Compiler Abstractions and Runtime for Extreme-scale SAR and CFD Workloads

Connor Imes
USC Information Sciences Institute
cimes@isi.edu

Alexei Colin
USC Information Sciences Institute
acolin@isi.edu

Naifeng Zhang
University of Southern California
naifengz@usc.edu

Ajitesh Srivastava
University of Southern California
ajiteshs@usc.edu

Viktor Prasanna
University of Southern California
prasanna@usc.edu

John Paul Walters
USC Information Sciences Institute
jwalters@isi.edu

***Abstract*—As HPC progresses toward exascale, writing applications that are highly efficient, portable, and support programmer productivity is becoming more challenging than ever. The growing scale, diversity, and heterogeneity in compute platforms increases the burden on software to efficiently use available distributed parallel resources. This burden has fallen on developers who, increasingly, are experts in application domains rather than traditional computer scientists and engineers. We propose CASPER—Compiler Abstractions Supporting high Performance on Extreme-scale Resources—a novel domain-specific compiler and runtime framework to enable domain scientists to achieve high performance and scalability on complex HPC systems. CASPER extends domain-specific languages with machine learning to map software tasks to distributed, heterogeneous resources, and provides a runtime framework to support a variety of adaptive optimizations in dynamic environments. This paper presents an initial design and analysis of CASPER for *synthetic aperture radar* and *computational fluid dynamics* domains.**

***Index Terms*—parallel programming, runtime, dynamic compiler, high performance computing, synthetic aperture radar, fluid dynamics**

I. Introduction

As high performance computing (HPC) platforms evolve, several trends present challenges for application developers:

- *Scale* – next-generation HPC systems are targeting applications with million-way parallelism [6].
- *Diversity* – HPC platforms use a variety of microarchitectures, e.g., as of June 2020, the top 10 HPC systems include processor designs from Intel, AMD, ARM, Sunway, and IBM, and span multiple processor generations [32].
- *Heterogeneity* – HPC nodes contain multiple compute element types such as CPUs, GPUs, and FPGAs, and more complex memory and communication hierarchies [30].
- *Dynamics* – some runtime operating conditions are not known a priori, and may even change during application execution, e.g., due to manufacturing variability, shared resource contention, and workload imbalances [15, 17].

To run *efficiently*, HPC applications must optimally balance data partitioning, computation, storage, I/O, and runtime dynamics in increasingly complex systems. The classical approach is to redesign or even entirely rewrite software for new platforms, resulting in artifacts that lack good *portability* to other systems. Additionally, this optimization effort is made at the expense of developing new capabilities or applications, negatively impacting programmer *productivity*.

Domain-specific languages (DSLs) have been shown to be both efficient and productive [8]. Programmers specify applications as compositions of kernels while the language manages the data partitioning, parallelization, in-memory structures, communication, and synchronization. Finding the best task mappings to diverse and heterogeneous resources is a challenging problem, but the higher abstraction levels provided by some DSLs support generalizing these complex scheduling challenges. Recent advances in machine learning have shown benefits in navigating large and complex resource scheduling tradeoff spaces [7, 22]. Combining DSLs with automated learning systems for scheduling and resource management now makes it practical to simultaneously achieve the trio of aforementioned goals: efficiency, productivity, and portability.

Finally, adaptive runtime approaches have been shown to be helpful in optimizing application resource usage in dynamic computing environments [4, 12, 17]. As with compilation, the higher abstraction levels provided by DSLs support using a common runtime framework for adaptively rebalancing workloads to improve performance, and even to optimize in other dimensions such as power/energy consumption.

We propose CASPER—a novel domain-specific compiler and runtime framework to enable domain scientists to achieve high performance and scalability for their HPC applications. We demonstrate the approach with two application domains – synthetic aperture radar (SAR) and computational fluid dynamics (CFD). SAR represents a class of real-time HPC applications while CFD represents physical simulations, each of which have very different data processing structures and requirements. To support these domains, we implement CASPER to support the Halide DSL for SAR [28] and the PyOP2 DSL for CFD [29]. This paper motivates the need for CASPER and presents an initial design and proof-point analysis using these DSLs and application domains. For example, our initial schedule optimization results demonstrate that CASPER can achieve up to $1.4\times$ speedup over the optimized code produced by the Halide DSL.

