Exploiting high-level abstractions to extend capabilities and improve performance of domain-specific languages

Gábor Dániel Balogh

Theses of the PhD Dissertation

Supervisor:

Dr. István Reguly

Pázmány Péter Catholic University

Roska Tamás Doctoral School of Sciences and Technology

Budapest, 2023

1 Introduction

The relentless pace of innovation in processing technology has profoundly impacted how programmers and scientists achieve high performance. Where twenty years ago, a few gigaflops might reside within a single CPU, today's most powerful supercomputers deliver trillions of calculations per second across hundreds of thousands of hardware threads. Keeping pace with this exponential growth has required fundamental changes in both hardware and software.

In the early 2000s, performance revolved around optimizing for single superscalar CPUs. Programmers leveraged techniques like loop unrolling, prefetching, and tiling to minimize latency and maximize instruction throughput. With the emergence of multi-core chips, a new dimension of parallelism demanded algorithms designed for concurrent execution. Shared memory and locks became tantamount, necessitating careful orchestration to avoid contention bottlenecks.

Graphics processors debuted their immense streaming capabilities, recasting approximation problems like rendering and machine learning as highly data-parallel workloads. This transformed programming to follow a data-parallel mindset, abandoning instruction-centric CPU models for throughput-oriented many-core architectures. Libraries arose abstracting GPU kernels through frameworks like OpenCL and CUDA, easing porting efforts.

The use of domain-specific languages (DSLs) has seen growing adoption in technical computing domains as they offer substantial productivity gains over general-purpose languages. A DSL defines a high-level abstraction providing a custom language or embedded language to describe the problem to be solved in terms specific to the problem domain. By capturing common patterns and structures within the domain, a DSL allows programmers familiar with the domain to express solutions in a natural way using domain terms and abstractions, raising the level of abstraction compared to a general-purpose language. By distilling

expertise within a domain into abstract syntax and semantics, DSLs allow domain experts rather than language experts to write programs. As such, DSLs provide a middle ground between generality and specificity that has proven useful across diverse fields ranging from cellular biology to aerospace engineering simulation. Embedded DSLs or eDSLs act as a conventional library in a general-purpose language such as C++ or Fortran for the application developer and use source-to-source transformation techniques to provide target-specific performant implementations. This allows the DSL to provide capabilities tailored to the domain while reusing the existing compiler and runtime infrastructure of a general-purpose language.

The first exascale supercomputers appeared in the past few years using trillions of simultaneous threads. No longer can a single device deliver such prowess. The distributed agenda demands wide-area networking to integrate clustered systems. Meanwhile, hardware diversity proliferates as accelerator technologies like FPGAs join the tussle for exaflops. More than ever, high performance hinges on software's ability to portably extract inherent parallelism while skirting architectural idiosyncrasies.

Due to the rapidly changing hardware and programming models that run the most powerful computers in the world, performance portability and productivity became the focus point of any discussion on future-proof high-performance software. In this ever-changing landscape, future-proof applications have become synonymous with performance portable applications where the ultimate dream is supporting all current and future hardware with the best performance from a single source [1]. Domain-specific languages and declarative programming hold promise, raising abstraction to separate concerns of performance and the description of the computations.

In parallel and high-performance computing (HPC), DSLs have shown particular promise in addressing two key challenges. Firstly, they facilitate performance portability through architecture-agnostic problem

descriptions that abstract away low-level processor details. Effectively allowing the application developer to describe the problem to be computed instead of how the problem should be computed. Secondly, they aid application productivity by raising the level of abstraction for algorithm expression while retaining control over optimization opportunities. Together, these advantages have driven extensive research into DSLs for HPC applications over the past years.

One such domain-specific language family is the Oxford Parallel Domain-Specific Languages, consisting of two active libraries or embedded DSLs OPS [2] and OP2 [3]. These libraries provide DSL abstractions targeted at partial differential equation (PDE) solvers embedded in C/C++ and Fortran. Domain experts author PDE solvers within these DSLs through high-level parallel loops, expressing computations through element kernels while abstracting away parallel execution details and data motion. During compilation, the libraries use a code generation step to generate target-specific parallel implementations for the parallel loops, applying a wide range of optimizations.

My main motivation is to further develop the possibilities offered by domain-specific languages, thus making the performance portability and productivity available in new areas and problem classes. My research focuses on computations on structured and unstructured meshes. The aim of my dissertation is to present my results in extending the problem classes supported by DSLs and improving the robustness and extensibility of the code generation steps used by DSLs.

