An Extensible Framework for Composing Stencils with Common Scientific Computing Patterns

Leonard Truong, Chick Markley, and Armando Fox

Computer Science Division, University of California, Berkeley
565 Soda Hall, MC-1776
Berkeley, CA 94720
{lennytruong,chick,fox}@berkeley.edu

Abstract
The SEJITS framework supports creating embedded domain-specific languages (DSELS) and code generators, a pair of which is called a specializer, with much less effort than creating a full DSL compiler—typically just a few hundred lines of code. SEJITS’ main benefit is allowing application writers to stay entirely in high-level languages such as Python by using specialized Python functions (that is, functions written in one of the Python-embedded DSELS) to generate code that runs at native speed. One existing SEJITS DSEL is Sepya [10], a Python DSEL for stencil computations that generates OpenMP and Cilk+ code competitive with existing DSL compilers such as Pochoir and Halide. We extend Sepya to generate OpenCL code for targeting GPUs, and in the process, extend SEJITS with support for meta-specializers, whose job is to enable and optimize the composition of existing specializers written by third parties. In this work, we demonstrate meta-specialization by detecting and removing extraneous data copies to and from the GPU to compose multiple specializer calls (stencil and non-stencil). We also explore the variants of loop fusion to further improve performance of composing these operations. The performance of the generated stencil code is 20× faster SciPy and competitive with existing stencil DSELS on realistic code excerpts. Since meta-specializers must compose and optimize specializers created by third parties, we extend SEJITS with support for meta-specializer hooks, allowing existing specializers to be incrementally enabled for meta-specialization without breaking backwards compatibility. The Sepya and SEJITS extensions together extend the range of platforms for which highly optimized code can be generated and open new possibilities for optimizing the composition of existing specializers.

Categories and Subject Descriptors D.2.11 [Software Architectures]: Domain-specific architectures; D.3.2 [Language Classifications]: Specialized application languages, Extensible languages; D.3.3 [Language Constructs and Features]: Frameworks

Keywords Software Engineering; Software Architecture; Domain-specific Architectures; Patterns

1. Overview & Contributions
Scientific application authors face a trade off between productivity—being able to code in a natural language whose abstractions match their own problem domains—and high performance. In some simple cases, high-performance libraries bridge this gap (BLAS, OSKI), but for more sophisticated computations such as stencils, different problem instances share common structure and abstractions, but the computations are too different to encapsulate using libraries alone. This observation has led to other approaches such as the use of domain-specific languages (DSLs) for stencils. By focusing on expressing what computation is to be done rather than how to do it, DSL compilers can often apply optimizations specific to both the computation and the target hardware whose applicability would be difficult to infer from imperative code expressing lower-level operations. In particular, embedded DSLs (DSELS) have the additional advantage [7] that programs can take advantage of most or all features of the embedding language, and that complete programs in the embedding language can include “subprograms” in multiple distinct embedded languages.

The SEJITS framework (SEJITS.org) provides the infrastructure for creating DSELS embedded in Python. The corresponding DSEL “micro-compilers”, called specializers in the terminology of that system, are written in Python and typically span a few hundred to a couple of thousand lines of code. By restricting a specializer to targeting a specific computational pattern or kernel [14], the specializer writer can leverage expert knowledge of domain-specific optimizations to generate high-quality, efficiency-level code (C/OpenMP, Cilk+, Cuda). The application writer’s experience is one of writing entirely in Python, with the ability to call specialized functions that exhibit the same behavior as a native Python function. Sepya, a Python DSEL for stencil computations, is one of the many varieties of extant specializers. It produces high-quality OpenMP or Cilk+ code whose performance is competitive with hand-crafted and autotuned code [9].

This paper extends previous work on SEJITS in three ways.

1. We add support for generating high-quality GPU code to compute stencils without changing the user facing DSEL. That is, existing Python programs that encode SEJITS stencil operations run unmodified and with high performance on GPU-equipped systems, providing both source and performance portability for such programs.

2. We extend the SEJITS framework with a generic mechanism to avoid copying of intermediate results to/from the GPU between multiple specialized function calls.

3. We extend the SEJITS framework with a mechanism that enables the construction of meta-specializers that can implement...
optimizations across existing specializers. In particular, we use this mechanism to implement loop fusion [11] across specializers, inspired by the approach of Anderson et al [1]. Multiple consecutive specialized operations can be fused including stencils and common linear algebra operations such as matrix addition.

