Isometry: A Path-Based Distributed Data Transfer System

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Sean Treichler
NVIDIA
sean@nvidia.com

Galen Shipman
Los Alamos National Laboratory
gshipman@lanl.gov

Pat McCormick
Los Alamos National Laboratory
pat@lanl.gov

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Data transfers in parallel systems have a significant impact on the performance of applications. Most existing systems generally support only data transfers between memories with a direct hardware connection and have limited facilities for handling transformations to the data's layout in memory. As a result, to move data between memories that are not directly connected, higher levels of the software stack must explicitly divide a multi-hop transfer into a sequence of single-hop transfers and decide how and where to perform data layout conversions if needed. This approach results in inefficiencies, as the higher levels lack enough information to plan transfers as a whole, while the lower level that does the transfer sees only the individual single-hop requests.

We present Isometry, a path-based distributed data transfer system. The Isometry path planner selects an efficient path for a transfer and submits it to the Isometry runtime, which is optimized for managing and coordinating the direct data transfers. The Isometry runtime automatically pipelines sequential direct transfers within a path and can incorporate flexible scheduling policies, such as prioritizing one transfer over another. Our evaluation shows that Isometry can speed up data transfers by up to 2.2× and reduce the completion time of high priority transfers by up to 95% compared to the baseline Realm data transfer system. We evaluate Isometry on three benchmarks and show that Isometry reduces transfer time by up to 80% and overall completion time by up to 60%.

ACM Reference Format:
Zhihao Jia, Sean Treichler, Galen Shipman, Pat McCormick, and Alex Aiken. 2018. Isometry: A Path-Based Distributed Data Transfer System. In *ICS '18: International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205289.3205301

1 INTRODUCTION

Most existing parallel systems for high performance computing (e.g., MPI [4], Legion [7], ParSEC [8], Sequoia [11], Spark [32], and StarPU [6]) generally support only *direct* data transfers between two memories that have a physical hardware connection between them. Examples include data movement between a CPU memory and a GPU framebuffer on the same node or between CPU memories on different nodes. In most cases, these systems place the responsibility for managing *multi-hop* data transfers, which are not directly supported by hardware, on higher-level runtimes or the application.

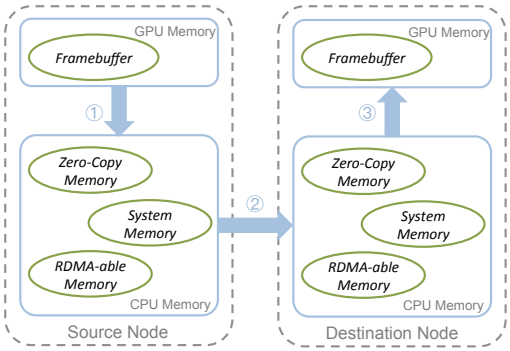


Figure 1: Data transfer between GPU framebuffers on different nodes.

For example, to move data between GPU memories on different nodes in Legion, as shown in Figure 1, the Legion runtime explicitly divides this transfer into three sequential direct transfers (labelled 1, 2, 3) and launches them separately to the underlying Realm low-level runtime [26]. In addition, applications typically use multiple data layouts to achieve optimal performance (e.g. neural networks [15], graph analytics [25], and simulations [10]). If the data movement requires layout conversions, the parallel runtime also decides which direct transfer performs a layout transformation.

We believe that managing only direct transfers is the wrong level of abstraction for today's parallel runtimes. We propose that the interface between the application or a high-level runtime and the low-level data movement engine, which we refer to as the *DMA engine* to keep with standard terminology, should be to move data from a source to a destination memory, with the DMA engine selecting and optimizing the *path* of the potential multi-hop transfer.