978-0-7381-1074-5/20/$31.00 ©2020 IEEE
DOI 10.1109/ESPM251964.2020.00010

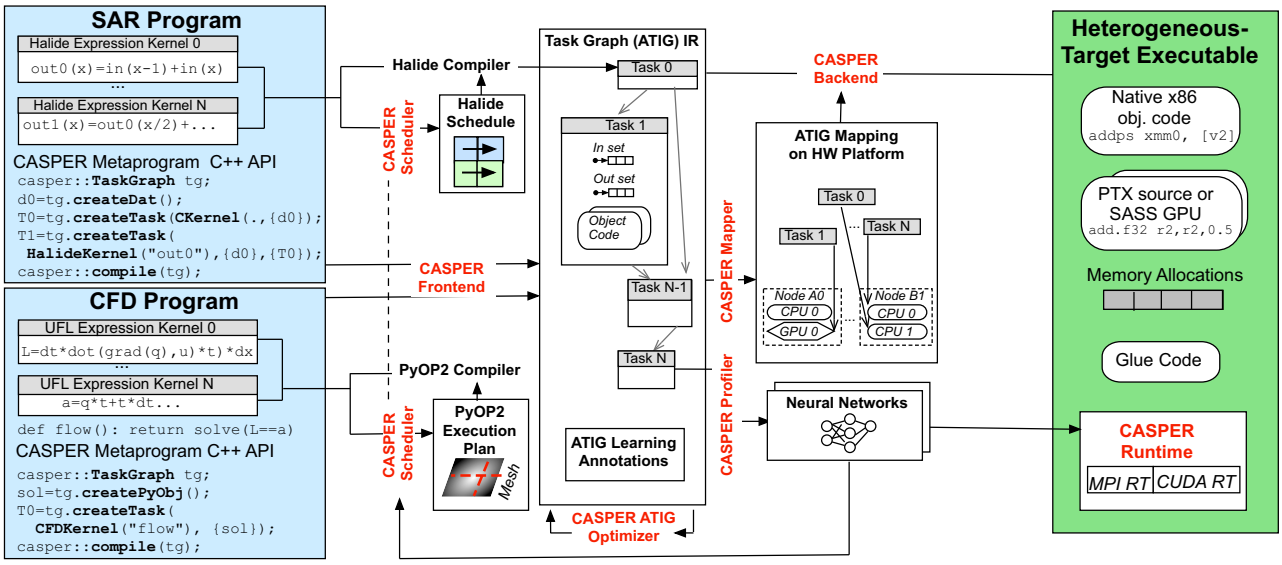


Fig. 1: CASPER system architecture.

II. Background

Currently, HPC application developers must be experts in both their scientific domain and HPC programming techniques. C/C++ and Fortran, combined with frameworks like OpenMP, MPI, and UPC++ [2], still dominate HPC codebases today. Implementing parallel HPC programs in these languages requires manually—and correctly—managing communication and synchronization. These requirements are difficult enough when programming for a single homogeneous HPC platform and still require significant experimentation with different approaches to find the best configurations. When new platforms—and heterogeneous resources like GPUs—need to be supported, this process must be repeated and application code modified or entirely rewritten to support new hardware architectures.

Domain-specific languages (DSLs) help overcome some challenges in programmer *productivity*. Some DSLs allow users to specify application kernels declaratively with implicit iteration, i.e., without explicit inner loop structures. Data partitioning and parallelization is then managed by externally-specified schedules or automatically by the DSL. However, determining efficient application resource schedules is a hard problem, and one that is not limited to a particular programming language. CASPER, introduced in the following section, offers an automated and accurate scheduling approach for SAR and CFD applications written in high-level DSLs.

III. Design

Figure 1 presents the top-level CASPER architecture with artifacts produced/consumed by compiler components shown in blocks and the components labeled on the edges.

A. CASPER Programs

A CASPER application is a meta-program written in a host language (C++) that constructs a target program from sources written in a DSL (Halide [28] or UFL [29]). A CASPER meta-program constructs an intermediate task graph representation of the application, which is then lowered into an executable binary in several steps, as outlined in Figure 1. The multi-step compilation process allows CASPER to optimize applications at both higher and lower abstraction levels. In contrast to existing DSL-based systems, in CASPER the program code is strictly decoupled from the meta-program code. The meta-program executes once ahead-of-time to compile the program, and the program executes on the target system without any just-in-time compilation. This separation allows the CASPER compiler to optimize over the complete program and keeps compilation off the critical path at runtime.

In DSL code, iteration is implicit—the DSL code defines the iteration domain but lacks the associated control flow in the form of loops. In Halide programs, iteration is implicit over the domain of variables in an expression; in UFL programs, iteration is implicit over the elements of a mesh. Before the application can be executed, it must be lowered into a representation with explicit iteration, i.e., loops. In fact, there may be more than one lowered *variant*, e.g., w.r.t. the structure of the generated nested loops or strides within loops. Since *diversity* and *heterogeneity* in HPC platforms impact which variants perform best, the CASPER compiler selects the variants using a machine-learning procedure.