An alternative strategy for optimizing across kernels is to define a top-level DSL or framework that does whole-program or whole-module analysis. Indeed, both earlier SEJITS frameworks [5] and other DSLs or DSELs for various scientific and image processing kernels [1, 15, 17, 18] work this way. However, in our ecosystem, individual specializers written by third-party developers are "plugged into" the SEJITS framework; meta-specializers allow existing third-party-authored specializers to participate in across-kernel optimizations, and the ability for a specializer to work with a given meta-specializer can be added incrementally, so that as more specialists add this capability, more cross-specializer optimization scenarios become possible.

The rest of this paper is organized as follows. Section 2 reviews the existing SEJITS framework and Sepya, the OpenMP/Cilk+ stencil specializer. Section 3 describes the new GPU code generator for Sepya, which consumes the same input as the OpenMP/Cilk+ code generator but produces data-parallel implementation of Stencils that leverages knowledge of modern GPU architecture to maximize efficiency. Unsurprisingly, the GPU code generator achieves up to $10 \times$ speedup over computing stencils using native functions in the SciPy library. Section 4 describes the analyses required to eliminate redundant data transfer between the GPU and host processor across specializer calls, which increases performance up to $20 \times$ on various problems. It also describes how the same machinery enables loop fusion across specializers, which when possible results in additional speedup of up to $5 \times$, depending on how many computations are fused. Section 5 reports our initial performance results, after which we discuss ongoing and related work and conclude.

2. Background: SEJITS

A SEJITS "specializer" is a micro-compiler that adapts a specific computational pattern to a specific hardware target. While the SEJITS concept is language-independent and works with any modern dynamic language that supports graceful embedding of DSELs [4], our prototype framework uses Python as both the source language and the language in which specializers (micro-compilers) are written. A typical specializer is a few hundreds to a couple thousand lines of Python [9].

In terms of implementation, a specializer is a Python class that subclasses from an abstract class provided by the framework. Each specializer defines a domain-specific language for the pattern it compiles and encapsulates expert knowledge of how to compile the abstractions in that language into low-level code. The SEJITS framework provides tools to the specializer writer including a c-family intermediate representation (IR), code generation, compilation, and linking machinery, integration with OpenTuner for tuning parameter exploration and caching.

For example, Sepya defines abstractions for describing stencil computations as the pattern DSL for the particular pattern. In this example, the DAST corresponds roughly to the semantic model associated with the pattern DSL [6].

In the second phase, a code generator specific to this pattern (stencils) maps the DSL-specific constructs to efficient data structures and traversal strategies for the desired target hardware. The existing Sepya implementation targets multicore CPUs with OpenMP or Cilk+ capable compilers [9]; in this work we add support for targeting GPUs via an OpenCL backend. A narrowly pattern-specific code generator working from a DAST can employ "best practice" optimization strategies specific to the <pattern, hardware> pair. For example, the existing Cilk+/OpenMP-targeted specializer exploits extensive prior work in optimizing and tuning stencils via cache and register blocking, tiling, loop unrolling, and so forth [8], producing code whose performance exceeds 90% of the theoretical peak based on the roofline model [19] for predicting when computations become memory-bound.

Because the DAST expresses domain-level concepts such as "neighborhood" rather than relying on the programmer to define and traverse data structures explicitly, the code generators avoid much of the difficult task of analyzing generic code faced by optimizing compilers such as unintended loop carried dependencies or improper loop nesting. The second phase transformation emits source code in a lower-level language that is passed to a downstream optimizing compiler (initially, an OpenMP or CilkPlus compiler; in this work, the Clang OpenCL compiler). The resulting ob-

---

**Figure 1.** A simple stencil example using the SEJITS stencil DSEL uses the the DSEL rather than expressing traversal of the grids explicitly. By setting the neighborhood definition, the programmer can now leverage the DSEL abstraction neighborhoods rather than expressing the traversal of neighborhoods explicitly. The DSEL also provides interior_points allowing the abstract traversal of grids.

```python
from stencil_kernel import *
class Jacobi(StencilKernel):
    neighbor_definition = [
        [0.0, -1.0], (0, 1)],
        [-1.0, 0.0, 1.0]
    ]

def kernel(self, in_grid, out_grid):
    for x in self.interior_points(out_grid):
        for y in self.neighbors(x, 0):
            out_grid[x] += .1 * in_grid[y]
        for y in self.neighbors(x, 1):
            out_grid[x] += .3 * in_grid[y]
    jacobi_stencil = Jacobi(backend='ocl')
    out_img = jacobi_stencil(in_img)
```