When data movement can be expressed only as a series of direct transfers to the DMA engine, two performance issues arise:

First, at the scheduling level, selecting an efficient path for a multi-hop transfer requires making a number of decisions, such as where to allocate *intermediate buffers* to hold temporary data, the size of each intermediate buffer, and the choice of how and where to perform data layout conversions if needed. These decisions should be made together, not separately, as they are not independent. The DMA engine is better positioned to make these decisions, as it has

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205301

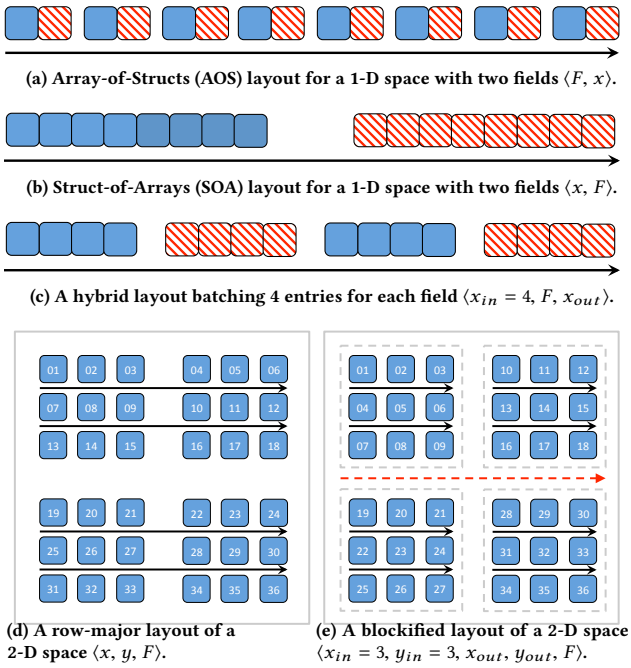
more knowledge about the state of system resources than the application; furthermore, putting the responsibility in the DMA engine avoids baking machine-specific decisions into the application.

Second, at the application level, pipelining sequential direct transfers within a path may be difficult to achieve when application developers are forced to explicitly implement path-specific synchronization and management. Moreover, to save system resources and allow concurrent transfers, the intermediate buffer sizes must generally be smaller than the entire data. This requires additional effort by application developers to properly synchronize and reuse the intermediate buffers to satisfy all dependencies. Finally, supporting scheduling policies such as prioritizing a transfer becomes more difficult, since this requires additional coordination both within a transfer and across transfers. All of these issues, however, can be automated and optimized by a DMA engine that has control over decision making for the entire transfer path.

We present Isometry, a path-based distributed data transfer system for managing and optimizing direct and multi-hop data transfers in distributed parallel systems. In Isometry, direct (one step) transfers between connected memories are represented by *transfer descriptors* (XDs). XDs are an intermediate level of abstraction. Below the XD layer, each XD is decomposed into a number of *requests* to move a portion of the data; requests are sent to a dedicated *channel*, which performs the actual data movement. Breaking direct transfers up into requests provides a natural way to support scheduling policies at the transfer level, such as prioritizing one transfer over another. Above the XD layer, Isometry includes a *path planner* that selects an efficient path for a multi-hop transfer. The path planner has a *full* planning algorithm that generates paths with optimal expected performance for large transfers and a *simple* planning algorithm that is faster and usually generates efficient paths for small transfers. Compared to simply choosing the shortest paths, both the full and simple path planning algorithms can generate paths that are much faster. The path planner also decides where to allocate the intermediate buffers, the size of each intermediate buffer, and where to perform layout conversions if needed. The path planner decomposes a multi-hop data transfer into multiple XDs and sends them to the runtime to perform the data movement.

We implement Isometry as an independent DMA engine and integrate it into Realm, the low-level runtime used by Legion. Compared to the original DMA engine in Realm, which is state-of-the-art and highly optimized for only direct transfers, Isometry achieves the same throughput for direct transfers and speeds up multi-hop transfers by up to 2.2×. Compared to MVAPICH2-GPU, a multi-hop DMA engine for GPU clusters, Isometry achieves up to 4× speedup for transfers to/from GPUs. We also evaluate Isometry on three benchmarks and show that Isometry reduces transfer time by up to 80% and overall completion time by up to 60%.

The rest of this paper is organized as follows. Section 2 presents the Isometry data model and interface. Section 3 presents the Isometry runtime. Section 4 describes the Isometry path planner. Section 5 describes the implementation of Isometry. We then evaluate Isometry in Section 6, survey related work in Section 7, and conclude in Section 8.



