Automated Mapping of Task-Based Programs onto Distributed and Heterogeneous Machines

Automated Mapping of Task-Based Programs onto Distributed and Heterogeneous Machines

Thiago S. F. X. Teixeira
Stanford University
Stanford, CA, USA
thiagoxt@cs.stanford.edu

Alexandra Henzinger
Massachusetts Institute of Technology
Cambridge, MA, USA
ahenz@csail.mit.edu

Rohan Yadav
Stanford University
Stanford, CA, USA
rohany@cs.stanford.edu

Alex Aiken
Stanford University
Stanford, CA, USA
aiken@cs.stanford.edu

ABSTRACT

In a parallel and distributed application, a *mapping* is a selection of a processor for each computation or task and memories for the data collections that each task accesses. Finding high-performance mappings is challenging, particularly on heterogeneous hardware with multiple choices for processors and memories. We show that fast mappings are sensitive to the machine, application, and input. Porting to a new machine, modifying the application, or using a different input size may necessitate re-tuning the mapping to maintain the best possible performance.

We present *AutoMap*, a system that automatically tunes the mapping to the hardware used and finds fast mappings without user intervention or code modification. In contrast, hand-written mappings often require days of experimentation. AutoMap utilizes a novel *constrained coordinate-wise descent* search algorithm that balances the trade-off between running computations quickly and minimizing data movement. AutoMap discovers mappings up to 2.41× faster than custom, hand-written mappers.

KEYWORDS

Parallel Programming, Mapping, Autotuning, Distributed Systems, Heterogeneous Hardware, High Performance Computing

1 INTRODUCTION

Modern compute platforms are fundamentally distributed, offering multiple places where computation may be performed and multiple distinct memories where data may be placed. The *mapping problem* is finding an assignment of computation to processors and of data to memories such that an application achieves good performance. We show that a new automatic approach, AutoMap, can achieve substantial speedups over even custom, hand-written mapping strategies with much less programmer effort.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607079

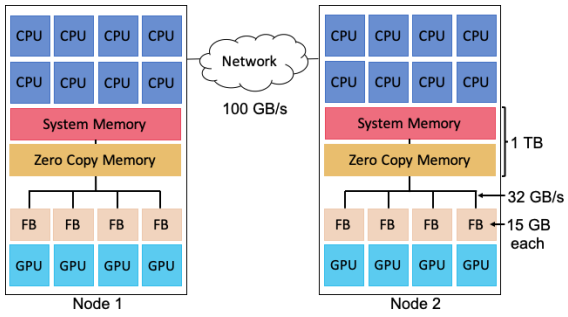


Figure 1: Sample two-node heterogeneous machine, with 2 kinds of processors (CPU cores and GPUs) and 3 kinds of memories (System, Zero-Copy, and Frame-Buffer).

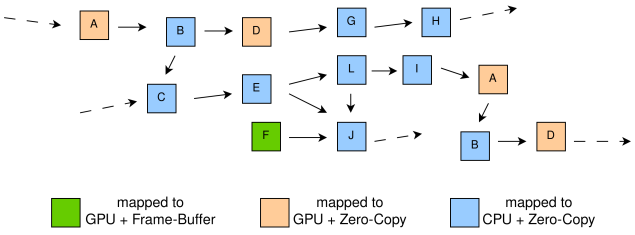


Figure 2: Partial dependence graph of a multi-physics application, and a mapping discovered by AutoMap on Shepard cluster. Each letter corresponds to a different task.

When the compute platform is a distributed system of multiple nodes, computation and data must also be distributed, or mapped, across the nodes of the machine to achieve good performance. However, the presence of accelerators such as GPUs has made mapping a concern even within a single node, where the typical configuration of a multicore CPU with several attached GPUs is itself a smaller distributed system. Clusters of accelerated nodes, illustrated in Figure 1, have multiple levels of distribution, heterogeneous processors, and a complex hierarchy of distinct memories.

For example, Figure 1 shows three *kinds* of memory: *System memory*, addressable only by the CPUs (one per socket), *Frame-Buffer*

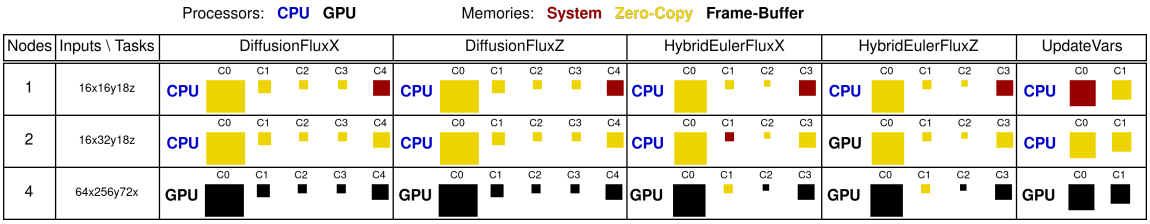


Figure 3: Partial visualization of the best mappings found by AutoMap for a subset of HTR's tasks for two different inputs on 1, 2, and 4 nodes of Shepard cluster. Tasks are assigned to CPU or GPU. Collection arguments are in red, black, or yellow, representing the mapping of collection argument C_i to Zero-Copy, Frame-Buffer, and System memory, respectively. The rectangles under each collection argument represents its relative size in bytes to the largest one of the application.

memory, addressable only by the GPUs, and *Zero-Copy memory*, addressable by both. A GPU computation t_1 that accesses data c placed in Zero-Copy memory will generally run more slowly because of the increased latency and decreased bandwidth incurred by accesses to Zero-Copy memory instead of Frame-Buffer memory. However, if a subsequent computation t_2 that runs on a CPU, or even on a different GPU, also accesses c, then placement in Zero-Copy memory could be faster than alternatively placing c in Frame-Buffer memory for t_1 and then copying the updated c to a memory addressable by t_2. Similarly, if another computation t_3, concurrent to t_1 and executing on the same GPU, intends to access data placed in Frame-Buffer memory, the Frame-Buffer memory may not be large enough to also hold c. To select the fastest memory assignment for c, we must know the costs for each of these mapping choices. Real applications have exponentially many combinations of such mapping decisions, made complex by the dependencies between application components, the different speeds of the communication links, and the capacity constraints of hardware resources.

By far the most common approach to addressing the mapping problem is to use greedy heuristics in the runtime system, such as always mapping tasks to GPUs if there is a GPU variant and always mapping task arguments to the closest memory to the chosen processor that has enough capacity. These heuristics are not suitable for all applications to achieve high performance, so some systems provide mechanisms for programmers to affect the mapping, and at least one provides a full interface allowing applications to control mapping decisions [6]. Hand-written mappings use knowledge about the application and target machine to achieve higher performance than heuristic-based mappings chosen by runtime systems. However, experimenting with mappings by hand requires deep knowledge of the application and target machine, and in our experience can require anywhere from a day to a few days for a complex application.

There is a significant body of previous work on mapping for the case where every processor has a single memory that it can access [22, 33, 38]. In this scenario, the mapping of tasks and collections is unified: the choice of processor for a task fully determines where the data must be placed. The mapping problem where there are degrees of freedom in the choice of memory for data is more general, and solving the problem outlined above—whether two tasks should share a co-located collection or the application should incur an extra copy—is central to our approach.

To solve the mapping problem in this more general setting, we present *AutoMap*, a system that automates and optimizes the mapping of tasks to processors and collections to memories on parallel, heterogeneous, and distributed machines (Section 3). AutoMap is used in an offline search to test many different mappings with the application and return the fastest mapping found.

The core of AutoMap is a new search algorithm called *constrained coordinate-wise descent* or CCD (Section 4). CCD alternates between optimizing the mapping of tasks and the mapping of data and manages the trade-off between mapping tasks to run as fast as possible and mapping data accessed by multiple tasks to minimize communication. AutoMap performs a dynamic analysis, which ensures that the search knows the actual costs of executing tasks and copying data, rather than relying on static estimates. As we will see, individual mappings can have significant variation in performance from run to run, necessitating multiple executions to obtain reliable estimates of the performance mean and variance.

We evaluate AutoMap on five benchmark applications on two different clusters (Section 5). These benchmarks include a large multi-physics application [12] and a multi-fidelity ensemble computational fluid dynamics application, where computation is performed on data of different resolutions. In all experiments, AutoMap produces mappings that are at least as fast as the hand-written mappings. In some cases, the mappings found by AutoMap are up to 2.41× than the hand-written mappers. The results indicate that the mapping problem can be automated with equal to, and in some cases, better than human performance, relieving programmers from the burden of writing a custom, hand-written mapper.