The CASPER compiler front-end represents the program as an Annotated Task Interaction Graph (ATIG), described further in Section III-C [33]. CASPER provides a C++ meta-programming API for constructing task graphs. Each task in the graph is created from an implementation of a kernel function written in DSL code or in a general-purpose language (C, Python). As the meta-program executes (at compile time), each task is added to the task graph, including the metadata about the input/output arrays. For example, Listing 1 demonstrates a

CASPER meta-program to load, invert, blur, and save a bitmap image. The blur kernel is implemented as a Halide pipeline, shown in Listing 2. The meta-program refers to this kernel by its identifier (`halide_blur`) and invokes it on a pair of data buffers. The invert task as well as auxiliary I/O tasks are implemented in C (not shown).

```
1  TaskGraph tg;
2  const int W = 1695, H = 1356, F = 3;
3  Dat *img = &tg.createDat(H, W);
4  Task& t_load = tg.createTask(CKernel("bmp_load"), {img});
5  Task& t_inv = tg.createTask(CKernel("img_inv"), {img},
6                              {&t_load});
7  Dat *img_blurred = &tg.createDat(H - F/2, W - F/2);
8  Task& t_blur = tg.createTask(HalideKernel("halide_blur"),
9                               {img, img_blurred}, {&t_inv});
10 Task& t_save = tg.createTask(CKernel("bmp_save"),
11                              {img_blurred});
12 compile(tg);
```

Listing 1: CASPER example application.

```
1  Input<Buffer<double>> input{"input", 2};
2  Output<Buffer<double>> blur_y{"blur_y", 2};
3  Func blur_x("blur_x");
4  Var x("x"), y("y"), xi("xi"), yi("yi");
5  blur_x(x,y)=(input(x,y)+input(x+1,y)+input(x+2,y))/3;
6  blur_y(x,y)=(blur_x(x,y)+blur_x(x, y+1)+blur_x(x,y+2))/3;
7  // Schedule
8  blur_x.compute_at(blur_y, x_o)
9    .split(x, x_vo, x_vi, v1).vectorize(x_vi);
10 blur_y.compute_root()
11   .split(x, x_o, x_i, v2).split(y, y_o, y_i, v3)
12   .reorder(x_i, y_i, x_o, y_o)
13   .split(x_i, x_i_vo, x_i_vi, v4)
14   .vectorize(x_i_vi).parallel(y_o).parallel(x_o);
```

Listing 2: A CASPER task implemented in the Halide DSL.

The CASPER front-end invokes the respective DSL compiler on each of the tasks constructed in the meta-program. The object code produced by the DSL compiler is then referenced from the node in the task graph representation.

B. Variant Selection

Before a task can be lowered into executable code, the compiler must choose one of many possible loop structures for traversing the iteration domain of the task. For example, a task that doubles each element in a matrix could iterate by row then by column, or by column then by row, or in square tiles, etc. In Halide tasks, the loop structure is controlled by the schedule written as part of the meta-program, e.g., beginning at line 8 in Listing 2. A schedule may also be generated by an "auto-scheduler," a search procedure based on a genetic algorithm that is shipped with Halide. UFL does not expose scheduling directives—the form compiler chooses a schedule when it annotates the procedural code it generates.

CASPER accepts both manually written and auto-generated schedules for Halide and inherits the schedules generated by the form compiler. Schedules define different implementation variants through parameters, e.g., width of the `vectorize` directive, the tile size, and/or the parallel fan-out factor. The meta-programming API accepts partial schedules, where some parameters are not bound to a specific value, but are left to CASPER to choose. CASPER evaluates different schedule variants using a performance predictor trained on profiling measurements. The best-performing variant for each processing element type is linked into the target binary.

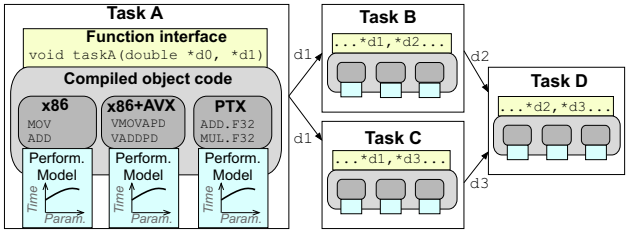


Fig. 2: ATIG representation for a CASPER application.

C. ATIG Mapping

The Annotated Task Interaction Graph (ATIG) program representation constructed by the CASPER front-end is illustrated in Figure 2. Graph nodes represent the static tasks in the application and edges represent the dependencies between the tasks given by the dataflow. Each task contains one or more *implementation variants*, generated by the variant selector. When multiple variants are available, CASPER's ATIG Mapper chooses the best ones to use depending on available resources, which may still result in multiple options to be decided on at runtime, e.g., when running on diverse or heterogeneous resources, or to select between options with different scaling behaviors. ATIGs contain all information necessary to execute the application, as well as metadata like performance models for estimating runtime on a candidate target compute resource.

Given an ATIG, the CASPER Mapper generates a close-to-optimal allocation of tasks on the available hardware resources. To formulate the optimization problem, we first observe that the computations in many scientific applications have a pipelined nature, i.e., the tasks are expected to occur in stages. Even when the tasks are not explicitly pipelined, it is possible to convert any ATIG to a graph with pipelined stages using its topological ordering because the graph is directed and acyclic.