The first part of my dissertation focuses on source-to-source transformation techniques in DSLs or active libraries. The extensibility and complexity of code generation steps are critical for the longevity of DSLs. Starting from the domain-specific model, generating the same structure or code takes a significant portion of the generated code, and formulating the generation is error-prone. In my first thesis, I concentrated on the optimal mapping of calculations on unstructured grids to heterogeneous hardware. I present the method of parallelization skeleton-based

generation on the OP2 DSL and compare the performance of various programming models and languages (Thesis I).

After my work on the code generation step of DSLs, I worked on extending the support for structured mesh applications. The second and third thesis focus on extending the OPS DSL. In the second part, I supplemented the OPS DSL with support for a performance portable and scalable linear solver library suitable for the Alternating-Direction Implicit method [4]. ADI applications are an important special case of structured mesh computations, where, in addition to stencil calculations, solutions are approximated with the help of linear solvers. Thus, to support ADI applications, support for batch-tridiagonal solver libraries is also needed. A key feature of DSLs is generality to provide support for an entire domain with the narrowest abstraction possible. However, in the case of batch-tridiagonal solver support, the abstraction is narrow enough so we can provide direct library support for CPU and GPU clusters. The second part deals with the challenges of MPI-scalable solution algorithms for batch-tridiagonal solvers through localizing and minimizing the necessary communication to find the exact solution and improving the communication and scaling properties of exact and iterative solvers (Thesis II.).

However, many scientific computing applications require not just a PDE solution but also the sensitivity information of the outputs with respect to some input variables. Algorithmic differentiation (AD) is recognized as a key enabling technique for uncertainty quantification and design optimization through its ability to automatically and accurately compute derivatives of computer programs by exploiting the mathematics of function composition. However, performance portable solutions for AD are lacking, and no library is applicable to DSLs like OPS. Furthermore, these tools do not consider the performance-critical optimizations demanded on modern massively parallel processors typified by many-core GPU accelerators for industrially relevant applications. The third part of my research focuses on extending OPS with reverse-mode AD support,

automatically mapping high-level structured mesh code to multi-core CPUs and many-core GPUs (Thesis III.1 and Thesis III.2), with additional support for external tools like linear solvers from Thesis II. and finally, with an abstraction to control the memory overhead of the differentiation with checkpointing and recomputing (Thesis III.3.). I show performance results and the main contributing factors of overheads and the advantages of using the domain-specific information during orchestration of the adjoint loops on multiple representative applications.

2 Methods and tools

The first part of my research is based on the OP2 domain-specific language [3], which provides a high-level abstraction for the solution of unstructured-mesh applications defining an API to describe computational kernels with all the necessary information for orchestrating parallelism. The implementation of the new source-to-source generator is based on the refactoring tool support of Clang's LibTooling library. The correctness and performance of the generated code were tested on two applications written using the OP2 abstraction: a benchmark simulating the airflow around the wing of an aircraft called Airfoil and a tsunami simulation software called Volna [5].

The second part of my dissertation focuses on the distributed solution of batch-tridiagonal systems with special attention to applications using the Alternating-Direction Implicit method [4]. The base of our work is the single-node solver library Tridsolver [6]. We combined the exact iterative PCR algorithm [7] with the distributed communication strategies of the TridiagLU library [8].

Finally, the third part of my research focuses on the other domain-specific language of the OP-DSL family OPS [2] and computing sensitivity information for outputs. The structure and abstraction of OPS are similar to OP2's, but OPS targets structured meshes with stencil loops. We applied reverse (adjoint) mode algorithmic differentiation (AAD) to

applications written in OPS. We used three applications to analyze the performance of AAD with OPS: a 2D code solving the Poisson equation using Jacobi iterations applying AAD directly and using fixed point iterations [9], the CloverLeaf[10] is a mini-app that solves the compressible Euler equations on a Cartesian grid using an explicit, second-order accurate method and a code for 2D convection-diffusion equation code from computational finance [11].

The applications and library extensions were implemented in C++ in combination with the CUDA language extension for targeting GPUs. While the code generator for OP2 was implemented in C++, the code generator for the AAD support of OPS was implemented in Python. I used the NCCL and MPI libraries for message passing for distributed memory parallelism, and for shared memory parallelism, I used OpenMP and CUDA.