To summarize, we make the following contributions:

- We formalize the mapping problem for task-based programming models on machines where the mapping of tasks does not fully determine the mapping of data.
- We describe *AutoMap*, a system that optimizes an application's mapping by dynamically searching the space of possible mapping decisions (Section 3).
- We design a search algorithm tailored to finding high-performance mappings for task-based programs called *constrained coordinate-wise descent*, which explicitly balances the trade-off between computation and communication costs, and find mappings that outperform the other search algorithms by up to 1.57× (Section 4).

- We implement and evaluate AutoMap, showing that it finds fast mappings that often outperform hand-written mappings without any code modification or user intervention. We demonstrate that high-performance mappings depend highly on the machine configuration, input size, and other realistic constraints. AutoMap also outperforms standard strategies of mapping a multi-fidelity ensemble CFD application to heterogeneous systems and discovers mappings up to 50× faster on memory-constrained experiments by mapping a subset of the collections to slower memories (Section 5).

We do not claim that AutoMap is the best possible solution to the mapping problem. While our work demonstrates that jointly considering the assignment of tasks to processors and data to memories is necessary to automatically discover good mappings on current distributed, heterogeneous machines, we will discuss additional decisions that we do not currently consider that could be incorporated to further improve the quality of mappings. And while CCD is effective, we view it as a first step; there may well be improved algorithms that are faster, find better mappings, or both.

2 MODEL AND BACKGROUND

We model a machine $\mathcal{M}$ as a graph where the nodes are *processors* and *memories*. Each processor has a kind (either CPU or GPU in this paper) and each memory has a kind and a capacity in bytes. The edges are of two types: An edge between a processor p and a memory m indicates that m is addressable by p, and an edge between two memories indicates that there is a communication channel between the two memories.

Task-based systems are a common programming model for distributed programming with accelerators. In scientific computing, task-based systems include PaRSEC [8], StarPU [4], Legion [6], recent versions of OpenMP [26], OmpSs [13], COMPSs [21], and PyCOMPSs [37]; in data analytics, widely-used task-based programming models include Spark [41], TensorFlow [1], PyTorch [27], Dask [11], and Ray [23].

While task-based programming models vary considerably, there is agreement on aspects important for mapping. Programs represent and are translated into acyclic dependence graphs that are executed at runtime: the nodes are *tasks* and edges represent a partial order on task execution, as illustrated for a multi-physics application in Figure 2. Tasks are functions of named *data collections* that they may read, write, or both. In all task-based systems we know of, these collections are some variation of a multi-dimensional array. The important aspect for mapping is that tasks require each collection argument be placed in a single memory. To run on a processor kind, a task must have a *variant* for that processor kind—i.e., there must be object code for the task that executes on that type of processor.

For a machine $\mathcal{M}$ and task graph $\mathcal{G}$, a *mapping* f is a function of type $\textit{tasks} \times \textit{collections} \to \textit{processors} \times \textit{memories}$ such that

- for each collection argument c_i of a task t in $\mathcal{G}$, $f(t, c_i) = (p, m_i)$, where m_i is a memory in $\mathcal{M}$ accessible to p.
- t has a variant for p's kind.

AutoMap aims to solve the *mapping problem*, which entails finding a mapping f that minimizes an application's execution time. Throughout this work, we refer to *optimizing* mappings as decreasing their execution times.

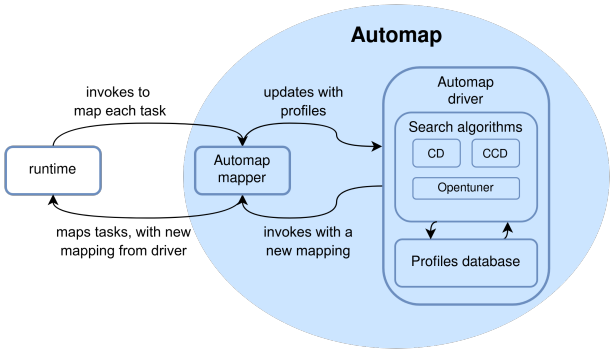


Figure 4: Architecture of AutoMap.

A mapping may imply data movement not explicit in the task graph. Given a mapping f, consider tasks t_1 and t_2 where t_2 depends on t_1, t_1 writes a collection c, and t_2 reads c. Let $(\_, m_1) = f(t_1, c)$ and $(\_, m_2) = f(t_2, c)$. There is no requirement that $m_1 = m_2$, but if $m_1 \neq m_2$, then once t_1 has finished executing, c must be copied from m_1 to m_2 before t_2 can begin execution.

Figure 2 shows a fast mapping for a portion of a multi-physics application, illustrating that the decisions of which tasks should run on which processors and what memories should be used for data can be non-trivial in real applications. Figure 3 shows partial fast mappings found by AutoMap for the same multi-physics application on 1, 2, and 4 nodes and different inputs. These mappings are 1.44×, 1.5×, and 1.11× faster than the default strategy, respectively. They present non-trivial choices, such as the fastest one on 4 nodes, which (the full mapping) places 9 collection arguments in Zero-Copy memory.

3 AUTOMAP

AutoMap consists of two components shown in Figure 4:

(1) the *mapper*, which interacts with the runtime system to coordinate which mapping is used and to obtain performance profiles;
(2) the *driver*, which contains the search algorithms and the profiles database used to decide which mapping should be executed and evaluated next.

For this work, we use the Legion runtime system because it exposes an API to dynamically control mapping decisions [6]. In our implementation, the mapper is an implementation of Legion's mapping interface, and the search algorithms are pluggable components that can be replaced to implement a different algorithm. While our implementation is in Legion, we believe AutoMap is applicable to other systems using a task-based programming model. For another system to use AutoMap, it must support the following features: a) tasks with separate data collections as arguments; and b) the existence of per-collection dependence information between tasks (this information can either be user-provided or computed by the runtime system). StarPU [4] and PaRSEC [8] are systems that support these features. While the details of the mapping interfaces are different for each task-based runtime system, we believe the overall design and the algorithms we propose are applicable to other task-based systems.

Algorithm 1: Constrained Coordinate-wise Descent

CCD $(\mathcal{M}, \mathcal{G})$
  input : machine model $\mathcal{M}$, dependence graph $\mathcal{G}$
  output: the fastest mapping found f, its performance p
  Initialize f to starting point, p to its performance;
  $C \leftarrow$ induced graph over collections in $\mathcal{G}$;
  foreach rotation r from 1 to *num_rotations* **do**
    /* for each task t and collection c, gather all collections that overlap with c */
    $O \leftarrow \{(t,c) \mapsto (t,c) \cup \{(t',c') : (c,c') \in C\}\}$;
    foreach task $t \in \mathcal{G}$, ordered by runtime **do**
      $f, p \leftarrow$ `OptimizeTask`$(t, f, p, \mathcal{M}, O)$;
    Remove $\frac{original\_num\_edges}{num\_rotations-1}$ lightest edges from C
  return (f, p)
OptimizeTask $(t, f, p, \mathcal{M}, O)$
  /* optimize distribution setting */
  foreach distribution setting $d \in$ {true, false} **do**
    $f, p \leftarrow$ `TestMapping`$(f[$*distribute t according to d*$], f, p)$
  /* optimize processor kinds */
  foreach processor kind $k \in \mathcal{M}$ **do**
    /* optimize memory kinds */
    foreach collection $c \in t.collections$, ordered by size **do**
      foreach memory kind $r \in \mathcal{M}$ accessible from k **do**
        $f' \leftarrow f$[map t on k, c on r];
        $f'' \leftarrow$ `ColocationConstraints`(f', t, c, k, r, O);
        $f, p \leftarrow$ `TestMapping`(f'', f, p);
  return (f, p)
TestMapping (f_{test}, f, p)
  p_{test} = `EvaluateMapping`(f_{test});
  if $p_{test} < p$ **then**
    return (f_{test}, p_{test})
  return (f, p)