Even though ATIGs are pipelined, the problem of finding an optimal mapping is still NP-Hard. It remains so, even when there is only one stage in the pipeline [10]. Greedy algorithms may perform extremely poorly—examples can be generated to show greedy is worse by any arbitrary factor. If we allow violation of capacity constraints by sequentially executing tasks on resources, then a greedy solution results in a 2-approximation. Further, this error can accumulate over multiple stages of an ATIG pipeline. Therefore, we avoid using greedy algorithms by default, and invoke one only when our other proposed mapping solutions are intractable. See Wang et al. for additional formulation details and ATIG evaluation [33].

D. Compiler Back-end

The CASPER back-end assembles a self-sufficient executable binary capable of invoking all tasks in the task graph. The representation of each task already contains the compiled object code that implements the task's computation. The object code is generated by the respective DSL compiler when the meta-program executed, but is still isolated from other tasks in the application. CASPER's back-end generates the glue code that sets up the dataflow between tasks.

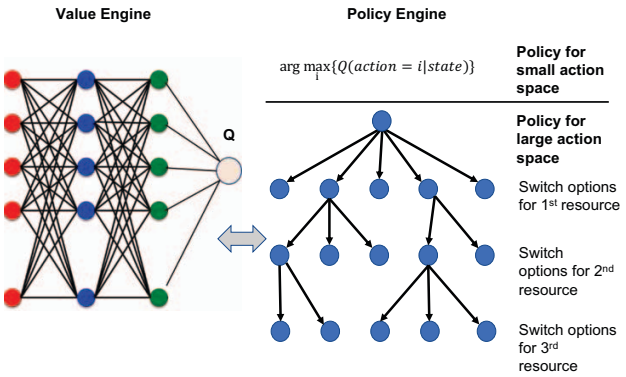


Fig. 3: Runtime Dataflow Remapper neural network design.

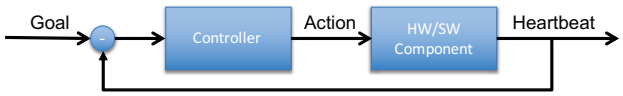


Fig. 4: Runtime Decision Engine closed-loop design.

Based on the metadata associated with each task, propagated from the top-level application meta-program, CASPER allocates data arrays in memory to hold input and output data that flows between tasks. The type of memory and location in the memory array is chosen according to the resource mapping produced by the CASPER Mapper. When two adjacent tasks are allocated to compute resources that cannot share the same memory array, CASPER generates procedures for copying the data between memory arrays. For example, a task allocated to a GPU that consumes data produced by a task allocated to a CPU requires a procedure for copying memory from host memory into the GPU device memory.

The glue code generated by CASPER's back-end may leave some degrees of freedom to be resolved at runtime. For example, tasks may provide multiple implementations, each of which may also be parameterized and may be derived from the same source code but compiled for different target hardware. The CASPER Runtime may then choose the most efficient implementation variant at execution time, e.g., the number of threads, or the CPU object code vs the GPU object code.

E. Runtime

The CASPER Runtime is responsible for assigning tasks to available resources and adaptively reconfiguring the application to improve its behavior. Some conditions can only be observed at runtime, such as workload imbalance [3, 15], hardware variation [5], inter-node communication latencies, and shared resource contention. Furthermore, these conditions may change during execution due to shifting application workloads and other *dynamic* behavior within the larger system. The CASPER Runtime instruments software and hardware sources as *heartbeats* to quantitatively describe the current state of the application and the resources assigned to it, e.g., measuring task wall-clock performance, data exchange throughput or latency, or system hardware performance counters.

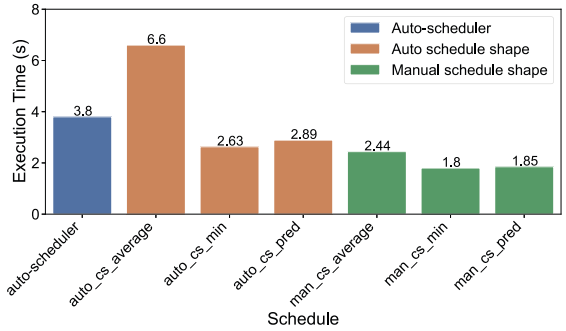


Fig. 5: Blur runtimes for different schedules.

While tasks are initially assigned to resources by the CASPER Mapper offline, Dataflow Remappers may re-assign tasks at runtime to different resource types, counts, and locations. As shown in Figure 3, Remappers are a combination of a value engine and a policy engine, which predict if a change to the application dataflow is desirable. The value engine evaluates the quality of the current state based on heartbeat patterns modeled offline. The policy engine generates a distribution over the action space from where an action is sampled that is expected to improve the performance in terms of the value estimate. A simple example of the action decision would be whether (1) to switch from GPU to the CPU on the same node or (0) do nothing. This can be modeled as deep Q learning approach [23] where the evaluation of the state is approximated by a neural network and the policy is determined by evaluating it for both options (0 and 1).