Implementation of the regular grid example above.
```c
__kernel void kernel(__global const float* in_grid, __global float* out_grid, __local float* block) {
    int global_index = get_global_id(1) + get_global_id(0) * 1024;
    int thread_id = get_local_id(0) * get_local_size(1) + get_local_id(1);
    int block_size = (get_local_size(0) + 2) * (get_local_size(1) + 2);
    int num_threads = get_local_size(0) * get_local_size(1);
    for (int tid = thread_id; tid < block_size; tid += num_threads) {
        int local_id0 = tid / (get_local_size(1) + 2);
        int local_id1 = tid % (get_local_size(1) + 2);
        out_grid[global_index] += 0.1 * block[local_array_macro(local_id0 + 0, local_id1 + 1)];
        out_grid[global_index] += 0.3 * block[local_array_macro(local_id0 + 1, local_id1 + 0)];
    }
}
```

Figure 2. OpenCL code generated by the SEJITS-based stencil specializer Sepya for the code in Figure 1.

3. Extension to OpenCL and GPUs

Our goal was to add support for high-performance GPU implementations of stencils without modifying the stencil DSEL. That is, existing Python+SEJITS programs that used Sepya should run unchanged and with good performance on GPU-equipped systems.

Achieving this goal entails two tasks. The first is to create a code generator that consumes existing stencil DASTs but generates OpenCL code that maps the stencil’s parallelism to the data-parallel, OpenCL programming model. Furthermore, we leverage GPU programming techniques including coalesced memory accesses and shared local memory to improve performance on the GPU. The rest of this section discusses the implementation of this code generator on a high level, which was straightforward given the SEJITS framework’s support for AST manipulation, code generation, and so on.

The second and more difficult task is avoiding unnecessary copies between the host and GPU; it turns out the machinery necessary to do this also exposes an optimization opportunity when consecutive “loop-like” kernels with compatible loop geometries can be fused. Section 4 describes both of these optimizations.

As described in Section 2, the SEJITS framework supports extensibility by separating the DAST generation from backend code generation. New backends (hardware targets) can be added that consume the same DAST, without reimplementing the logic required for handling the initial Python tree.

4. Composing Specializers

A typical application using SEJITS involves the use of many different specializers, multiple instances of the same specializer, or both. Specializers targeting the GPU trade time spent moving data to and from the device for increased parallelism. Our meta-specializer can recognize opportunities for keeping data on the device to be used...
by another kernel, amortizing the cost of data movement across 
specializers.

For example, consider the program excerpt in Figure 4 where 
\texttt{jacobi\_stencil\_solver} is a plain Python function that contains 
calls (line 4) to three specializers: \texttt{jacobi\_stencil}, element-wise 
multiplication (by \texttt{D}), and element-wise subtraction (from \texttt{b}). (We 
overload Python’s multiplication and subtraction operator as syn-
tactic sugar to calls to specializers that perform matrix-vector and 
matrix-matrix operations.)

All three specializers can generate code for a GPU, but a 
naive code generation strategy would copy the intermediate re-
sults back from the GPU after each operation and copy the in-
put back to the GPU to be consumed. Indeed, since analysis and 
JIT compilation in the SEJITS framework happens at runtime, 
and since \texttt{jacobi\_stencil\_solver} is a plain Python function 
whose execution will \textit{not} be intercepted by SEJITS, a straight-
forward implementation is doomed to perform these needless 
data copies as there is no opportunity to pre-inspect the body of 
\texttt{jacobi\_stencil\_solver}. Our goal is for the intermediate results 
of GPU kernels to stay on the GPU until either the final result has 
been computed or some intermediate result needs to be consumed 
by code that cannot be run on the GPU.