For performance measurements, a range of hardware architectures and platforms were used. For benchmarking the scaling performance of the Tridsolver library we used two of the UK's HPC systems: ARCHER2[1], a CrayEX system with AMD Rome CPUs (2×64 cores per node) and 256 GB of RAM, and Cirrus[2], a HPE/SGI system with 36 GPU nodes, each with 4×NVIDIA V100 16GB GPUs, interconnected with NVLink, and FDR Infiniband between nodes. For evaluating the performance of AAD in OPS, we ran the OpenMP measurements on a single socket of an Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading and 376 GB RAM and the CUDA measurements were executed using an AMD EPYC 7F72 24-core Processor and an NVidia A100 GPU with 40 GB RAM. In most cases, the runtimes reported are the results of averaging 10 repeated runs.

[1] https://www.archer2.ac.uk/
[2] https://www.cirrus.ac.uk/

3 New Scientific Results

Thesis I.

I designed an automatic translation toolchain that uses a parallelization skeleton based approach. My solution improves the stability and robustness of source-to-source translation in the OP2 DSL, generating code for CPU clusters and GPUs, and improving memory locality. The performance of the generated code is demonstrated on a set of representative applications, and I performed a comparative analysis of the effects of various programming languages and compilers on the efficiency of parallel loops.

Publications related to this thesis: [J1, C3, C4]

The OP2 API was constructed to make it easy for a parsing phase to extract the relevant information about each loop that will describe which computation and memory access patterns will be used - this is required for code generation aimed at different architectures and parallelization. The `op_arg_dat` provides all the details of how an `op_dat`'s data is accessed in the loop. With this information, the `op_par_loop` call contains all the necessary information about the computational loop to perform the parallelization. It is clear that due to the abstraction, the parallelization depends only on a handful of parameters, such as the existence of indirectly accessed data or reductions in the loop, plus the data access modes that lend to optimizations.

The fact that only a few parameters define the parallelization means that in the case of two computational loops, the generated parallel loops have the same lines of code with only small code sections with divergences. The identical chunks of code in the generated parallel loops as an important blueprint of the target code to be generated. This leads us to the idea of using a parallel implementation (with the invariant chunks) of a dummy loop and carrying out the code generation process as a refactoring or modification of this parallel loop. Figure 1 illustrates partial

parallel skeletons we can extract for the generated OpenMP implementation for indirect loops. The code generator can use this dummy parallel loop as a skeleton (or template) and modify it to generate the required candidate computational loop. One can imagine similar skeletons for all target parallelizations. This approach can reduce the cost of introducing new targets since it requires only to implementation of a dummy loop to use as a skeleton instead of implementing code generation paths for the whole kernel. At the same time, since implementing a loop is much less error-prone than writing code to generate it, this approach reduces the risk of introducing bugs in the invariant bits of code in the parallel loops.

Based on the parallelization skeletons, the code generation for a parallel loop can be considered as a refactoring step. Clang's LibTooling library provides great support for code refactoring tasks by matching specific parts of the code's Abstract Syntax Tree (AST) and modifying the source code behind the matched nodes. To complete the whole process of source-to-source transformation in OP2, the code generator requires two steps. The first collects data about the parallel loops to be generated and replaces the original `op_par_loop` calls with calls to the generated functions, and the second is generating code for the target hardware as shown in Figure 2. The first phase parses all the arguments in the `op_par_loop` calls to collect all the information that is required to fill in the loop-specific code in the parallelization skeleton. The second phase will choose the appropriate skeleton, build the AST, and perform a set of refactoring steps, such as changing the function signature, to generate the final specialized loop implementation. This approach provides two advantages over the conventional Python code generator. The first is that the code generator can easily reuse bits of refactoring steps between target structures, making extensions for new targets easier. The second is that by using the compiler infrastructure, the new code generator can provide more sophisticated semantic checks over the generated code at the time of the translation.

```
// elemental kernel function
void skeleton(double * __restrict__ d) {}

void op_par_loop_skeleton(char const *name,
                          op_set set,
                          op_arg arg0) {
  //number of arguments
  int nargs = 1; op_arg args[1] = {arg0};
  int ninds = 1; op_arg inds[1] = {0};

  /*----------- Invariant code -----------------*/
  int set_size =
      op_mpi_halo_exchanges(set, nargs, args);
  op_plan *Plan = op_plan_get(name, set, 256, nargs,
                              args, ninds, inds);
  int block_offset = 0;
  for (int col = 0; col < Plan->ncolors; col++) {
    if (col == Plan->ncolors_core)
      op_mpi_wait_all(nargs, args);
    int nblocks = Plan->ncolblk[col];
    #pragma omp parallel for
    for(int blockIdx = 0; blockIdx<nblocks;
        blockIdx++) {
      int blockId =
          Plan->blkmap[blockIdx +block_offset];
      int nelem = Plan->nelems[blockId];
      int offset_b = Plan->offset[blockId];
      for(int n = offset_b; n<offset_b+nelem; n++) {
  /*-------------------------------------------*/
        // Prepare indirect accesses
        int map0idx =
            arg0.map_data[n * arg0.map->dim + 0];
        // set up pointers, call elemental kernel
        skeleton(&((double *)arg0.data)[2*map0idx]);
      }
    }
  }
```

Figure 1: Skeleton for OpenMP (excerpt) - indirect kernels

Thesis II.