(a) Array-of-Structs (AOS) layout for a 1-D space with two fields $\langle F, x \rangle$.

(b) Struct-of-Arrays (SOA) layout for a 1-D space with two fields $\langle x, F \rangle$.

(c) A hybrid layout batching 4 entries for each field $\langle x_{in} = 4, F, x_{out} \rangle$.

(d) A row-major layout of a 2-D space $\langle x, y, F \rangle$.

(e) A blockified layout of a 2-D space $\langle x_{in} = 3, y_{in} = 3, x_{out}, y_{out}, F \rangle$.

Figure 2: Layout examples

2 DATA MODEL AND INTERFACE

The Isometry data model describes the organization of data, including in which memory the data is currently stored and the layout of the data in that memory. This allows Isometry to support a wide variety of layout conversions for data transfers (see Section 4).

An *instance* in Isometry names a collection of data in a specific memory. An instance is defined by an *index space* I and a *field space* F. The index space specifies the set of entries in an instance (e.g., a set of Cartesian grid points of arbitrary dimensions), while the field space describes the values stored for each entry in the index space. Each field $f \in F$ has a specific type T_f. A pair (i, f) where $i \in I$ and $f \in F$ uniquely identifies an entry of type T_f in the instance. For example, a matrix of complex numbers is represented as a two dimensional index space with two fields, one for the complex number's real component and one for its imaginary component.

An instance exists in a specific memory (e.g., in DRAM, in a GPU framebuffer, etc.) with a particular *layout*, which describes how entries in that instance are linearized in that memory. A *layout* is a vector describing the linearization order. The first element of the vector is the one that is stored densely; i.e., varies the fastest. The possible elements of a layout are:

- The name of the field space F;
- An index dimension x;
- A blocked index dimension and block size $x_{in} = c$;
- A blocked index dimension x_{out}

The field space F appears exactly once, and for each dimension x, either x is in the layout or $x_{in} = c$ and x_{out} are both in the layout.

Figure 2a shows an 1-D instance with two fields in array-of-structs (AOS) layout. This layout is described as $\langle F, x \rangle$, meaning we first iterate over the field space F and then over the single dimension

x. Iterating over the x dimension first (i.e. $\langle x, F \rangle$) results in a struct-of-arrays (SOA) layout, as shown in Figure 2b. Figure 2c shows a hybrid layout [17] that allows multiple entries for each field to be stored compactly for use with vectorized SSE or AVX loads and stores. The hybrid layout can be described as $\langle x_{in} = 4, F, x_{out} \rangle$, meaning that every 4 entries for each field are compactly stored.

The layout vector can also specify the order of entries within the index space for multi-dimensional instances. Figure 2d shows a row-major ordering for a 2-D index space. Isometry can also describe blockified layouts, as shown in Figure 2e, which are beneficial for applications that use the outer dimensions to define partitions with minimized boundaries. The blockified layout is $\langle x_{in} = 3, y_{in} = 3, x_{out}, y_{out}, F \rangle$. The $\langle x_{in} = 3, y_{in} = 3 \rangle$ indicates that every block containing 3×3 entries is compactly stored, with the x-dimension innermost. The $\langle x_{out}, y_{out} \rangle$ defines the linearization order among different blocks, with the x-dimension again varying fastest. Finally, F at the end means that the field space varies slowest. Entries for only one field are shown in Figures 2d and 2e.

Isometry provides the following API for launching a copy that transfers data from `src_mem` to `dst_mem` and transforms the data layout from `src_lyt` to `dst_lyt`. A multi-hop transfer is launched by a single invocation to `copy`.

```
Event copy(Memory src_mem, Layout src_lyt,
           Memory dst_mem, Layout dst_lyt,
           int priority = 0 /*optional*/);
```

Data transfers are asynchronous in Isometry. Every invocation to `copy` returns an `Event` that triggers when the transfer completes. An optional parameter `priority` can be provided to assign a transfer high priority. Isometry starts high priority transfers immediately and suspends all low priority transfers using the same channels.