Algorithm 2: Co-location Constraints

ColocationConstraints (f, t, c, k, r, O)
  input : current mapping f, task t, collection c, processor kind k, memory kind r, collections overlapping map O
  output: new mapping f'
  $f' \leftarrow f$
  $t\_check, c\_check \leftarrow \emptyset, \emptyset$
  /* map all collections overlapping with c to r */
  foreach $(t_i, c_i) \in O[(t,c)]$ **do**
    $f' \leftarrow f'[c_i \neq c$: map c_i on $r]$;
    /* record tasks with moved collections */
    $t\_check \leftarrow t\_check \cup \{t_i\}$
  /* after changing the mapping of all collections that overlap with c, the mapping of those collections' tasks may change, requiring iteration to a fixed point */
  while $t\_check \neq \emptyset$ *or* $c\_check \neq \emptyset$ **do**
    /* adjust mappings for all tasks with moved collections */
    while $t\_check \neq \emptyset$ **do**
      $t_i \leftarrow t\_check.pop()$
      foreach $c_i \in t_i.collections$ **do**
        if c_i*'s mem kind not addressable by* t_i*'s proc kind* **then**
          $f' \leftarrow f'[t_i \neq t$: map t_i on $k]$
          $c\_check \leftarrow c\_check \cup \{(t_i, c_i)\}$
    /* adjust mappings for all collections with moved tasks */
    while $c\_check \neq \emptyset$ **do**
      $t_i, c_i \leftarrow c\_check.pop()$
      $m \leftarrow$ select a mem kind addressable by t_i's proc kind
      if $(t,c) \in O[(t_i, c_i)]$ **then**
        continue
      $f' \leftarrow f'$[map c_i on m];
      foreach $(t_j, c_j) \in O[(t_i, c_i)]$ **do**
        if $(t_j, c_j) == (t_i, c_i)$ *or* c_j*'s mem kind* $== m$ **then**
          continue
        $f' \leftarrow f'$[map c_j on m];
        if m *not addressable by* t_j*'s proc kind* **then**
          $t\_check \leftarrow t\_check \cup \{t_j\}$
        $c\_check \leftarrow c\_check \setminus \{(t_j, c_j)\}$
  return f'

3.1 Extensions to the Mapping Problem

We extend the mapping problem to address features of realistic task-based programs. First, many of the applications we consider use *group tasks*, which are sets of independent tasks launched in a single operation. All elements of a group are instances of the same task. We extend our notion of mapping to include whether a group task should be executed entirely on the initial *leader* node or whether the tasks in the group should be distributed in a blocked fashion among all machine nodes. We do not consider different strategies for distributing group tasks across machine nodes. For uniformity, we assume programs have only group tasks by treating individual tasks as groups of size one. Second, we allow mappings to fail at runtime if a collection assignment exceeds the capacity of the physical memory. It is possible to generalize a mapping $f(t, c)$ in a straightforward manner to a priority list of memories, all addressable by the chosen processor, where the first memory that can hold c will be used. For simplicity of presentation, we do not discuss this generalization, but our implementation handles this more general form of mapping.

3.2 The Search Space

To estimate the size of the search space of possible mappings, we momentarily make the simplifying assumption that all tasks can be assigned to all processor kinds and all data collections can be assigned to the same number of memories for each processor kind. There are then $P^T M^C$ possible mappings, where P is the number of processor kinds, T is the number of tasks, M is the number of memory kinds, and C is the number of collection arguments. In practice different processors have different numbers M of memories that they can address, but $M \geq 2$ for all processor kinds in the machines we consider.

Such an immense search space cannot be traversed exhaustively. Thus, we factor the problem into two parts: A search over the *kinds* of processors/memories to use and runtime logic to select specific processors/memories of the appropriate kind. This structure reduces the search space without sacrificing high-performance mappings. Specifically, the driver invokes a search algorithm to choose the kind of processor for each task, and the mapper distributes tasks in a deterministic, blocked fashion among processors

of the selected kind. There are more sophisticated strategies for distributing the tasks. Autotuning the distribution of groups of tasks across processors is an opportunity for further improvements and potential future work for AutoMap. Similarly to mapping tasks, the driver invokes a search algorithm to choose the kind of memory for each collection, and the mapper instantiates each collection in the the memory of the desired kind that is closest to the selected processor. Finally, tasks in a group task are assigned the same mapping of processor kind and collection kinds.

Based on our factorization of the mapping problem, AutoMap searches to find a mapping function with the signature $tasks \times collections \rightarrow bool \times processor\ kind \times memory\ kind$. Then $f(t, c) = (d, k_p, k_m)$ where the boolean d indicates whether the group task is distributed or not, k_p is the processor kind for t, and k_m is the selected memory kind for c.

3.3 AutoMap Usage

AutoMap requires no modification to the application. The input is a file containing the search space and machine model representation containing all or a subset of tasks and data collections of the target application. The file representing the search space is generated automatically by running and profiling the application once. Given the representation of the search space, AutoMap then begins an offline search for mappings, and invokes the application automatically to evaluate potential mappings. While in this work we optimize execution time, AutoMap is suitable for minimizing other metrics (e.g., power consumption). The search always has a current best mapping, and so the search can be time-limited if desired.

4 SEARCH ALGORITHMS

The AutoMap framework supports the use of different search algorithms to propose candidate mappings for evaluation. AutoMap includes coordinate-wise descent, OpenTuner, and we introduce *constrained coordinate-wise descent*, a novel algorithm tailored to finding solutions for the mapping problem.

4.1 Coordinate-wise Descent (CD)

Coordinate-wise descent considers each task in turn, optimizing the mapping of that task and all of its collections (Algorithm 1 excluding line 17). When considering each dimension, all other decisions are held constant; i.e., one mapping decision is changed at a time. Specifically, for a group task, CD first greedily optimizes its distribution setting, then the task's processor kind, and finally the memory kind of each collection. Thus, CD's runtime is linear in the number of tasks and the number of collections in the application.

CD loops over tasks (Algorithm 1 line 6) from longest running to shortest and over collections (Algorithm 1 line 14) from largest to smallest. Intuitively, this ordering should accelerate convergence to a high performance mapping, as the best mapping(s) of expensive tasks and large collections are less likely to be influenced by the rest of the application's mapping.

Starting point. We pick a starting point that is good for many applications: Group tasks are distributed across all nodes, all tasks with GPU variants are placed on GPUs, and all collections (that fit) are placed in Frame-Buffer memory.

4.2 Constrained CD (CCD)

While there are many dimensions to the search, our experience is that often the central trade-off is the tension between tasks running as fast as possible and minimizing data movement. Algorithm 1, *constrained coordinate-wise descent*, iteratively optimizes the placement of tasks and of data to maximize performance. The search begins with a high penalty for data movement that is gradually relaxed to balance the costs of compute and data movement in an increasingly fine-grained way.

CCD optimizes an initial mapping in a sequence of N *rotations*. In each rotation, CCD runs a full CD, and the best mapping of rotation r_i is the starting one of rotation r_{i+1}.

Minimizing data movement through constraints. As part of the search, AutoMap maintains a dependence graph $\mathcal{G}$ of tasks; in our implementation, this graph is obtained from runtime profiling information. From this graph we induce a graph $C = (V, E)$ on the collections, where each collection $c \in V$ and there are edges between collections that overlap: $(c_1, c_2) \in E$ iff $c_1 \cap c_2 \neq \emptyset$. The *weight* of the edge is $|c_1 \cap c_2|$.

Intuitively, collections overlap when they reference non-disjoint components of the same logical data structure. For example, the halo regions in a partitioned stencil computation overlap, as each halo region references data used by multiple tasks. CCD models this sharing of data directly, and uses it to guide the search.

CCD enforces two constraints for each mapping it considers:

(1) A task argument is mapped to a memory visible to the task's processor.
(2) If $(c, c') \in E$, then c and c' are mapped to the same memory kind.

The first constraint is necessary for correctness (or else the mapping will not be executable). The second constraint is the co-location constraint on overlapping collections to minimize data movement, described in Algorithm 2. At each step of a rotation, CCD has a current mapping f to which it makes one change, creating a mapping f'. If f' violates constraint (1) because task t cannot access collection argument c, then t is moved to a processor kind that can access memory kind c. If f' violates constraint (2) because collection c was moved to memory kind k and $(c, c') \in E$ where c' is mapped to some different memory kind, then c' is also moved to memory kind k (Algorithm 2 lines 4-6). Adjusting the mapping to locally satisfy the constraints may violate constraints for different tasks and collections, so these two rules are iteratively applied until the mapping f'' satisfies both constraints globally (Algorithm 2 lines 7-26). This iterative process converges, as the limiting case is that all tasks/collections are mapped to the same processor/memory kind.

