the finite element method by composing abstractions. ACM Trans. Math. Softw. **43**(3), 1–27 (2016). https://doi.org/10.1145/2998441, URL https://dl.acm.org/doi/10.1145/2998441, 1501.01809
6. Thaler F, Moosbrugger S, Osuna C, Bianco M, Vogt H, Afanasyev A, Mosimann L, Fuhrer O, Schulthess TC, Hoefler T (2019) Porting the COSMO weather model to manycore CPUs. In: proceedings of the platform for advanced scientific computing conference, PASC 2019, Association for Computing Machinery, Inc, New York, NY, USA, pp 1–11, https://doi.org/10.1145/3324989.3325723, URL http://dl.acm.org/doi/10.1145/3324989.3325723
7. Adams, S.V., Ford, R.W., Hambley, M., Hobson, J.M., Kavcic, I., Maynard, C.M., Melvin, T., Mueller, E.H., Mullerworth, S., Porter, A.R., Rezny, M., Shipway, B.J., Wong, R.: LFRic: meeting the challenges of scalability and performance portability in weather and climate models. J. Parallel. Distr. Com. **132**, 383–396 (2018). https://doi.org/10.1016/j.jpdc.2019.02.007, URL http://dx.doi.org/10.1016/j.jpdc.2019.02.007, 1809.07267
8. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knupfer, A., Nagel, W.E., Bussmann, M.: Alpaka - an abstraction library for parallel kernel acceleration. In: proceedings - 2016 IEEE 30th international parallel and distributed processing symposium, IPDPS 2016, Institute of Electrical and Electronics Engineers Inc., pp 631–640. (2016). https://doi.org/10.1109/IPDPSW.2016.50
9. Edwards, H.C., Sunderland, D., Porter, V., Amsler, C., Mish, S.: Manycore performance-portability: Kokkos multidimensional array library. Sci. Program. **20**(2), 89–114 (2012). https://doi.org/10.3233/SPR-2012-0343
10. Beckingsale DA, Scogland TR, Burmark J, Hornung R, Jones H, Killian W, Kunen AJ, Pearce O, Robinson P, Ryujin BS (2019) RAJA: portable performance for large-scale scientific applications. In: Proceedings of P3HPC 2019: International Workshop on Performance, Portability and Productivity in HPC - Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis, Institute of Electrical and Electronics Engineers Inc., pp 71–81, https://doi.org/10.1109/P3HPC49587.2019.00012
11. Kuffour, B.N.O., Engdahl, N.B., Woodward, C.S., Condon, L.E., Kollet, S., Maxwell, R.M.: Simulating coupled surface-subsurface flows with ParFlow v3.5.0: capabilities, applications, and ongoing development of an open-source, massively parallel, integrated hydrologic model. Geosci. Model Dev. **13**(3), 1373–1397 (2020). https://doi.org/10.5194/gmd-13-1373-2020 URL https://www.geosci-model-dev.net/13/1373/2020/
12. Woodward CS (1998) A Newton-Krylov-multigrid solver for variably saturated flow problems. Transactions on Ecology and the Environment 17
13. Kollet, S.J., Maxwell, R.M.: Integrated surface-groundwater flow modeling: a free-surface overland flow boundary condition in a parallel groundwater flow model. Adv. Water Resour. **29**(7), 945–958 (2006). https://doi.org/10.1016/j.advwatres.2005.08.006
14. Maxwell, R.M.: A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling. Adv. Water Resour. **53**, 109–117 (2013). https://doi.org/10.1016/j.advwatres.2012.10.001
15. Pleiter D, Herten A (2020) Enabling applications for the JUWELS booster [A21365]. NVIDIA GPU Technology Conference