I designed a set of novel high-performance, scalable, distributed memory algorithms for the solution of batch-tridiagonal systems of equations, targeting large-scale heterogeneous supercomputers based on modern multi-core and many-core processor architectures. My algorithms can compute both the approximate and the exact solution of individual systems, and seamlessly integrates with Alternating-Direction Implicit methods commonly used in the solution of large-scale high-dimensional

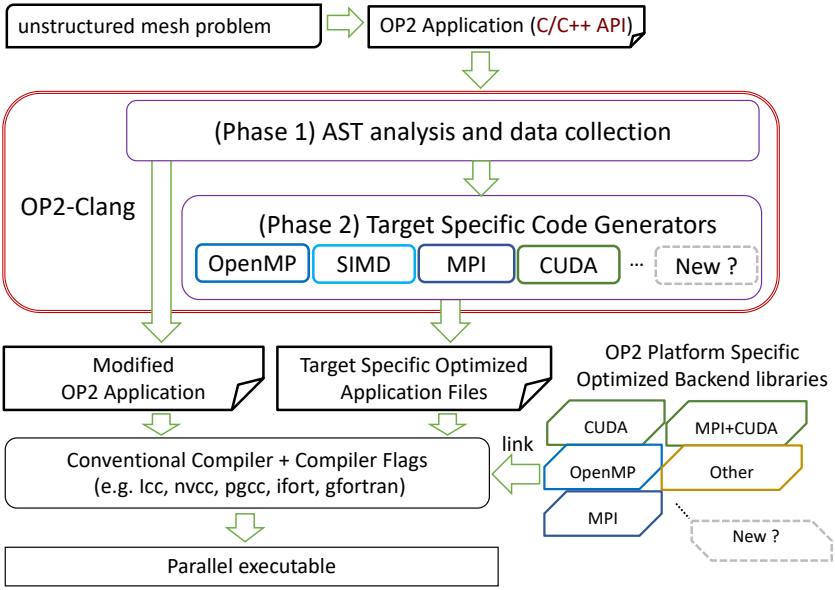


Figure 2: The high-level architecture of OP2-Clang and its place within OP2

Partial Differential Equations. I published my implementation as an extension to the open-source Tridsolver library.

Publications related to this thesis: [J2]

The state-of-the-art distributed memory algorithms for tridiagonal systems divide the system into subsystems and form a smaller decoupled tridiagonal system connecting the partitions (reduced system). Then, commonly, this reduced system is either gathered into a single process to solve and then the solution is scattered among the processes or solved via iterative solver algorithms like Jacobi iterations. The former scales poorly due to the all-to-all communication patterns, and the latter, while using only point-to-point communications, produces approximate solutions.

In ADI, the coefficients are calculated for each grid point in a way that matches the underlying data structure of the application. MPI nodes are defined along all dimensions and data for the diagonals are stored con-

tiguously in either a row-major (Z is contiguous, Y, and X are strided) or, more commonly, a column-major (X is contiguous, Y and Z are strided) format. This poses a challenge for algorithms that then solve multiple tridiagonal systems simultaneously; the different directions will use different memory layouts, which in turn require different optimizations. Moreover, improving on the state-of-the-art, our library supports all of the three different memory layouts possible for 3 or higher-dimensional problems.

By extending the Thomas-PCR hybrid algorithm to distributed memory environments, I designed a tridiagonal solver algorithm that gives exact solutions for batch tridiagonal problems while retaining the scaling properties of the approximate algorithms. The overall structure of the distributed tridiagonal solver can be summarized as follows. Each subsystem of size M belongs to a separate MPI process, which performs the hybrid Thomas-PCR forward pass. This produces a reduced system with two rows per MPI process. The solution to the reduced system is implemented using the distributed PCR algorithm. This algorithm uses only point-to-point communications, which is a crucial criterion for scalability. Once the reduced system is solved, the backward pass of the hybrid Thomas-PCR is performed on each MPI process.

Table 1: Comparison of communications in distributed solver algorithms.