After each rotation, CCD prunes a fraction $1/(N-1)$ of the lowest-weight edges from C to relax the data movement constraint, where N is the total number of rotations to be performed. In CCD's last rotation over all tasks, all edges from C have been pruned, so all constraints on data collection placement are lifted. The purpose of this constraint relaxation approach is twofold: First, by initially constraining overlapping collections to be mapped in the same way, we collapse their mapping into a single decision, simplifying the search space. Second, this approach guarantees that CCD tries to find the fastest mappings at multiple different thresholds of

tolerance for data movement. In our experiments, we set the number of rotations to 5 and prune 1/4 of the edges of C at the end of each rotation.

As we will see in our evaluation (Section 5.3), CCD performs the best among these algorithms, finding mappings that are at least as fast, often in less time. We believe two features stand out as the most important.

(1) CCD is systematic, deterministically enumerating a wide variety of different mappings. While hardly exhaustive, exploration of each dimension of the search space is guaranteed.
(2) Constraints on data movement allow CCD to make coordinated moves, deciding to place multiple collection arguments of different tasks in the same memory to minimize data movement. Search strategies that make purely local decisions about individual collections are much less likely to explore such colocations.

Consider the multi-physics solver described in Figure 5: two different group tasks operate on two large shared collections. The fastest known strategy for some of the inputs (and the strategy discovered by AutoMap) is to place these two collections in Zero-Copy memory to minimize data movement. Having all tasks place the collections in Frame-Buffer memory would be faster than having only some of the tasks place them in Frame-Buffer memory and others place them in Zero-Copy memory, which slows the execution of some of the tasks (due to the slower accesses to Zero-Copy memory) while still incurring substantial data movement. In this scenario, algorithms that optimize the mapping of a single task at a time and accept only strict improvements may fail to converge to the fastest known mapping, as there is no sequence of strictly improving mapping decisions that progress from the collections all being in Frame-Buffer memory to all being in Zero-Copy memory. Even an algorithm like OpenTuner is unlikely to find this solution as it requires accepting multiple cost-increasing moves. In our experiments, the alternative algorithms considered here fail to find the fastest known mapping. CCD succeeds, as it makes a coordinated decision to jointly map these collections in its first rotation.

4.3 OpenTuner

AutoMap maps the search space onto OpenTuner search data types. OpenTuner uses ensembles of search techniques, which run simultaneously, testing candidate mappings. Techniques that find better mappings have a larger budget to select the subsequent mappings for evaluation, while the ones that perform poorly evaluate fewer mappings. It is not possible to represent constrained search spaces in OpenTuner. Automap tries to overcome this limitation by returning a high value whenever OpenTuner suggests an invalid mapping (e.g., selects a memory for a data collection that is not accessible by the processor chosen for the task), so it does not suggest similar mappings in the future, although that is not guaranteed.

5 EVALUATION

We evaluate AutoMap's overall performance and individual components on five benchmark applications described in Figure 5. The selected benchmarks use Legion's task-based programming model for distributed heterogeneous machines. The benchmarks are implementations for electrical circuit simulation (Circuit) [6], structured stencil (Stencil) [40], Lagrangian hydrodynamics simulation (Pennant) [16], multi-physics solver (HTR) [12], and multi-fidelity ensemble computational fluid dynamics (Maestro). HTR and Maestro are production-level applications used in large-scale simulations. We discuss the performance of mappings that AutoMap is able to find, and individually compare the CCD algorithm to other reasonable algorithms that could be used for the mapping problem.

Experimental Setup. We perform experiments on two different clusters of accelerated nodes: Shepard from Stanford University HPC Center and Lassen from Lawrence Livermore National Laboratory. Each node of Shepard has 2 Intel Xeon Platinum 8276 2.2 GHz CPUs with 28 cores each, 196 GB of RAM, and one NVIDIA Tesla P100 with 16 GB of Frame-Buffer. Each node of Lassen has 2 IBM Power9 3.45 GHz CPUs with 22 cores each (only 20 usable), 256 GB of RAM, and four NVIDIA Volta V100 GPUs with NVLink 2.0 (AC922 server) and 16 GB of Frame-Buffer each. We separate 8 cores to the Legion runtime, and leave the rest of the CPU cores and all GPUs on each system for the application.

In our experiments, we configure AutoMap to use the following three memory kinds: Frame-Buffer memory (a high-throughput memory, local to each GPU), Zero-Copy memory (a pinned memory on the host that all GPUs and CPUs can access), System memory (CPU-accessible RAM, one per socket). We reserved 60 GB of host memory per node for the Zero-Copy memory. The CCD experiments ran for five rotations, which we found to perform the best in practice. More rotations increased the search time without improving performance, and fewer rotations made CCD perform similarly to CD. We selected five as a trade-off between finding faster mappings and search time, but the optimal number of rotations can vary depending on the application, machine, and input. During the search, each mapping ran 7 times, and the average was used to select the best mapping for each experiment. As a final step of the search, the applications were executed with each of the top 5 mappings 30 times; we report results for the mapping with the fastest average runtime.

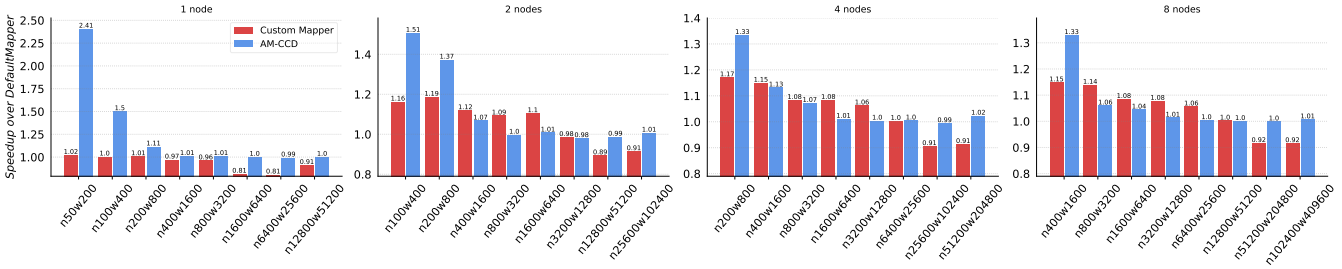
Baselines for comparison. We evaluate AutoMap's overall performance by comparing it to Legion's default mapper and to custom, hand-written mappers for each application. The default mapper is packaged with the Legion runtime system and is invoked if no mapper is provided by the user. It applies a set of fixed heuristics to determine processor and data placement, such as using GPUs and placing collections in the highest bandwidth memories whenever possible. As all applications considered include GPU variants for each task, in our experiments the default mapper places all tasks on the GPUs and all of their collections in Frame-Buffer memory.

The hand-written mappers are application-specific custom mappers implemented by a domain expert, often requiring multiple days of effort. Domain experts rely on their understanding of the application and of the target computer architecture to select the mapping. For the applications we consider, hand-written mappers generally follow a similar strategy as the default mapper but sometimes place large or shared data in Zero-Copy memory and move less important tasks to CPUs to utilize more compute cores and conserve Frame-Buffer memory.