Decision Engines are closed-loop *controllers* that, based on measured heartbeats, optimize within a dataflow configuration. Controller implementations might include control-theoretic approaches, statistical models, machine learning estimators and classifiers, or heuristic algorithms. Decision Engines can implement existing approaches, e.g., for handling workload imbalance [3, 15] or performance/power heterogeneity and optimization [5, 12, 16], as well as future adaptive approaches.

IV. Evaluation

As a preliminary step in developing CASPER, we empirically study task scheduling and optimization, and performance and portability behaviors in CASPER SAR and CFD application domains using diverse and heterogeneous resources.

A. Schedule Optimization

To evaluate the schedule optimization, we focus on the parameters passed to the scheduling directives in the Halide Blur application from Listing 2. Figure 5 compares the execution time (averaged over 5 trials) when scheduling parameters are chosen by competing strategies. The schedule generated by the Halide auto-scheduler [26] completes in 3.8 s. We characterize the search space through brute force, by measuring the runtimes of 500 schedule variants with alternative parameter values. The average runtime of these variants is 6.6 s (auto_cs_average) and the minimum is

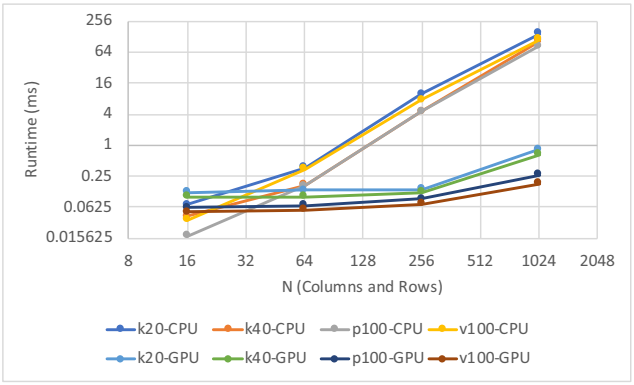


Fig. 6: FFTW vs cuFFT kernel runtimes on CPU and GPU.

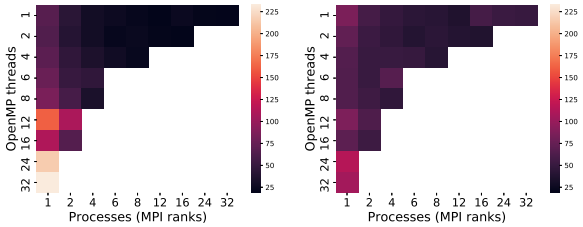
2.63 s (auto_cs_min). This brute-force search yields an average speedup of $1.41\times$ over the schedule generated by the Halide auto-scheduler. CASPER's non-brute-force optimizer achieves a speedup of $1.31\times$ by quickly finding a schedule that runs in 2.89 s (auto_cs_pred). In an analogous experiment we invoke CASPER's parameter search procedure on a schedule created manually by Halide experts [27]. A brute-force exploration of the parameter space for this schedule reveals an average runtime of 2.44 s (man_cs_avg) and a minimum of 1.8 s (man_cs_min). CASPER finds a schedule with a runtime of 1.85 s (man_cs_pred), i.e., an improvement of $2.1\times$ over the parameters chosen by experts.

These results demonstrate the benefit of a neural network predictive approach over a manual or automated sweep of the parameter space. CASPER works by training a compact neural network performance model [31] with less than 100 weights to *predict* the execution time of variants without running them. Due to its compact size, it can estimate the runtimes of all the candidates in under 1 s. Our compact neural networks do not perform rote learning, but instead model performance; none of the parameters in the candidate set were seen during training.

B. Synthetic Aperture Radar

We investigate the performance and portability of FFT kernels, since DFTs are a critical component of SAR image processing. We compare two implementations of single-precision complex-to-complex FFTs, which CASPER might select from in a heterogeneous environment: FFTW3, a state-of-the-art open source library for CPUs [11], and cuFFT, a CUDA-based library for GPUs [9].

Figure 6 compares FFTW, running on host CPUs, against cuFFT on four different Nvidia GPU platforms – K20, K40, P100, and V100. cuFFT is superior for all but the smallest data size evaluated (64x64). However, if we also account for "plan" creation and data transfer overheads between the host DRAM and GPU memory (not visualized), then FFTW outperforms cuFFT in all evaluated cases. The optimal choice depends on data size, the GPU type(s) available, and whether the data transfer overhead can be amortized by pipelining with other kernels in an application. Failing to account for these conditions can result in orders of magnitude differences in



(a) MUMPS (b) SuperLU_Dist

Fig. 7: Effect of parameter choices on runtime of CFD task.

performance, e.g., hundreds of milliseconds compared with tens or hundreds of seconds for 1024x1024 FFTs, thus motivating the need for automatic task scheduling. CASPER will account for these and select the best option without requiring developers to specify it a priori.