	Accuracy	Communication pattern	number of messages	message size
Allgather	exact	all-to-all	1	$3 \times 2 \times N_{proc} \times N_{sys}$
Jacobi	approximate	local point-to-point, all-to-all for error	$2 \times N_{iterations}$	N_{sys}
PCR	exact	point-to-point with increasing distance	$2 \times log_2(N_{proc})$	$3 \times N_{sys}$

Table 1 shows a comparison between the three major solving strategies for the reduced system and the trade-offs for using them, where N_{proc} marks the number of MPI processes and N_{sys} marks the batch size. To create a scalable algorithm, it is critical to avoid all-to-all communications, which would lead to message size correlating with the number of processes. For the algorithms using point-to-point communications,

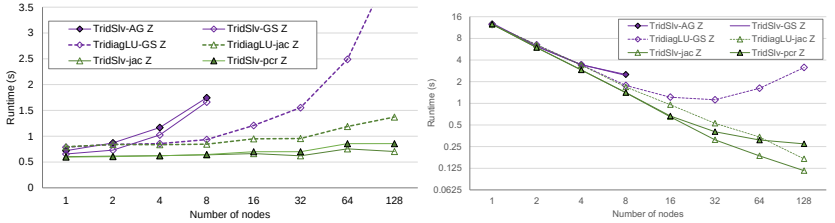
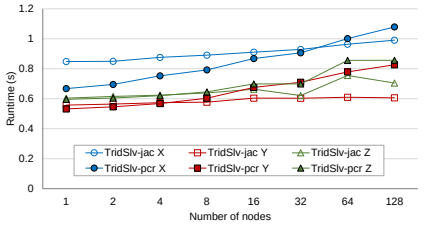


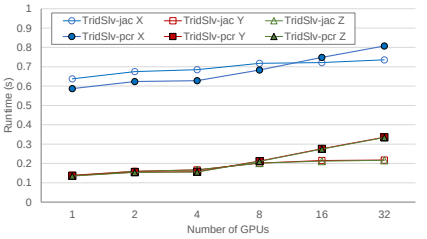
Figure 3: Comparison of the Tridsolver library to TridiagLU. **Left:** weak-scaling 512^3 grid points per node, **Right:** Strong-scaling, 8192 points in the direction of solve, and 512 in others. AG - AllGather, GS - Gather-Scatter

the number of messages and the distance between the communicating nodes affect the overhead of the communication. We can see that the PCR algorithm will use bigger messages between nodes that are further away from each other, but in return, it does not require any global communication and produces exact solutions.

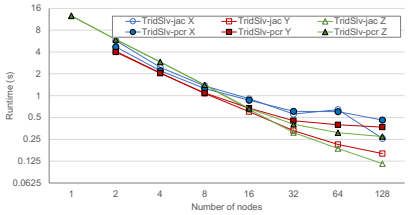
Two of the three steps in the hybrid algorithm scales trivially. The only part that contributes to the scaling properties is the distributed solver for the reduced system. Figure 3 shows that algorithms relying on global communication collectives take over the runtime after a certain point. For point-to-point communications, the message size and the distance of the nodes that are required to communicate are the two factors defining the overhead. In the results shown in Figure 3, we used a problem-specific upper bound on the number of Jacobi iterations instead of using global reduce calls to compute the error; hence, the Jacobi iteration used a fixed number of communications, and each node communicated only with neighboring nodes, but such an upper bound can't be defined for the general case. On the other hand, PCR has one additional communication step at each data point on the figure but does not need any problem-specific heuristics to compute the solution. The increasing cost of the communication clearly shows in the case of strong scaling after 16 MPI nodes, where the cost of the far messages (leaving



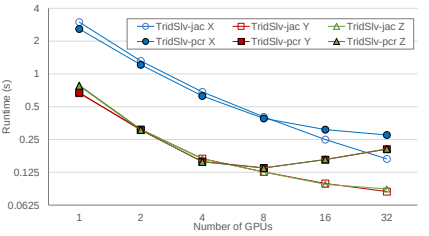
(a) Tridsolver weak scaling on ARCHER2



(b) Tridsolver weak scaling on Cirrus



(c) Tridsolver strong scaling on ARCHER2



(d) Tridsolver strong scaling on Cirrus

Figure 4: ARCHER2 scaling (MPI+OpenMP): (a),(c) Cirrus scaling (MPI+CUDA):(b),(d) - All weak-scaling using 512^3 points per node. Strong scaling on ARCHER2 uses 8192 points in the direction of solve while Cirrus measurements use 2048 points and 512 points in others.

local memory of ARCHER2 nodes) dominates the total runtime while still beating global communication patterns significantly.