3 ISOMETRY RUNTIME

Isometry is designed to be applicable to a variety of host runtimes. In general, any parallel system that meets the following requirements can use Isometry as its DMA engine:

- All data transfers are scheduled, managed and performed by an individual DMA engine.
- The DMA engine has the ability to allocate intermediate buffers in specific memories to perform potentially multi-hop data transfers. For example, the DMA engine can allocate buffers in CPU memory to transfer data between GPU framebuffers and disk storage.
- The data collections and layouts can be described by the Isometry data model in Section 2.

The Isometry runtime manages and optimizes direct data transfers, whether they have been directly requested by the application or as the result of the decomposition of a multi-hop request by the path planner (Section 4). The Isometry runtime can automatically coordinate and pipeline sequential direct data transfers within a multi-hop data transfer. This is accomplished through a hierarchical architecture, illustrated in Figure 3. This hierarchy is composed of memories, channels, requests, transfer descriptors, and DMA workers, which are described in detail below.

Table 1: Memories in the Isometry memory hierarchy.

Memory	Description
ZCM	Registered CPU memory with direct GPU access
REG	Registered CPU memory that is directly accessible by the network interface card (NIC)
SYS	Generic unregistered CPU memory
FBM	GPU device memory
DSK	Persistent storage on disk

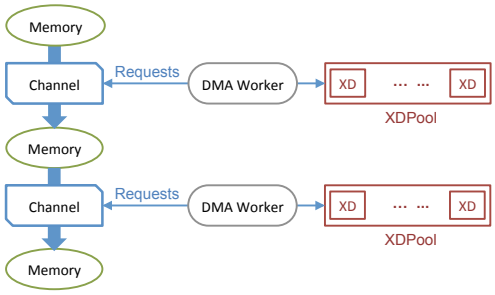


Figure 3: Isometry architecture.

Table 2: Channels in the Isometry memory hierarchy.

Channel	Source Memory	Destination Memory
Intra-node channels		
Memcpy	SYS/ZCM/REG	SYS/ZCM/REG
GPURead	FBM	SYS/ZCM/REG
GPUWrite	SYS/ZCM/REG	FBM
GPUCopy	FBM	FBM
DiskRead	DSK	SYS/ZCM/REG
DiskWrite	SYS/ZCM/REG	DSK
Inter-node channels		
RemoteMemcpy	SYS/ZCM/REG	SYS/ZCM/REG on remote nodes

3.1 Memories

Distinct storage locations (e.g., CPU memory, GPU framebuffers, and disk) have different performance characteristics (e.g., bandwidth, access latency, capacity, etc.) and are modeled as separate *memories* by Isometry. Moreover, today's heterogeneous systems typically have the ability to register a block of memory to facilitate direct access by the hardware DMA engines for optimizing data movement performance. Compared to normal system memories, a registered memory usually has better affinity with the hardware, such as higher bandwidth and lower access latency. Isometry captures this heterogeneity by further modeling each block of registered memory as a separate memory in the Isometry runtime. Examples of different types of memories are shown in Table 1.

3.2 Channels

A *channel* manages data movement between a memory pair with a hardware connection. Table 2 lists some channels in the Isometry runtime. A channel optimizes the hardware performance by coordinating invocations to the low-level data copy APIs. For example, each channel controls the size of each copy request and the number of concurrent requests to the low-level interface to accomplish optimized hardware throughput while maintaining low latency.

In addition to supporting bit-wise data copies, Isometry channels also exploit hardware capabilities by enabling data transposition

and directly offloading the transposition requests to the hardware DMA engines whenever possible. For example, the GPURead/GPUWrite channels support gathering and scattering data to and from GPU memories by using the `cudaMemcpy2D` API [1].

3.3 Requests

A *request* is a unit of direct data transfer that is generated from a transfer descriptor (see Section 3.4) and sent to the channel, which performs the actual data movement. Multiple requests to the same channel are performed by the channel in first-in-first-out order.

3.4 Transfer Descriptors

The Isometry runtime uses a *transfer descriptor* (XD) to represent a direct data transfer. A XD is associated with a particular channel and coordinates with other XDs to pipeline sequential direct transfers within a multi-hop transfer.