C. Computational Fluid Dynamics

The final experiment assesses the potential improvement in performance CASPER can gain from tuning the implementation parameters exposed by the "solve" task in CFD applications. We consider an application that uses one of two state-of-the-art direct solvers: MUMPS [1] or SuperLU_DIST [19]. The parallel implementation of each solver exposes two parameters: the number of partitions (MPI ranks) and the number of OpenMP threads in the thread pool. Figure 7 demonstrates the significant effect these two degrees of freedom have on running time. First, we observe that HyperThreads do not add effective computation capacity. However, even ignoring HyperThread configurations, we observe an approximately $3\times$ difference between using 1 process with 12 threads and 12 processes with 1 thread each. Furthermore, we observe that adding more than 4 processes has a negligible benefit. To reach these conclusions in a traditional implementation approach, the developer is forced to manually measure the runtime for different combinations of these parameters. In contrast, CASPER finds the efficient configuration automatically by profiling and selecting the best settings.

V. Related Work

A major challenge to scaling SAR processing is determining how to partition both the raw SAR data and the final output image to maximize performance and reduce communication. Early work on parallelizing SAR image construction focused on distributing work between CPUs [18]. Recent work has gravitated toward accelerating computations with GPUs because of their ability to efficiently perform linear algebra operations in parallel [20]. Other efforts address scaling out SAR processing to multiple shared memory systems, still with a strong interest in utilizing GPUs [14, 34]. CASPER applies Halide DSL [28] to the SAR domain.

CASPER targets CFD on unstructured meshes. Prior works have evaluated CFD processing at scale, e.g., at 1024 nodes with CPUs/GPUs [35], 256 nodes with SIMD and cache optimizations [25], and even at 14,000 nodes (224,000 cores) [21].

Ghysels et al. present an algorithm for overlapping the all-to-all global reductions with other communications and calculations [13]. CASPER incorporates the PyOP2 backend, which overlaps communication of halo data with computation and has demonstrated $3.4\times$ speedup over Fluidity's advection-diffusion solver [29]. OP2, a predecessor of PyOP2, demonstrated speedups on GPUs over OpenMP of $3.5\times$ and $2.5\times$ for single and double precision computations, respectively [24].

VI. Conclusion

In this paper, we introduced our initial design and proof-point demonstrations for CASPER – Compiler Abstractions Supporting high Performance on Extreme-scale Resources. CASPER is a novel domain-specific compiler and runtime framework to enable domain scientists to achieve high performance and scalability for their HPC applications. Specifically, we target synthetic aperture radar (SAR) and computational fluid dynamics (CFD) application domains. CASPER uses Annotated Task Interaction Graphs (ATIGs) to efficiently map application kernels to *diverse* and *heterogeneous* resources, and an adaptive runtime to support performance optimization in reponse to *dynamic* runtime conditions. These features enable developers to write domain-specific HPC applications with potential to *scale* to thousands of nodes in these complex and challenging environments.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0019. Computation for the work described in this paper was supported by the University of Southern California's Center for High-Performance Computing.

References

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIMAX*, vol. 23, no. 1, Jan. 2001.

[2] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *IPDPS*, 2019.

[3] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Multi-level load balancing with an integrated runtime approach," in *CCGRID*, 2018.

[4] S. Barati, F. A. Bartha, S. Biswas, R. Cartwright, A. Duracz, D. Fussell, H. Hoffmann, C. Imes, J. Miller, N. Mishra, Arvind, D. Nguyen, K. V. Palem, Y. Pei, K. Pingali, R. Sai, A. Wright, Y. Yang, and S. Zhang, "Proteus: Language and runtime support for self-adaptive software development," *IEEE Software*, vol. 36, no. 2, 2019.

[5] D. Chasapis, M. Casas, M. Moretó, M. Schulz, E. Ayguadé, J. Labarta, and M. Valero, "Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes," in *ICS*, 2016.

[6] DARPA. (Apr. 2019). "Performant automation of parallel program assembly (PAPPA)," [Online]. Available: https://beta.sam.gov/opp/a0f600ae8e74078e981d8ed46adaf907/view (visited on 07/08/2020).

[7] C. Delimitrou and C. Kozyrakis, "Qos-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, Dec. 2013.

[8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A domain specific language for building portable mesh-based pde solvers," in *SC*, 2011.

[9] c. C. T. Documentation. (2020). "cuFFT :: CUDA toolkit documentation v9.2.148," [Online]. Available: https://docs.nvidia.com/cuda/archive/9.2/cufft/index.html (visited on 05/14/2020).

[10] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum general assignment problems," in *SODA*, 2006.

[11] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[12] N. Gholkar, F. Mueller, and B. Rountree, "Uncore power scavenger: A runtime for uncore power conservation on hpc systems," in *SC*, 2019.

[13] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose, "Hiding global communication latency in the gmres algorithm on massively parallel machines," *SISC*, vol. 35, no. 1, 2013.