Figure 4 shows the scaling performance of the Tridsolver library for solver calls in all directions of a 3D application on ARCHER2 and on the Cirrus system. On CPUs, the PCR version achieves 70% scaling efficiency up to 128 nodes, while on GPUs, the cost of the communication outside a single Cirrus node (4 GPUs) is significantly higher due to slower interconnect, which has a great impact on the scaling of the PCR solver.

Thesis III.

I proposed an advanced, abstract computational model for reverse mode algorithmic differentiation of complex stencil applications and in-

tegrated it into the OPS domain-specific language. The model enables OPS to generate multi-core CPU and massively parallel GPU implementations for the adjoint loops, leveraging the metadata provided by the DSL. The model uses a new mapping of the algorithm to novel execution patterns for the adjoint loops. Furthermore, the extension enables OPS to follow the computational steps at a loop level by integrating an AD tape tailored for the OPS DSL, creating a streamlined storage mechanism thanks to the OPS abstractions.

Publications related to this thesis: [J3, C2]

Subthesis III.1. - I created a computational model based on the OPS abstraction for structured-mesh stencil applications that describes computational patterns, data, and control flow and described how adjoint mode Algorithmic Differentiation can be performed with this model. I extended the OPS abstraction to handle AD active datatypes and code regions with custom adjoint functions such as linear solvers.

Computing sensitivities efficiently is crucial in many areas, and getting efficient parallel implementations of the gradient propagation in reverse mode AD is especially challenging. The two key challenges of reverse mode AD in a parallel environment are the data races introduced by the reversal of the access patterns and following the control flow at runtime. The OPS API uses a description of loops and the data access inside the loops to generate efficient parallel implementations. Using this description, I created a model that describes the access patterns, describing the potential data races for the loops executing the adjoints of the loops, enabling the generation of parallel implementations. Building on the loop chain registered at run-time, this model enables OPS to compute the gradient through reverse-mode AD.

Subthesis III.2. - I designed and implemented a mapping of the high-level model to optimized, low-level parallel computational kernels supporting both multi-core CPUs and many-core GPUs with architecture-specific optimizations.

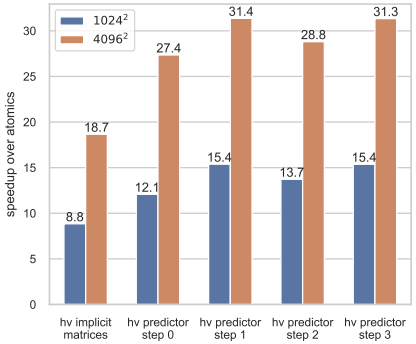


Figure 5: Speedup of the version using the access pattern aware reductions over atomic operations in kernels for the CDE application with a mesh size of 1024^2 and 4096^2.

In an OPS loop, each dataset is accessed through a stencil with an assigned access pattern: read, write, or increment. All loops in OPS must use gather stencils only, meaning that read can appear with any stencil with any number of points, but increment and write stencils must have one single point access with zero offsets. In Reverse mode AD in the adjoint loops, the data flow will be reversed from gathering stencils, and we will get scatter operations on the adjoint data. Due to the reversal, the write and increment stencils on the datasets will turn into one-point read stencils on the adjoint data and read stencils will turn into increment stencils with multiple points. The above has two implications: first, the primal loops are race-free, the parallelization is trivial, and second, the adjoint loops will have data races on the adjoint data. However, the location and structure of these data races are defined by the read stencils of the primal loops. I devised and implemented an execution pattern that avoids race conditions on CPUs executing the loop in two sweeps with synchronization between. This approach avoids the cost of atomic operations. On GPUs, the cost of atomic operations is lower; hence, the adjoint kernels are using atomic operations on the derivatives.

Another important access pattern in terms of performance is read-

ing to lower dimensional data (data that is invariant in some dimension). These accesses will result in a large amount of writes on the same derivative values in the backward pass. I introduced a specialized code generation path for lower-dimensional datasets in CUDA adjoint kernels using reductions in only the required dimensions. Figure 5 shows the speedup gain on an NVidia A100 using this optimization.

Subthesis III.3. - I extended the OPS abstraction to integrate the model and the mapping, automatically orchestrating the forward and adjoint computations, including control over the memory overhead of differentiation and enabling the efficient parallelization of the gradient computation. I demonstrated the utility and performance of this extension on industrially representative applications.