An XD decouples the entire direct transfer into multiple requests with smaller granularity and decides in which order the requests are sent to the channel. This design enables channel-specific optimizations. For example, XDs for the DiskRead and DiskWrite channels send requests that access the disk in sequential order to optimize disk performance. As another example, XDs for the GPURead, GPUWrite and GPUCopy channels opportunistically batch multiple bit-wise copies into a matrix or a 3D copy (i.e., `cudaMemcpy2D` or `cudaMemcpy3D`) and send a single request to the channels. This amortizes the overhead on the channel side for offloading requests to the hardware DMA engines.

An XD defines data movement from an *input buffer* to an *output buffer*. These buffers may be the source or destination buffer of the overall transfer, or they may be intermediate buffers managed by Isometry. When reading from an intermediate buffer, the XD cannot send requests to the channel until the previous XD writes the data to its input buffer. Such *copy dependencies* exist between adjacent XDs within a multi-hop data transfer. Each XD maintains which subsets are written by the previous XD and which subsets are read by the next XD and uses this information to generate requests that satisfy copy dependencies. (Note that an XD's incoming subsets of the data and its outgoing subsets may be different; e.g., the XD may perform layout conversions on its input data, it may batch several incoming requests into one outgoing request, etc.)

An XD becomes *idle* if additional requests cannot be generated because the previous XD hasn't written additional data into its input buffer, or the output buffer cannot be reused because the next XD hasn't read the data. Completion of requests from the previous or the next XD can potentially resolve copy dependencies and activate an idle XD, meaning the XD is able to generate additional requests.

3.5 DMA Workers

Each Isometry channel has a dedicated *DMA worker* that coordinates concurrent XDs. Each DMA worker manages a *transfer descriptor pool* (XDPool) that stores all unfinished XDs for the channel. A DMA worker iteratively selects requests from XDs in the XDPool using a certain policy and sends the requests to the channel.

Isometry allows DMA workers to implement channel-specific scheduling policies to meet different criteria for optimizing channel performance. By default, we use a priority mechanism for each DMA worker to allow prioritizing requests based on an application-specified priority for each transfer. Priorities can be especially beneficial for applications with critical tasks that may delay large amounts of subsequent work from being executed. By assigning a higher priority to the transfers for these tasks, the Isometry runtime automatically suspends all lower priority transfers and lets the high-priority transfers start immediately.

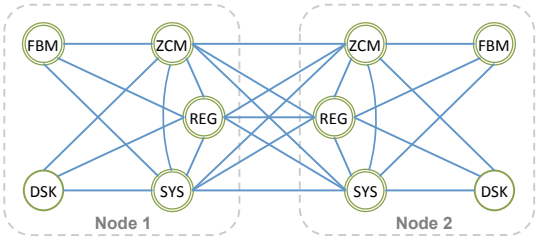


Figure 4: Transfer graph for two nodes. Each memory with double circles has a self-loop.

3.6 Intermediate Buffer Allocation

Isometry automatically allocates intermediate buffers for multi-hop data transfers. Since a multi-hop transfer may require allocating multiple intermediate buffers (see Figure 1), Isometry prevents potential resource allocation deadlock by imposing a global order on all Isometry memories. For each multi-hop transfer, Isometry allocates its intermediate buffers respecting the global order.