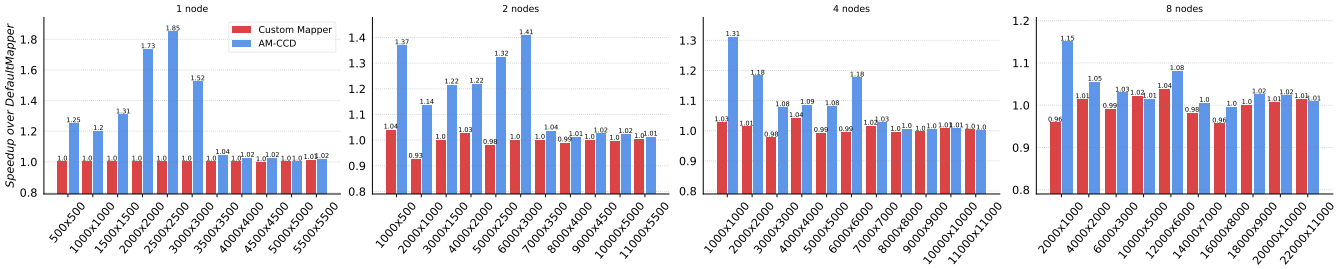
As presented in Figure 5, the benchmark applications are extensively optimized codes that have appeared in prior publications, along with their hand-written mapper implementations. While the

Application	Description	Tasks	Collection Arguments	Search Space Size	CCD Search Time (hours)
Circuit	Electrical circuit simulation [6]	3	15	$\sim 2^{18}$	1-2
Stencil	2D structured stencil [40]	2	12	$\sim 2^{14}$	1-2
Pennant	Lagrangian hydrodynamics calculation [16]	31	97	$\sim 2^{128}$	1-4
HTR	Multi-physics solver [12]	28	72	$\sim 2^{100}$	4-7
Maestro	Multi-fidelity Ensemble CFD	13 (only LFs)	30	$\sim 2^{43}$	1-2

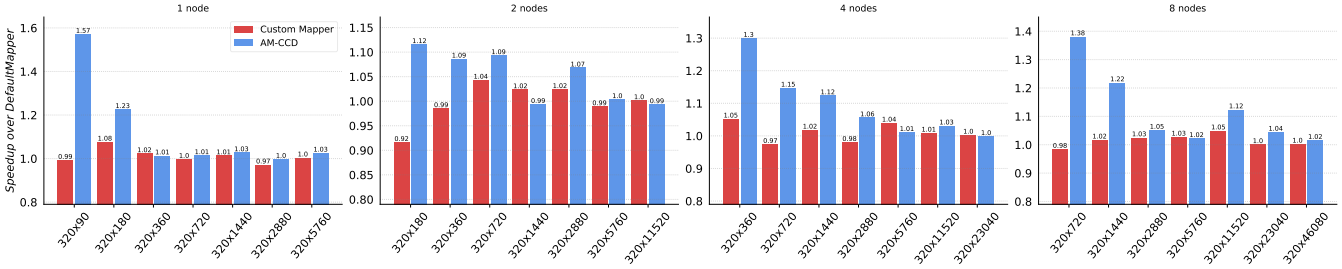
Figure 5: Description of the five benchmark applications.



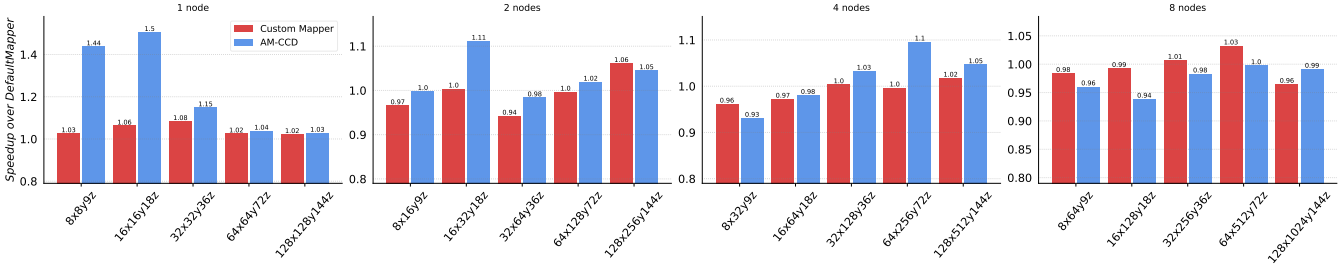
(a) Circuit



(b) Stencil



(c) Pennant



(d) HTR

Figure 6: Performance evaluation of custom mapper and AutoMap relative to the default mapper on Shepard cluster.

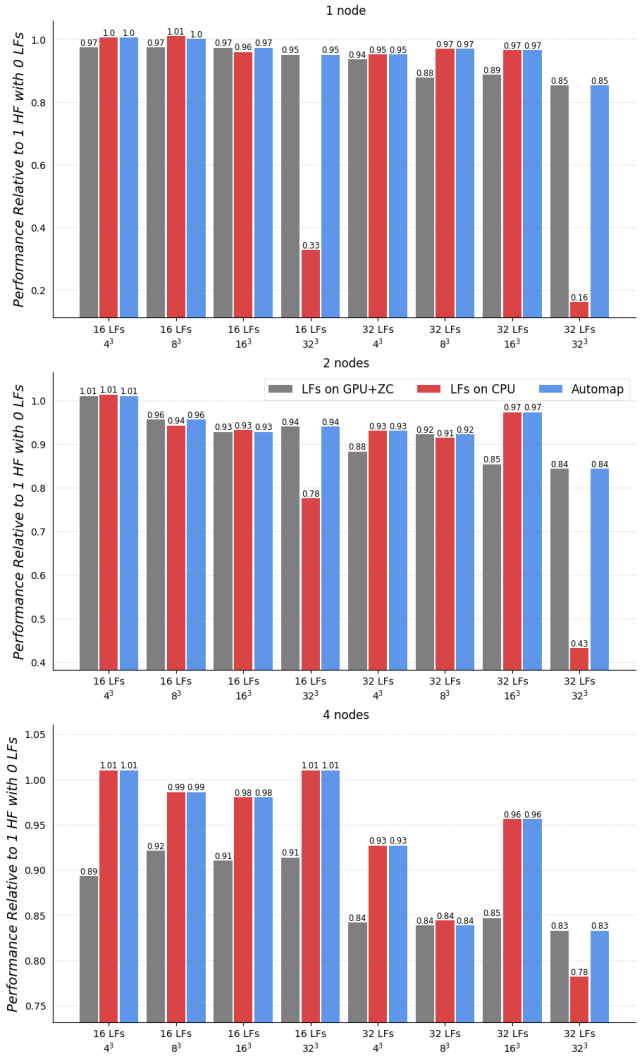


Figure 7: Multi-fidelity Ensemble CFD varying the number of low-fidelity samples and resolution.

hand-written mappers were not necessarily tuned exactly for Shepard and Lassen, the optimizations they implement target similar multi-GPU machines, and have been heavily optimized. In our experiments, we use the benchmark codes as-is, and only change the mapping strategy when we compare the performance of AutoMap, the default mapper, and the benchmark application's custom mapper. The more complex applications require a longer search.

Results. The comparisons between the three mappers (default, custom, and AutoMap) for a variety of inputs are shown in Figure 6. These experiments were conducted on Shepard. Each application was weak-scaled when moving to multiple nodes. In each figure, we plot the speedup achieved by AutoMap and the custom mapper over Legion's default mapper. These plots represent improvement achieved by using AutoMap or a custom mapper over the default mapper.

We see that AutoMap finds better or equal mappings to the default mapper. Except for some inputs of Circuit and HTR, AutoMap also generally matches or outperforms applications' custom mappers. Circuit's custom mapper decomposes group tasks in a blocked manner across nodes in the machine, while AutoMap uses a round-robin strategy. AutoMap does not currently consider different ways to decompose group tasks, allowing the custom mapper to find a better mapping in some cases.

The results for Stencil are in Figure 6b, and the most significant speedups arise when AutoMap places the tasks in the CPU. Moreover, use different combinations of collection arguments into System and Zero-Copy memories. It is crucial to notice that placing data in System and Zero-Copy is not the same on multi-socket systems (Shepard and Lassen have two sockets). In such systems, two independent allocations are created in the System memory to be used for tasks running on each socket. Therefore, data accessed by tasks in a different socket requires a data transfer from one allocation to the other by the Legion runtime, whereas Zero-Copy is a single allocation addressable by all the processors, including GPUs.

Pennant results are in Figure 6c. The most significant speedups achieved on Pennant are due to mixed mappings with up to 26 of the 31 tasks on the CPU and 4 collection arguments in Zero-Copy. As the input size increases, AutoMap places more tasks on the GPU and data on the Frame-Buffer memory. We see similar results on HTR (Figure 6d), and the biggest AutoMap gains are because of placing tasks on the CPU and the data on Zero-Copy. AutoMap, however, found a mapping with a 10% performance increase (see Figure 3, input 64x256y72z on 4 nodes) that places 9 collection arguments on the Zero-Copy memory and 2 tasks on CPU.

AutoMap is able to find mappings that outperform custom mappers on some configurations of each application in our benchmark suite. Custom mappers are usually implemented for a specific configuration and input, limiting the number of situations that the custom mapper can be applied to and achieve peak performance. AutoMap helps users discover efficient mapping strategies to tune their custom mappers to new application configurations, or improve the performance of their application without even writing a customized mapper.