We solve this using Python decorators, an aspect-oriented pro-
gramming mechanism that allows for an interceptor function to 
be called before the decorated function is executed. Our decor-
ator \texttt{@meta} triggers a meta-specializer that performs the following 
steps on the decorated function’s AST:

1. Determine which adjacent operations in the function body cor-
respond to specialized functions that are candidates for cross 
specializer optimization.
2. Determine whether an operation produces a result that is con-
sumed by a subsequent operation on the same device and if so, 
remove the machinery to copy the data to and from the host via 
AST manipulation.
3. Determine whether adjacent operations have compatible loop 
geometries that enable loop fusion, and if so, perform loop 
fusion by AST manipulation.

The following subsections explain each step in more detail. The 
key underlying “hook” is the implementation of a new interface 
called \texttt{fusable} that specializers that wish to be candidates for 
fusion must implement. The meta-specializer uses the \texttt{fusable} 
interface to gain access the ASTs and data dependency information 
about the specializers.

4.1 Eliminating Extraneous Copies
The meta-specializer first generates a control flow graph of the 
decorated Python function, introducing temporary variables where 
necessary and de-sugaring overloaded operators such as \texttt{*} into the 
actual functions they call. The technique is inspired by PyPy’s 

Flow Object Space [16]. Figure 5 shows the result of applying this 
transformation to the example program of Figure 4.

A group of adjacent specializers that are candidates for com-
bining is called a ComposableBlock; the current example trivially 
resolves into a single ComposableBlock. The meta-specializer then 
identifies adjacent pairs of specializers \texttt{A} and \texttt{B} within the 
ComposableBlock between which extraneous data copies can be re-
moved, which is possible when both \texttt{A} and \texttt{B} use the same device 
memory (GPU or CPU) and when \texttt{A’s} output is consumed by \texttt{B}. (Any symbols in the ComposableBlock that are referenced beyond 
the end of the ComposableBlock must of course be copied back to 
the host at the end of the execution of the block.) Figure 6 (center) 
shows an example of combining two specializer’s AST subtrees 
when these conditions hold: the intermediate copy-to-host and 
copy-to-GPU nodes have been removed.

To perform the required dataflow analysis, the meta-specializer 
uses a technique similar to symbolic execution [3]. With access to 
the current runtime scope including any defined symbols as well as 
arguments to the function being analyzed, the meta-specializer can 
generate the placeholder objects to be passed to specializers. Spe-
cializers are written to produce trees dependent on the values they 
are consuming, thus the meta-specializer must provide placeholder 
objects to trigger the generation of ASTs. For example, the Sepya 
AST generator uses information about the dimensions of the input 
grid to generate the AST representation of the proper loop geometry 
for iterating over the input; the placeholder value passed by the 
meta-specializer provides this information as readily as the “real” 
arguments would.

The analysis allows the meta specializer to guarantee that the 
transformations to the program execution are legal. It ensures that 
a live variable is available on the platform at the time it is needed. 
Transformations to remove data movement are only performed 
when dataflow analysis reveals that a variable is no longer live 
outside of its enclosing block.

4.2 Loop Fusion
If a pair of adjacent specializers can be combined by removing 
extraneous data copies, the meta-specializer next inspects the two 
ASTs for compatible loop geometries that can be fused. The meta-
specializer supports traditional loop fusion [11] which promotes 

Figure 4. The \texttt{jacobi\_stencil} in line 4 calls a specialized func-
tion, but unless it can be combined with the element-wise multipli-
cation and subtraction on the same line, data copying to and from 
the GPU will occur during every iteration of the loop that starts in 
line 3, and the overhead of this copying will negate much of the 
benefit of the GPU parallelism in computing the actual stencil.

Figure 5. First step in meta-specialization: decompose the deco-
rated function of Figure 4 into a control flow graph by introducing 
temporary variables and de-sugaring overloaded operators.

Figure 7. This is an example of how the specializer writer ex-
presses loop dependencies for a laplacian stencil. The first 
argument specifies a paramater, and the second describes a vector magni-
tude and direction expressing the loop iterations that the body of 
the loop nest is depedent on.
intermediate values to registers to leverage data locality to increase performance, as well as a block-level loop fusion scheme which divides the work of computing intermediate values among the work-items in a work-group and stores there results in local memory.