4 PATH PLANNER

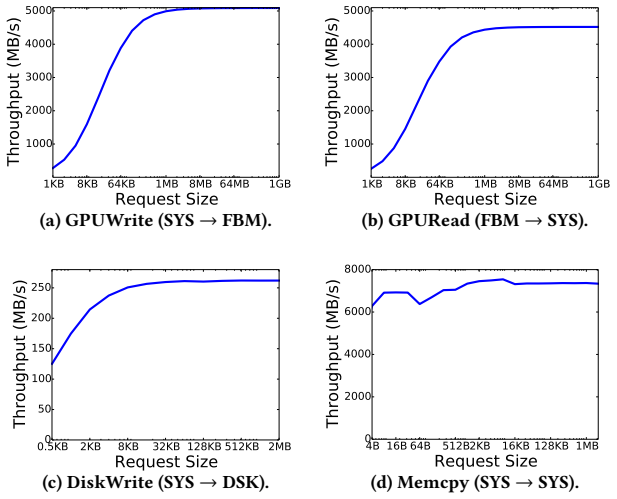
The increasingly deep and complex memory hierarchies in today's machines make selecting a path for multi-hop transfers non-trivial. The problem exists even in a cluster with two compute nodes. For example, to move data between GPU framebuffers on different nodes, as shown in Figure 1, it is unclear whether the transfer should use ZCM, REG, or SYS. Using ZCM improves the transfer performance from the GPU framebuffer but reduces the inter-node transfer performance. Conversely, using REG has the opposite effect. Using both is better in some cases, despite the additional copy required. In addition, for transfers with layout conversions, deciding how and where to convert the layouts adds another level of complexity.

A key idea in Isometry is to dynamically select efficient paths for data transfers based on the connectivity between memories, the bandwidth for each connection, and the layout of the instance being transferred. The data model introduced in Section 2 is used to reason about layout transformations.

4.1 Transfer Graphs

A data *transfer graph* is a directed graph $G = (V, E)$ that describes the physical connectivity between memories. Each node $m \in V$ is an individual memory, while each directed edge $\langle m_s, m_d \rangle \in E$ is a channel supporting data movement from m_s to m_d.

Figure 4 shows the transfer graph for two compute nodes connected by a network. Each node has a GPU and a local disk. We assume each node has CPU memory fragments that are registered with direct GPU/NIC access, as is common in HPC systems. For simplicity, if a pair of memories have channels in both directions



(a) GPUWrite (SYS → FBM). (b) GPURead (FBM → SYS). (c) DiskWrite (SYS → DSK). (d) Memcpy (SYS → SYS).

Figure 5: Throughput function examples.

that is depicted by an undirected edge. Note that these complementary channels may have different performance characteristics. The self-loops on FBM, ZCM, REG and SYS (the double circles in Figure 4) are useful—see the discussion of layout conversions below.

For each directed edge $\langle m_s, m_d \rangle$ in the transfer graph, there is a *throughput function* that describes the estimated throughput of the hardware connection transferring data from m_s to m_d. At startup, the Isometry runtime measures the throughput of each channel with respect to the request size and defines the edge's throughput function $tp(m_s, m_d, r)$ where r is the size of the request. Figure 5 illustrates some examples of throughput functions. Many channels (e.g., GPURead and GPUWrite channels) have suboptimal throughput for small requests, and some channels (e.g., Memcpy channels) perform consistently regardless of the request sizes. The latter channels play an important role for optimizing data transfers with layout conversions, since transforming layout may result in small requests being generated (see Section 4.2).

The other important property of a transfer graph is the *space threshold TH* for intermediate buffers: no intermediate buffer may be larger than *TH*. We currently use a global limit on buffer size because we have not found anything more complex to be necessary, though there is no difficulty in using a threshold per memory. Note that the size constraint is the limit for a single stage of a transfer, not a global constraint on the aggregate space used by all transfers.

4.2 Path Planning Algorithm

To initiate a transfer, the application provides (m_s, l_s) and (m_d, l_d), indicating the source memory and initial layout of the instance as well as its destination memory and the layout in the destination memory. The goal of the path planner is to generate a path

$$path = \langle (m_s, l_s), (m_1, l_1), (m_2, l_2), ..., (m_n, l_n), (m_d, l_d) \rangle$$

where *path* describes a transfer route that moves data from memory m_s to memory m_d and transforms the instance's layout from l_s to l_d. Each internal pair $(m_i, l_i) \in path$ $(1 \le i \le n)$ indicates an *intermediate buffer* in memory m_i that is used to temporarily hold part of the instance during the transfer. The partial instance in the intermediate buffer m_i is linearized with layout l_i.