5.1 Multi-fidelity Ensemble CFD

Maestro is a multi-fidelity ensemble computational fluid dynamics (CFD) solver resolving the single-component compressible Navier-Stokes equations with explicit finite-difference schemes. We configured Maestro to utilize a bi-fidelity ensemble comprised of one high-fidelity and multiple low-fidelity samples, all operating on a 3D volume.

The inherent cost of high-fidelity simulation limits its usage in engineering applications requiring many samples, such as uncertainty quantification or optimization. In a multi-fidelity setting, low-fidelity simulations are used to complement high-fidelity simulations by enabling the collection of many samples with lower computational cost but at reduced accuracy [14].

The processing of a single high-fidelity sample requires significant computational power and utilizes large amounts of memory. Thus, the high-fidelity simulation is mapped onto the GPUs and its

collection arguments fill up the entire Frame-Buffer memory. The low fidelity simulations, however, can be assigned to the CPU or the GPU, and their collection arguments can be mapped to Zero-Copy memory or System memory. The main goal of the Maestro developers is to run low fidelity simulations that do not affect the performance of the expensive high-fidelity simulation by utilizing additional resources available on the machine. The number and resolution of the low fidelity simulations are tuned according to the performance of the high-fidelity when executing alone.

Figure 7 presents AutoMap's results along with two standard strategies of mapping the tasks for the low-fidelity simulations: 1) mapping all low-fidelity tasks and collections to CPUs and System memory, and 2) mapping all low-fidelity tasks and collections to GPUs and Zero-Copy memory. We compare these strategies against the runtime of Maestro without any low-fidelity simulations running, to see which strategies impact the execution time of the high-fidelity simulation the least. Figure 7 plots the amount of performance degradation when running different amounts of low-fidelity simulations against running only the high-fidelity simulation, so values close to 1.0 indicate the low-fidelity simulations do not affect the performance of the high-fidelity simulation. Our results show that the simple strategies are not always optimal, and can lead to performance degradation of the high-fidelity simulation for certain low-fidelity simulation counts and resolutions. AutoMap outperforms the other strategies by placing all the relevant tasks in the same processor kind. For instance, on two nodes, the execution of 16 low-fidelity simulations and resolution 32^3 is faster when mapped to GPUs and Zero-Copy memory, but 32 low-fidelity simulations and 16^3 is faster when mapped to CPUs. These decisions are non-trivial as they also depend on the hardware used. Different processors and GPUs may present different results. Furthermore, the low-fidelity simulations are adaptable and change according to the simulation requirements. Automap can relieve the burden on developers of these mapping decisions and finds the best results for each configuration.

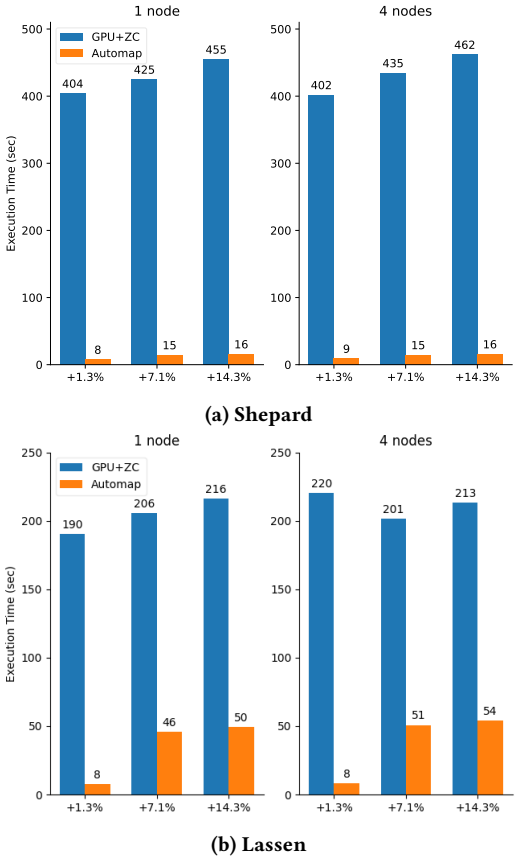


Figure 8: Pennant execution times on Shepard and Lassen clusters with inputs 1.3%, 7.1%, and 14.3% larger than the maximum input that fits completely in the Frame-Buffer. AutoMap provides speedup of at least 4× compared to all the data in the GPU Zero-Copy.

5.2 Memory Constrained Experiments

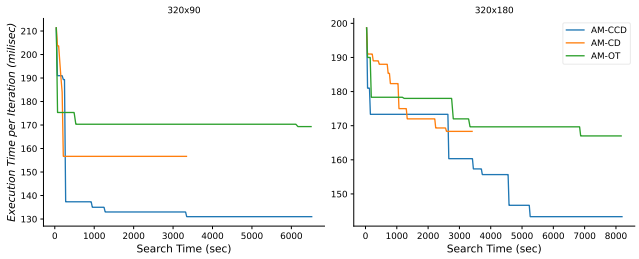
Users with limited resources may desire to run their applications with larger inputs than the ones that fit in the fastest memory of their fastest processor. The GPU Frame-Buffer memory, for instance, has the highest throughput, but the most limited capacity in modern systems. Selecting which collections to keep in the fastest memory and which collections to demote to larger, but slower memories is challenging, as complex applications may have dozens of collection arguments (Pennant has 97). This selection is also not robust—as the input size increases, the optimal selection may change.

The most straightforward approach is to place all data in a bigger but slower memory, which results in slow execution times. AutoMap's search is able to find a subset of the collections that fit in the fast memory to significantly speed up execution time. Our implementation of AutoMap is resilient to mappings that exceed the fastest memory's capacity, detecting when a mapping results in an out of memory error and moving on to a different mapping.

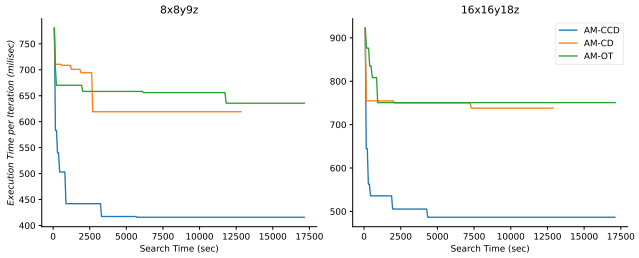
Figure 8 presents the execution time of Pennant with inputs larger than the maximum size that fits in the Frame-Buffer memory (320×40320 zones per GPU) for 1 and 4 nodes. On Lassen, the best mappings found by AutoMap place 7, 12, and 13 collection arguments in Zero-Copy for inputs 1.3%, 7.1%, and 14.3% bigger than the maximum Frame-Buffer capacity, respectively. Interestingly, it also placed 2 tasks on the CPU for the input +1.3% (1 and 4 nodes), and 1 task on the CPU for the inputs +7.1% and +14.3% (4 nodes). Shepard results are similar for the +1.3% input (1 and 4 nodes), but for +7.1% and +14.3% AutoMap placed all tasks on CPU and collection arguments in System memory. AutoMap finds mappings much faster than placing all collection arguments in Zero-Copy. The mappings discovered by AutoMap are up to 50× faster for the +1.3% input on one node on Shepard. As expected, as the input grows, the mappings discovered are slower as fewer collection arguments fit in the Frame-Buffer memory.

5.3 Search Algorithm Evaluation

We evaluate the choice of using the CCD algorithm instead of other algorithms discussed in Section 4.3, such as CD and OpenTuner. All three algorithms were given the same budget of time. CD is equivalent to the one rotation (the last one) of CCD and terminates



(a) Pennant



(b) HTR

Figure 9: Execution time per mapping evaluation (lower is better) as a function of search time for the three search algorithms used by AutoMap: OpenTuner, CD, and CCD.

earlier. In our experiments, we evaluate both the quality of mappings that each algorithm is able to find, as well as the amount of time each algorithm took to find the best mapping. We evaluate the algorithms on different problem sizes of the most complex of our benchmark applications, Pennant and HTR. The results for our experiments are in Figure 9, which shows the performance of the best mapping found by each algorithm plotted against the elapsed time of the search.

We see that CCD consistently finds mappings that outperform CD and OpenTuner by up to 1.57×. By explicitly considering the relationships between overlapping data collections, CCD can collapse the placement decisions of multiple collections into a single choice, while a system like OpenTuner has to make all of those decisions independently. CD operates without these constraints on collection placement, and thus is also unable to make these coordinated decisions.