Fusing loops requires understanding whether loop-carried dependencies are present. Since specializers are generally written by third parties, attempting to analyze their ASTs in full generality is “compiler-hard”, so we instead require the specializer writer to provide this information via the fusible interface (Figure 7).

Since a well-designed DSEL prevents programmers from expressing hard-to-analyze constructs—for example, the Sepya DSEL provides abstractions for neighbors and grid iteration, preventing the programmer from writing hard-to-analyze grid traversals directly—the specializer writer need only report the loop-carried dependencies resulting from the generated code, without worrying about any such dependencies in the application writer’s code. Given that the specializer writer is typically a sophisticated programmer, we believe this is a reasonable trade-off, since it greatly simplifies the meta-specializer’s dependency analysis task.

For loops with no loop-carried dependencies, such as element-wise matrix operations, fusion is easy: we reuse values in subsequent operations, and promote values to registers when the buffers being passed between operations are killed after exiting the the body of the fused loop. In Figure 4, we see that the result of computing an element in the output of jacobi_stencil can be immediately used in the subtraction and multiplication operations, preventing the need to store the results in an intermediate buffer. By removing the need to allocate buffers and reducing the number of cache penalties, fusion can greatly improve the performance of programs involving symmetric loops. However there are situations where loop fusion can incur performance penalties, for example when introducing too many live variables causes register pressure, or when reading and writing too many different buffers causes thrashing in the cache. Our implementation performs maximal fusion (fusing where possible), where a smarter implementation would use a model and/or autotuning to explore the space of possible fusion combinations.

Stencil loops pose a more complex problem because they contain loop-carried, read-after-write (RAW) data dependencies. One way to fuse the loops would be to recompute the needed values from the previous operation for each loop body. This incurs a lot of compute penalty, so instead, we share the work of computing a block of the intermediate values among work items, performing a block-level loop fusion. The results of intermediate operations are stored into local memory as mentioned above, and memory barriers to ensure all work items’ intermediate values are ready before continuing to the next result. Figure 5 shows how the intermediate results of \( A \times \alpha \text{pha} \) are ready before continuing to the next result. Figure 8 shows how the intermediate results of \( A \times \alpha \text{pha} \) are stored in local memory. The space required for the intermediate results, which depends on the size of the workgroups and the stencil’s ghost zones, therefore limits the number of operations that can be fused. The tradeoff for this method of fusion is having to recompute values in the ghost zone for each work group. Block level fusion results in an increased computational intensity by reducing the amount of DRAM traffic incurred by two subsequent kernels. Because most stencils are bandwidth bound [10], the cost of recomputing values in the ghost zone is amortized by the increased amount of possible operations per second [19]. We currently do not support fusion of loops containing antidependencies (WAW) or output dependencies (WAR).

Similarly to the data movement analysis in Section 4.1, the meta-specializer leverages the information provided by the special-
We were first interested in examining how our optimizations faired against various naive implementations such as SciPy library calls, as well as a specialized implementation with no cross pattern optimizations. We then investigated further to purely assess the effect of our block-level loop fusion with stencil computations.

SciPy is a reasonable naive baseline because of its wide use in the Python scientific computing. Figure 9 shows that two successive applications of a laplacian filter with Sepya+GPU is up to 20× faster than using scipy.ndimage.filter.convolve, strengthening the argument that an application writer can get both productivity and high performance using our approach. More interestingly, the figure reveals the gains for removing extraneous data copies (green line) and the additional gains obtained from loop fusion (red line).

Figure 10 concentrates on isolating the performance increases from fusing two, three, and four consecutive stencil computations. The speedup is relative to the performance of calling the stencils sequentially with no loop fusion. Most stencils are bandwidth bound, and therefore can benefit from transformations like loop fusion that can reduce the amount of DRAM traffic and increase arithmetic intensity. Block-level loop fusion works like an explicit on-chip caching scheme that prevents unnecessary reads and writes to the global address space. The reduced communication and increased amount of computation corresponds to an increase in arithmetic intensity (AI). Because these are bandwidth bound kernels, the increased AI translates to higher achievable GFLOPs[19].

6. Related Work
The related work in both stencil DSL’s and image processing pipelines is substantial, but the two most closely related projects that influenced our extensions to SEJITS are Hindemith and Halide.