Reverse mode AD tools can only follow the control flow at most an expression level, leading to high memory usage. Building on the OPS abstraction, I introduced a tape data structure that keeps track of the parallel loop descriptors and stores overwritten data for the loops. The high-level tape drastically reduces the memory overhead of storing control flow information. However, storing all overwritten data for a large number of iterations in an application would lead to huge tape. To address this issue, I extended the tape with the Revolve[12] checkpointing strategy, providing the user with fine control over the memory usage of the adjoint computation using loop re-execution to recompute the intermediate states.

I evaluated the performance of the gradient computation on three industrially representative applications. Figure 6 shows the relative runtime of the whole gradient computation(including the evaluation of the original function) compared to a single evaluation of the original application.

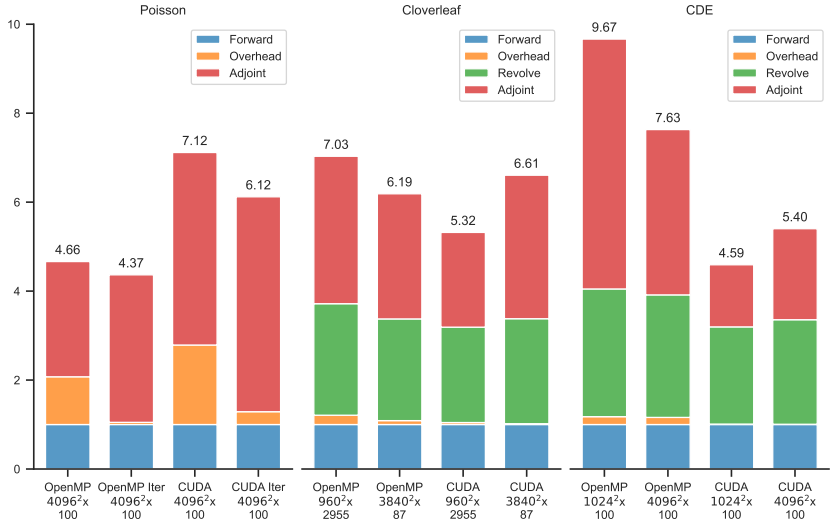


Figure 6: The overhead of AD on the benchmark applications, compared to a single evaluation of the passive, original application. The values show that evaluating the gradient takes N times of evaluating the original application. The **Overhead** values show the additional cost of collecting the DAG, saving intermediate states, and storing revolve checkpoints during the forward pass, **Revolve** values represent the additional time spent on replaying sections of the applications to restore states for the adjoints loops and the **Adjoint** part shows the time spent in the actual adjoint loops.

4 Potential applications and benefits

My work on the OP2-Clang tool was directly applicable as the source-to-source translation layer of the OP2 DSL. The use of a compiler-based tool could increase the robustness and diagnostic abilities of the code generation and make integration into industrial build systems easier at the same time.

Results carried out in the context of batch-tridiagonal solver libraries can be used in large-scale scientific applications using the ADI method. This research was performed partially as part of the UK's ExCALIBUR project, which aims to deliver the next generation of high-performance simulation software. As part of the project, a discussion of the use of the library in the xCompact3D library [13] is ongoing. This library is an industrial strength library for simulating turbulent flows and is used for simulations such as airflow around an entire wind farm.

Adjoint mode Algorithmic Differentiation is often used in computational fluid dynamics and computational finance. Our results show how domain-specific languages or abstractions, in particular OPS, can drastically reduce memory overhead and improve the runtime of computing gradients compared to general-purpose tools. The high-level OPS application code provides support for both CPUs and GPUs, which makes OPS one of the first performance portable adjoint mode AD libraries. Furthermore, with OPS's support for AD completed, we plan to enable this functionality in higher-level libraries building upon OPS, such as the Navier-Stokes solver OpenSBLI library, which focuses on shocks and boundary layer interactions.

5 Acknowledgments

First of all, I would like to express my gratitude to my supervisor Dr. István Reguly, for introducing me to the world of High-Performance Computing, providing me with countless opportunities, and for his im-

mense support, guidance, and patience that led me through these years. His positive and supportive attitude helped me a tremendous amount during the past years. I am deeply grateful to Dr. Gihan Mudalige for welcoming me into his research group during my stay in Warwick and for his help and insight during our collaboration. I am also grateful to Jacques du Toit, who generously provided knowledge and expertise. I would like to thank Prof. Uwe Neumann and Dr. Johannes Lotz for the opportunity to learn from them during my stay at RWTH Aachen University.

I would like to thank all my friends and colleagues for filling these past few years with laughter and joy. I would like to thank András Attila Sulyok, Bálint Siklósi, Bence Horváth-Keömley, Tamás Rudner, Mihály Vághy, and many others for the myriad thought-provoking discussions we've shared that have been a constant source of inspiration and growth.