Tailoring an algorithm to the mapping problem provides additional benefits in terms of search efficiency. We find that OpenTuner spends as little as 13% and up to 45% of the search time evaluating candidate mappings. In contrast, CCD and CD spend 99% of total search time evaluating candidate mappings. This difference arises due to the lack of domain knowledge that OpenTuner's generic algorithms possess about the mapping problem.

OpenTuner spends a significant amount of time suggesting invalid mappings (e.g., it places a task on CPU and one collection argument of the task in Frame-Buffer memory), which will incur an error during execution. AutoMap does not evaluate such mappings and returns a high value to OpenTuner, so it avoids suggesting similar mappings in the future. It is not possible to express constrained search spaces in OpenTuner. This limitation is seen in the number of mappings proposed by each search algorithm to AutoMap: For Pennant, CCD suggests 1941 mappings, and evaluates on average 460 mappings per search (the difference are repeated mappings). CD suggests 389 mappings, and evaluates on average 226 mappings, essentially performing the final rotation of CCD. OpenTuner suggests on average 157202 mappings, and evaluates 273 mappings, suggesting two orders of magnitude more mappings than CCD or CD.

Our results demonstrate that CCD is an effective search algorithm to target the mapping problem. CCD can find mappings that outperform other algorithms, and search the space of candidate mappings more efficiently than generic algorithms like OpenTuner.

6 RELATED WORK

Sequoia was the first task-based system to provide a programmer-controlled mapping interface [15]. Sequoia's static specification of application code and mapping was generalized in Legion [6], which has a dynamic mapping interface.

Task Scheduling for Heterogeneous Systems. The earliest works on task scheduling for heterogeneous clusters, such as the HEFT [38], MCT [22], and FCP [33] algorithms, focus on scheduling a task t on a processor taking into account processor speed, the cost of t, and the time needed clear each processor's current task queue. These heuristics assume a single memory in which data can be placed for a given processor. As we have already noted, when there are multiple memories, the choice can affect not just the time for t but also the cost of subsequent tasks that will use that data.

Work within the StarPU task-based system implements different scheduling strategies based on HEFT that consider data movement costs and prefetching for heterogeneous systems with multiple accelerators [3]. The described strategies place all of a task's data in a single memory and do not consider the impact of the data placement decisions on the cost of future tasks.

In contrast, AutoMap addresses the problem of jointly optimizing the choice of processor for each task and memory for each allocation. Considering the mapping of data collections as well as tasks introduces new difficulties, as the mapping of data collections can affect which processors are the best candidates to place the tasks that use those collections, and vice versa.

Machine Learning-based Mapping Strategies. Prior work translates multi-core code into OpenCL and uses a decision-tree classifier (learned from training data based on static compiler analysis) to estimate whether an application is profitable to run on GPUs [25]. Wang et al presents data sensitive and data insensitive machine-learning predictors for the number of threads and the scheduling policy for two different multi-core platforms [39]. In [25], the same processor is selected for the whole benchmark. These papers do not select memory per data collection or deal with distributed machines. AutoMap deals with a decision space per task and data collection to find faster mappings.

Communication-Aware Process Mapping. Another class of prior work optimizes the mapping of MPI processes on a cluster to compute cores to minimize communication between MPI processes. Examples include search-based strategies and profile-guided strategies [9]. These works use information about communication between nodes to find optimal process placements based on how the

individual processes communicate. In these approaches, compute and data are unified and always placed together; thus the problem is simpler than the one we consider, where independent choices of which memory to place data in can also affect performance.

Autotuning Performance Optimizations. Multiple domain specific languages implement a similar separation between the high-level source program and a lower-level specification of an implementation in the space of possible performance optimizations (e.g. Halide for image processing [29], GraphIt for graph applications [42], TACO for sparse tensor algebra [20]). This separation provides an interface for automated optimization: OpenTuner [2], an extensible framework for program autotuning based on ensembles of search algorithms, has been used to find high performance schedules for Halide and GraphIt. The optimizations considered in these systems are higher-level data structure layout and parallelization transformations [24, 29, 36]. These systems address optimization problems different from mapping.

FlexFlow is a deep learning engine that automatically finds fast parallelization strategies for deep neural networks (DNNs) [19]. As with the DSLs above, FlexFlow's optimization problem is distinct from mapping: it searches over data and compute partitioning strategies for DNNs, with a fixed mapping strategy (execute all tasks on GPUs and store all data in Frame-Buffers). This search relies on a task graph cost estimator, which, like AutoMap, uses profiling to estimate execution times. Unlike AutoMap, it uses static bandwidth estimates, assumes a fixed mapping, and works with a DNN computation graph instead of the lower-level task graph.

Dynamic Load Balancing. Many approaches have been developed for dynamic load balancing [5, 7, 34]. Load balancing algorithms typically assume a much more uniform machine and tasks than we consider here, and there is no need to model task dependencies, memory constraints, or communication times.

Automated and Domain-Specific Mapping. Prior work in automated mapping uses static analysis to assign tasks to processors on heterogeneous machines [28] or to assign data to software-managed memory hierarchies [32]. Sbîrlea et al. combine compile-time analysis with dynamic work stealing to map a data-flow programming model onto heterogeneous platforms [35]. AutoMap is the first work that we know of which addresses the general problem of simultaneously mapping both tasks and data collections.

Other work has used domain-specific information to derive tuned mapping strategies for different applications in the domain: Lux is a distributed multi-GPU system for graph processing, which uses a hand-written mapper enhanced with dynamic load balancing [17]. ROC is a distributed multi-GPU system for fast graph neural network (GNN) training and inference, which implements dynamic graph partitioning [18]. The selected GNN partitioning strategy and memory management strategy imply a specific application mapping. In contrast, AutoMap does not make a domain-specific assumptions and targets the large class of iterative programs.

Profile-Guided Optimization. Profile-guided optimization uses profiling data collected at runtime to inform optimization decisions used in production runs [10]. AutoMap uses profiles of task executions and data movement costs. The inspector-executor framework uses dynamic analysis (the *inspector*) to capture information about a target program, and then runtime optimization (the *executor*) uses this information to optimize a program component [30, 31]. While we do not consider it in this paper, in principle AutoMap could be used in an inspector-executor style, where AutoMap is run on-line during an initial portion of a production run to select a fast mapping for the remainder of that execution.

7 CONCLUSION

We present AutoMap, a system that automatically maps tasks to processors and data to memories on parallel, heterogeneous, and distributed computer architectures. Our CCD search algorithm converges quickly to the fastest known mapping in all experiments. We show that fast mappings can be sensitive to many axes of variation, including application, machine count, and even input size, necessitating an automated approach to find these fast mappings. We show that AutoMap always equals or outperforms Legion's default mapping strategy and often outperforms hand-written mappings without any code modification or user intervention.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable comments that helped us improve this manuscript. We thank the Legion team for their feedback and support during the development of AutoMap. We thank Charlelie Laurent for providing access and support for the use of Maestro application. Alexandra Henzinger was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 2141064 and an EECS Great Educators Fellowship. Rohan Yadav was supported by an NSF Graduate Research Fellowship. This work is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003968 within the PSAAP III (INSIEME) Program at Stanford University.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. ACM, New York, NY, USA, 303–315.

[3] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. 2010. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*. IEEE, Shangai, China, 291–298. https://hal.inria.fr/inria-00523937

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[5] Stephen T Barnard and Horst D Simon. 1994. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and experience* 6, 2 (1994), 101–117.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 66, 11 pages.

[7] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering* 15, 6 (2013), 36–45.

[9] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. 2006. MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proceedings of the 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06)*. Association for Computing Machinery, New York, NY, USA, 353–360. https://doi.org/10.1145/1183401.1183451

[10] Thomas M Conte, Kishore N Menezes, and Mary Ann Hirsch. 1996. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer. In *Proceedings of the International Symposium on Microarchitecture*. IEEE/ACM, New York, NY, USA, 36–45.

[11] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling*. Dask. https://dask.org

[12] Mario Di Renzo, Lin Fu, and Javier Urzay. 2020. HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics. *Computer Physics Communications* 255 (2020), 107262. https://doi.org/10.1016/j.cpc.2020.107262

[13] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21 (2011), 173–193.