[14] M. Gocho, N. Oishi, and A. Ozaki, "Distributed parallel backprojection for real-time stripmap sar imaging on gpu clusters," in *CLUSTER*, 2017.

[15] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz, "Juggle: Proactive load balancing on multicore computers," in *HPDC*, 2011.

[16] C. Imes, H. Zhang, K. Zhao, and H. Hoffmann, "Copper: Soft real-time application performance using hardware power capping," in *ICAC*, 2019.

[17] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *SC*, 2015.

[18] Jinwoo Suh, M. Ung, and V. K. Prasanna, "Parallel implementation of synthetic aperture radar on high performance computing platforms," in *ICA3PP*, 1997.

[19] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, no. 3, Sep. 2005.

[20] B. Liu, K. Wang, X. Liu, and W. Yu, "An efficient sar processor based on gpu via cuda," in *CISP*, 2009.

[21] L. C. McInnes, B. Smith, H. Zhang, and R. T. Mills, "Hierarchical krylov and nested krylov methods for extreme-scale computing," *Parallel Comput.*, vol. 40, no. 1, Jan. 2014.

[22] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning control for predictable latency and low energy," *SIGPLAN Not.*, vol. 53, no. 2, Mar. 2018.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013.

[24] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *InPar*, 2012.

[25] D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, and D. Keyes, "Exploring shared-memory optimizations for an unstructured mesh cfd application on modern parallel systems," in *IPDPS*, 2015.

[26] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, 2016.

[27] J. Ragan-Kelley. (2020). "Halide: A language for fast, portable computation on images and tensors," [Online]. Available: https://halide-lang.org/ (visited on 08/29/2020).

[28] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013.

[29] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly, "Pyop2: A high-level framework for performance-portable simulations on unstructured meshes," in *SC Companion*, 2012.

[30] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *VECPAR 2010*, 2011.

[31] A. Srivastava, N. Zhang, R. Kannan, and V. K. Prasanna, *Towards high performance, portability, and productivity: Lightweight augmented neural networks for performance prediction*, 2020.

[32] TOP500.org. (Jun. 2020). "Japan captures TOP500 crown with arm-powered supercomputer," [Online]. Available: https://www.top500.org/news/japan-captures-top500-crown-arm-powered-supercomputer/ (visited on 07/08/2020).

[33] T.-Y. Wang, A. Srivastava, and V. Prasanna, "A framework for task mapping onto heterogeneous platforms," in *HPEC*, 2020.

[34] A. Wijayasiri, T. Banerjee, S. Ranka, S. Sahni, and M. Schmalz, "Dynamic data driven image reconstruction using multiple gpus," in *ISSPIT*, 2016.

[35] C. Xu, L. Zhang, X. Deng, J. Fang, G. Wang, W. Cao, Y. Che, Y. Wang, and W. Liu, "Balancing cpu-gpu collaborative high-order cfd simulations on the tianhe-1a supercomputer," in *IPDPS*, 2014.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: COMPILER ABSTRACTIONS AND RUNTIME FOR EXTREME-SCALE SAR AND CFD WORKLOADS

A. Abstract

This artifact contains details for reproducing results in the paper "Compiler Abstractions and Runtime for Extreme-scale SAR and CFD Workloads."

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Image blur, FFT, and FEM
- **Program:** C, C++, Halide, Python, OpenMP, MPI, and CUDA codebases
- **Compilation:** GCC, LLVM, OpenMPI
- **Hardware:** Intel CPUs and NVIDIA GPUs
- **Publicly available?:** yes

2) How software can be obtained (if available):

Blur Scheduling—performance prediction models code is available at https://github.com/Naifeng/Variant-Selection.

SAR FFT—FFT code using FFTW3 and CUDA cuFFT is available at https://github.com/cimes-isi/cuda-samples/ in the `v9.2-casper-fft` branch.

CFD Benchmark—Lid-Driven Cavity benchmark is available at https://github.com/acolinisi/casper-utils/blob/2020-espm2/exp/apps/fenics/cavity/demo_cavity.py

3) Hardware dependencies:

Blur Scheduling—Intel Core i5-7360U @ 2.30 GHz, 2 physical cores, 8 GB RAM.

SAR FFT—Intel Xeon E5-2665 @ 2.40 GHz, 16 physical cores, 64 GB RAM, with NVIDIA Tesla K20 GPU; Intel Xeon E5-2640 v3 0 @ 2.60 GHz, 16 physical cores, 64 GB RAM, with NVIDIA Tesla K40 GPU; Intel Xeon E5-2640 v4 0 @ 2.40 GHz, 20 physical cores, 128 GB RAM, with NVIDIA Tesla P100 GPU; Intel Xeon Gold 6130 @ 2.10 GHz, 32 physical cores, 192 GB RAM, with NVIDIA Tesla V100 GPU.

CFD Benchmark—Intel Xeon E5-2670 @ 2.60 GHz, 16 physical cores, 48 GB RAM.