I am thankful to the Pázmány Péter Catholic University, Faculty of Information Technology, especially to Prof. Gábor Szederkényi and Prof. Péter Szolgay for the opportunity to participate in the doctoral program and for supporting me throughout. I am especially thankful to Dr. Tivadarné Vida for her kind help during my doctoral studies.

Finally, I am most grateful to my family for their limitless support and especially to Zsófia Balogh-Lantos for tolerating me and helping me through the difficult times.

Publication of the author

Journal publications

[J1] A. A. Sulyok, **G. D. Balogh**, I. Z. Reguly, and G. R. Mudalige, “Locality optimized unstructured mesh algorithms on GPUs", *Journal of Parallel and Distributed Computing*, vol. 134, pp. 50–64, 2019 doi: 10.1016/j.jpdc.2019.07.011.

[J2] G. D. Balogh, T. S. Flynn, S. Laizet, G. R. Mudalige and I. Z. Reguly, "Scalable Many-Core Algorithms for Tridiagonal Solvers", in *Computing in Science & Engineering*, vol. 24, no. 1, pp. 26-35, 1 Jan.-Feb. 2022, doi: 10.1109/MCSE.2021.3130544.

[J3] G. D. Balogh, J. Lotz, J. Du Toit, U. Naumann, and I. Reguly, "Performance portable adjoints for structured mesh applications with OPS", in *ACM Trans. Math. Softw.*, **under submission**.

Conference publications

[C1] G. D. Balogh and I. Reguly, "Automatic Parallelisation of Sturctured Mesh Computations with SYCL," *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, Portland, OR, USA, 2021, pp. 821-822, doi: 10.1109/Cluster48925.2021.00083.

[C2] G. D. Balogh and I. Z. Reguly, "Automatic parallel implementations of adjoint codes for structured mesh applications," *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, Melbourne, VIC, Australia, 2020, pp. 908-911, doi: 10.1109/CCGrid49817.2020.00019.

[C3] G. D. Balogh, G. R. Mudalige, I. Z. Reguly, S. F. Antao and C. Bertolli, "OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling," *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, Dallas, TX, USA,

2018, pp. 59-70, doi: 10.1109/LLVM-HPC.2018.8639205.

[C4] G. D. Balogh, I. Z. Reguly and G. R. Mudalige, "Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs", *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, vol. 10724, pp. 22–43, 2017, doi: 10.1007/978-3-319-72971-8_2

References

[1] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Navigating performance, portability, and productivity," *Computing in Science & Engineering*, vol. 23, no. 5, pp. 28–38, 2021.

[2] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2018.

[3] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–12, 2012.

[4] D. W. Peaceman and H. H. Rachford, Jr, "The numerical solution of parabolic and elliptic differential equations," *Journal of the Society for industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, 1955.

[5] D. Dutykh, R. Poncet, and F. Dias, "The volna code for the numerical modeling of tsunami waves: Generation, propagation and inundation," *European Journal of Mechanics - B/Fluids*, vol. 30, no. 6, pp. 598–615, 2011. Special Issue: Nearshore Hydrodynamics.

[6] E. Laszlo, M. Giles, and J. Appleyard, "Manycore algorithms for batch scalar and block tridiagonal solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 1–36, 2016.

[7] W. Gander and G. H. Golub, "Cyclic reduction—history and applications," *Scientific computing (Hong Kong, 1997)*, vol. 7385, pp. 73–86, 1997.

[8] D. Ghosh, E. M. Constantinescu, and J. Brown, "Efficient implementation of nonlinear compact schemes on massively parallel platforms," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C354–C383, 2015.

[9] B. Christianson, "Reverse accumulation and attractive fixed points," *Optimization Methods and Software*, vol. 3, no. 4, pp. 311–326, 1994.

[10] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, vol. 2013, 2013.

[11] M. Wyns and J. Du Toit, "A finite volume–alternating direction implicit approach for the calibration of stochastic local volatility models," *International Journal of Computer Mathematics*, vol. 94, no. 11, pp. 2239–2267, 2017.

[12] A. Griewank and A. Walther, "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 1, pp. 19–45, 2000.

[13] P. Bartholomew, G. Deskos, R. A. Frantz, F. N. Schuch, E. Lamballais, and S. Laizet, "Xcompact3d: An open-source framework for solving turbulence problems on a cartesian mesh," *SoftwareX*, vol. 12, p. 100550, 2020.