Hindemith framework [1] showed that loop fusion in image processing applications (which are stencil-rich) can lead to large (up to 60x) performance gains over reference implementations, and was one inspiration for this work. Hindemith used a subset of Python as the language for program description, and within this subset, performed loop analysis and other tasks to effect fusion. Unlike SEJITS, Hindemith requires the entire Python program to be written in its subset of Python, whereas SEJITS embeds DSELs into Python, allowing a single Python program to use multiple different computational patterns and regular Python code that will not be JITed. As well, Hindemith is a full-stack solution with its own code generators, JIT managers, and so on; reimplementing its core loop-fusion analysis in SEJITS results in 60% fewer lines of code than the original Hindemith stack.

Halide [17] is a C++ embedded DSL that provides re-orderable scheduling primitives targeting stencil-rich image processing pipelines. While Sepya can also handle such applications, it can be composed and used in non-image-processing applications as well. Halide is optimized for operating on pinned 8- and 16-bit integers, allowing it to take greater advantage of multicore CPU features such as wide SIMD, vector extensions, and so on, whereas Sepya generates code for double-precision floating-point arithmetic by default. Also, as mentioned in the Introduction, Halide is a DSEL targeting a particular application domain and performing optimizations over operators in that domain, whereas SEJITS is a framework to which a variety of DSELs can be and have been added by developers other than the framework authors. This architectural difference motivates our use of meta-specializers to perform cross-specializer optimizations, potentially allowing DSELs contributed by others to participate in meta-specialization.

7. Ongoing and Future Work

7.1 Maximal Fusion
Fusion can also occur at the register level by having each work item compute any values needed by its computation. The distribution of work for computing intermediate values requires the use work-item synchronization, as well as reads and writes to local memory. By computing values needed on demand, pressure is lifted off of local memory, but register pressure as well as compute bandwidth is increased. Halide demonstrated that in some spaces the optimal fusion implementation can be counter intuitive. By implementing this form of fusion, we would be able to leverage auto tuners to explore the space possible optimization methods.

7.2 Making other specializers Fusable.
As Section 4.1 described, the meta-specializer relies on the fusable interface to be implemented by specializers called from a decorated function, to get each specializer’s AST corresponding to possibly fusable loops and to perform symbolic execution of the specializer to analyze its dataflow. In keeping with the Open-Closed Principle (OCP) [13], which states that software modules should be open for extension but closed against modification, adding such
an interface to existing specializers written in Python does not require recompilation, and Python’s introspection allows SEJITS to immediately detect whether a given specializer does or does not implement the interface. The **fusible** capability can therefore be deployed incrementally to existing specializers. We are examining other cross-specializer optimizations from the perspective of structural design patterns [12] to identify similar opportunities, such as fusing stencils with reductions and upsampling/downsampling. Our underlying philosophy remains that individual specializer writers can augment their specializers with this new functionality without breaking backwards compatibility.

7.3 Autotuning

The SEJITS framework is tightly integrated with OpenTuner [2], which allows the programmer to define a tuning harness for each specializer. While several individual specializers are integrated with OpenTuner, extending autotuning to meta-specialization involves much more complex tuning spaces. Stencil meta-specialization in particular will require exploration to determine the optimal amount of fusion, which may be less than the maximum imposed by the size of local memory. Lessons learned from Halide’s scheduling system are helping to guide our efforts here.

7.4 Exploring more hardware targets

We examined meta-specialization in the context of GPU backends. We want to extend these techniques to other hardware targets where the tradeoffs could be similar or different. For example, in the context of a multicore CPU, data movement analysis would be unnecessary because all the work would be done inside the same memory region. However, we think the loop fusion techniques could still provide similar speedup.

8. Conclusion

Composition of specializers is another step in proving the effectiveness of a pattern-based approach for optimizing programs. By reducing compute heavy sections of programs into specific patterns, efficiency programmers can focus on high-performance code generation using specializers as a vehicle for delivering these capabilities to application programmers. Efficiency level programmers can now reduce cross pattern optimizations to composition operations like fusion to further increase the performance of programs composed of multiple specializers calls.

In our case study of composing multiple instances of Sepya as well as stencils with other specializers have shown that this approach can result in dramatic performance increases in real world cases. The design behind fusion lays the groundwork for further work with other compositional patterns, as well as providing seamless integration for tuning across these optimizations.

Acknowledgments

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

References