4) Software dependencies:

The versions listed capture the versions that were used and do not imply an exact version requirement.

Blur Scheduling—macOS 10.12.6, Apple LLVM version 9.0.0 (clang-900.0.39.2), Halide 8.0.0, performance prediction git repository fork (see above).

SAR FFT—CentOS Linux release 7.8.2003, GCC 8.3.0, FFTW 3.3.8 single-precision with OpenMP support, CUDA Toolkit v9.2, cuda-samples git repository fork (see above).

CFD Benchmark—System: CentOS Linux release 7.8.2003, GCC 10.1.0, CMake 3.17.3, OpenMPI 4.0.3, Python 3.8.3, pybind11 2.5.0; **Numerical libraries**: Blis 0.7.0, LAPACK 3.8.0, ParMETIS 4.0.3, Armadillo 9.860.1, VTK 8.2.0, MUMPS 5.1.2, SuperLU_Dist 6.3.1, PETSc 3.13.1 (configured with MUMPS and SuplerLU_DIST support), Eigen 3.2.90, Boost 1.73.0, libxml2 2.9.9, DOLFIN 2019.1.0.post0; **Python packages**: pkgconfig 1.5.1, numpy 1.18.5, dolfin 2019.1.0.post0, FFC 2019.1.0, FIAT 2019.1.0, Dijisto 2019.1.0, UFL 2019.1.0, matplotlib 3.2.1.

C. Installation

Except as specified here, follow build and installation instructions included in software README files and add paths as needed to common environment variables, e.g., `PATH`, `LD_LIBRARY_PATH` and `PKG_CONFIG_PATH`.

Blur Scheduling—Follow the standard installation instructions for Halide 8.0.0 and for each Python package.

SAR FFT—For FFTW3, build the single-precision floating point version and threaded libraries by passing `--enable-float --enable-threads --enable-openmp` to the `configure` command, then `make` and `make install` normally. In the `cuda-samples/Samples/simpleCUFFT/` directory, set the `CUDA_PATH` environment variable (or modify the Makefile) to point to your CUDA Toolkit installation prefix before running `make`.

CFD Benchmark—The complete software stack may be installed into a container-like directory using the Portage package manager as described in https://github.com/acolinisi/casper-utils/tree/2020-espm2 tag `2020-espm2`. Alternatively, follow the standard autotools installation instructions for each numerical library and standard setuptools installation instructions for each Python package.

D. Experiment workflow

Blur Scheduling—To reproduce Figure 5, follow the instructions in `Halide/Blur/` directory to compile and run code in `Halide/Blur/blur_cpu`. Use `pred_NN+C.py` as the predictor.

SAR FFT—To reproduce Figure 6, in the `cuda-samples/Samples/simpleCUFFT/` directory, run the `simpleFFT2D-omp` and `simpleCUFFT2D` binaries with a single parameter – the N value for the number of rows and columns to use (i.e., 16, 64, 256, and 1024). In a SLURM environment (even with an interactive job), you may need to launch `simpleCUFFT2D` with `srun`, e.g., `srun -n1 ./simpleCUFFT2D 1024`.

CFD Benchmark—To reproduce Figure 7, in `casper-utils/exp/heatmap/` directory, run `bash heatmap.job.sh > heatmap.log 2>&1` to generate the raw log with the timing measurements for each datapoint. Then, parse the datapoints from the log into CSV format with `cat heatmap.log | python parse_heatmap.py > heatmap.csv`. Finally, plot the data with `python plot_heatmap2.py heatmap.csv heatmap.pdf`.

E. Evaluation and expected result

Blur Scheduling—The runtime of the Manual schedule should be less than the runtime of Auto schedule. Performance prediction results (e.g., training/testing accuracy) will be outputted by the predictor to the console.

SAR FFT—Both binaries will output `Kernel Time = <val> msec` and `Total Time = <val> msec`. Our evaluation uses the kernel time, which should be less than the total time. Larger N values should take longer to execute, especially for the kernel times, but total times may be close with cuFFT where overheads dominate the execution time.

CFD Benchmark—The expected heatmap generated by the instructions above should be non-uniform, where neither maximum ranks nor maximum OpenMP threads yield the shortest execution time.

F. Experiment customization

Blur Scheduling—None.

SAR FFT—In `simpleCUFFT.cu`, you may optionally redefine `DO_CHECK_RESULT` to 0, as verification is slow during execution.

CFD Benchmark—Variables in the `heatmap.job.sh` script may be customized to run the experiment for a different input size (`MESH`) or with for a different range of ranks (`PROCS`) and OpenMP thread counts (`THREADS`).

G. Notes

Computation for some of the experiments described in this paper was supported by the University of Southern California's Center for High-Performance Computing (hpcc.usc.edu). Specifically, SAR and CFD results were collected on the old HPCC cluster, but most of its resources have now been decommissioned and replaced by the new Discovery cluster.