[14] Hillary R. Fairbanks, Lluís Jofre, Gianluca Geraci, Gianluca Iaccarino, and Alireza Doostan. 2020. Bi-fidelity approximation for uncertainty quantification and sensitivity analysis of irradiated particle-laden turbulence. *J. Comput. Phys.* 402 (2020), 108996. https://doi.org/10.1016/j.jcp.2019.108996

[15] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy, In SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* 0, 83–es.

[16] Charles Ferenbaugh. 2014. PENNANT: An unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* 27 (10 2014). https://doi.org/10.1002/cpe.3422

[17] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proceedings of the VLDB Endowment* 11, 3 (Nov. 2017), 297–310.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020*, Vol. 2. mlsys.org, Austin, TX, 187–198.

[19] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019*. mlsys.org, Stanford, CA, USA, March 31 - April 2, 2019, 1–10.

[20] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[21] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez Cid-Fuentes, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia. 2013. ServiceSs: An Interoperable Programming Framework for the Cloud. *Journal of Grid Computing* 12 (03 2013), 1–25.

[22] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. 1999. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *J. Parallel Distrib. Comput.* 59, 2 (Nov. 1999), 107–131. https://doi.org/10.1006/jpdc.1999.1581

[23] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 561–577.

[24] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[25] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. 2013. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494993

[26] OpenMP 2015. *OpenMP Version 4.5*. OpenMP. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 721, 12 pages.

[28] Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 50 (Oct. 2017), 28 pages.

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530.

[30] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. *SIGPLAN Not.* 50, 8 (jan 2015), 65–75. https://doi.org/10.1145/2858788.2688515

[31] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 72, 11 pages.

[32] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. 2008. A Tuning Framework for Software-Managed Memory Hierarchies. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 280–291. https://doi.org/10.1145/1454115.1454155

[33] Andrei Rădulescu and Arjan J. C. van Gemund. 1999. On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) *(ICS '99)*. Association for Computing Machinery, New York, NY, USA, 68–75. https://doi.org/10.1145/305138.305162

[34] Vijay A Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-based global load balancing. *ACM SIGPLAN Notices* 46, 8 (2011), 201–212.

[35] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. 2012. Mapping a Data-Flow Programming Model onto Heterogeneous Platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems* (Beijing, China) *(LCTES '12)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/2248418.2248428

[36] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 158 (2020), 30 pages.

[37] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel Computational Workflows in Python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

[38] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. https://doi.org/10.1109/71.993206

[39] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, NC, USA) *(PPoPP '09)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/1504176.1504189

[40] Rob Wijngaart and Tim Mattson. 2015. The Parallel Research Kernels. *2014 IEEE High Performance Extreme Computing Conference, HPEC 2014* 1, 1 (02 2015).

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, USA, 10.

[42] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 121 (2018), 30 pages.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8076466

ARTIFACT IDENTIFICATION

In a parallel and distributed application, a *mapping* is a selection of a processor for each computation or task and memories for the data collections that each task accesses. Finding high-performance mappings is challenging, particularly on heterogeneous hardware with multiple choices for processors and memories. We show that fast mappings are sensitive to the machine, application, and input. Porting to a new machine, modifying the application, or running on a substantially different input size may necessitate re-tuning the mapping to maintain the best possible performance.

We present AutoMap, a system that automatically tunes the mapping to the hardware used and finds fast mappings without user intervention or code modification. In contrast, hand-written mappings often require days of experimentation. AutoMap utilizes a novel *constrained coordinate-wise descent* search algorithm that balances the trade-off between running computations quickly and minimizing data movement. AutoMap discovers mappings up to 2.41× faster than custom, hand-written mappers.

We provide the source code of AutoMap as well as the code for the applications, scripts for job submission, and scripts for the generation of the charts. We also provide scripts to compile, build, and install AutoMap and the applications evaluated on the paper. The scripts for the job submission were used on two supercomputers: Shepard from Stanford University HPC Center and Lassen from Lawrence Livermore National Laboratory. Each node of Shepard has 2 Intel Xeon Platinum 8276 2.2 GHz CPUs with 28 cores each, 196 GB of RAM, and one NVIDIA Tesla P100 with 16 GB of memory. Each node of Lassen has 2 IBM Power9 3.45 GHz CPUs with 22 cores each (only 20 usable) and four NVIDIA Volta V100 GPUs with NVLink 2.0 (AC922 server) and 16 GB of memory.

We compiled all the codes with GCC 8.3.1 and CUDA 11.8. The artifact is accessible at https://gitlab.com/thiagotei/automap/-/releases/sc-2023 (RepoA). The applications' code and scripts for running the experiments and chart generation are at https://gitlab.com/thiagotei/automap-py-exps/-/releases/sc-2023 (RepoB). They are also accessible at https://doi.org/10.5281/zenodo.8076466.

Legion was built on commit 9cb37237 and was compiled using the following configuration:

```
export CC=gcc
export CXX=g++
export CONDUIT=ibv
export USE_CUDA=1
export USE_OPENMP=1
export USE_GASNET=1
export USE_HDF=1
export MAX_DIM=3
"$LEGION_DIR"/language/scripts/setup_env.py
```

This configuration builds Legion with optimizations, OpenMP, CUDA, HDF5, GASNet version 2023.3.0, and LLVM version 13.

REPRODUCIBILITY OF EXPERIMENTS

The experiment workflow consists of installing AutoMap and the applications, job submission of the AutoMap search for each application, job submission to execute the custom and default mappers, and the generation of the charts. The code for the Maestro application is still in the process of becoming open-source. The experiments used HTR version 1.4.1 mixture *ConstPropMix* from https://github.com/stanfordhpccenter/HTR-solver. All the other applications (Circuit, Pennant, and Stencil) are under *apps* in RepoB. During the search, each mapping ran 7 times, and the average was used to select the best mapping for each experiment. As a final step of the search, the applications were executed with each of the top 5 mappings 30 times, and the average of the fastest mapping reported in the charts.

The scripts to run the experiments shown in the paper for Shepard are under *shepard-results-numaware* in RepoB. The scripts *runAutomap{HTR, Pennant, Stencil, Circuit}.sh* execute the AutoMap search for each application. The scripts *runBaseline{HTR, Pennant, Stencil, Circuit}.sh* conduct the custom and default mappers experiments used for comparison. All these experiments take a few days to search for the best mapping for different inputs, applications, and the number of nodes. The script *genAllCharts.sh* can generate the figures. The script *runSearches.sh* conducts the experiments to compare search algorithms, and *RepoB/scripts/genChartSearchConverg.py* generates the respective figure. The repository has a folder with similar scripts under *lassen-results-numaware* for the experiments on Lassen. The script *simdata/runAllTaskGraph.sh* generates the two extra input files for the CCD search. The script *runAutomapMemCstrPennant.sh* run the experiments for the memory constraint section, and to generate the figure, we used *RepoB/scripts/genMemCstrChart.py*.

ARTIFACT DEPENDENCIES REQUIREMENTS

AutoMap is designed to optimize task-based applications running on heterogeneous systems containing multiple kinds of processors with access to multiple kinds of memories. The system is not limited to any particular processor or memory kind. It can optimize Legion applications developed using the C++ interface or the Regent language. AutoMap requirements and dependencies are:

- a Linux, macOS, or another Unix operating system;
- A C++ 11 (or newer) compiler (GCC, Clang, Intel, or PGI) and GNU Make;
- CUDA 7.0 or newer (for NVIDIA GPUs);
- Python 3.8 or newer (used by AutoMap driver);
- GASNet (for networking in distributed systems);
- LLVM 3.5 or newer (for applications developed in Regent).

AutoMap requires an input JSON file containing the application's arguments and the search space representation. The search space representation includes the processors and memory kinds of interest and all or a subset of tasks and data collections of the target application. The CCD search algorithm requires two extra files: one representing the data collection overlapping graph in

DOT format and another containing each task execution time and each task's data collection size in bytes in JSON format. These files are generated automatically by a script that runs and profiles the application once.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

The installation and deployments processes are as follows:

- Download and install Legion (estimated time: 30 minutes);
- Download and install AutoMap (estimated time: 15 minutes) and compile AutoMap's mapper (estimated time: 3 minutes);
- Compile the application to be optimized (the application must register AutoMap's mapper to be used by the Legion runtime);
- Invoke AutoMap to optimize the application through the search process.