We define $(m_0, l_0) = (m_s, l_s)$ and $(m_{n+1}, l_{n+1}) = (m_d, l_d)$. Each pair $(m_i, l_i), (m_{i+1}, l_{i+1}) \in path$ $(0 \le i \le n)$ describes a direct transfer and becomes an XD that moves the instance from memory m_i to m_{i+1} and transforms the instance from layout l_i to l_{i+1}.

Recall that layouts are vectors as illustrated in Figure 2. To analyze how to transfer data from layout l_i to l_{i+1}, we consider their *longest common prefix* and *longest common suffix*. Intuitively, a long common prefix indicates that two layouts have the same local linearization order, which allows large requests sent to the channel. On the other hand, a long common suffix means the two layouts have similar high-level linearization order, which reduces the minimum size of the intermediate buffers needed for holding temporary data. We name the longest common prefix and suffix as follows:

$$\begin{array}{r|c|l} l_i = \langle l(0), l(1), ..., l(pf), & ..., & l(sf), ..., l(L) \rangle \\ l_{i+1} = \langle l(0), l(1), ..., l(pf), & ..., & l(sf), ..., l(L) \rangle \end{array}$$

$l(0)...l(pf)$ and $l(sf)...l(L)$ are the longest common prefix and suffix between layouts l_i and l_{i+1}, respectively.

First we define the size of each layout component $|l(k)|$:

$$|F| = \text{ the sum of the sizes of the fields in the field space } F$$
$$|x| = \text{ the size of the } \times \text{ dimension}$$
$$|x_{in} = C| = C$$
$$|x_{out}| = |x|/|x_{in}|$$

We then define two functions $\text{PREFIX}(l_i, l_{i+1})$ and $\text{SUFFIX}(l_i, l_{i+1})$ that are the products of the sizes of each layout component in the longest common prefix and suffix, respectively, for layout l_i and l_{i+1}. We use $\text{TOTAL}(l_i)$ to count the total size of an instance with layout l_i [1], and define $\overline{\text{SUFFIX}}(l_i, l_{i+1}) = \text{TOTAL}(l_i)/\text{SUFFIX}(l_i, l_{i+1})$.

$$\text{TOTAL}(l_i) = \prod_{i=0}^{L} |l(i)|, \qquad \text{PREFIX}(l_i, l_{i+1}) = \prod_{i=0}^{pf} |l(i)|$$
$$\text{SUFFIX}(l_i, l_{i+1}) = \prod_{i=sf}^{L} |l(i)|, \qquad \overline{\text{SUFFIX}}(l_i, l_{i+1}) = \prod_{i=0}^{sf-1} |l(i)|$$

To transfer an instance from layout l_i to l_{i+1}, every $\text{PREFIX}(l_i, l_{i+1})$ entries can be merged into a single request and sent to the channel, since these entries are in the same linearization order in l_i and l_{i+1}. Sending large requests is a critical optimization for some channels.

To analyze the sizes of the intermediate buffers, Isometry focuses on XDs whose input and output buffers are both intermediate buffers, since other XDs always have at least one endpoint that will hold the entire instance. For XDs that transfer between intermediate buffers, $\overline{\text{SUFFIX}}(l_i, l_{i+1})$ measures the data shuffle size and provides a lower bound for the input and output buffer sizes.

The path planner automatically generates the following constraints for the *overall throughput* $\text{TP}_{overall}$ of *path*.

- *Connectivity constraints* require that adjacent memories in *path* must be connected by a channel (i.e., $\langle m_i, m_{i+1} \rangle \in E$).
- For adjacent pairs $(m_i, l_i), (m_{i+1}, l_{i+1}) \in path$, $\text{PREFIX}(l_i, l_{i+1})$ provides an upper bound for the request size. A *throughput*

[1] Note that we always have $\text{TOTAL}(l_i) = \text{TOTAL}(l_{i+1})$ since no data is removed or added during a transfer.

constraint requires that the overall throughput cannot exceed the throughput of this direct transfer: $\text{TP}_{overall} \leq tp(m_i, m_{i+1}, \text{PREFIX}(l_i, l_{i+1}))$.
- *Space constraints* require that an XD cannot allocate an intermediate buffer whose size is larger than a threshold *TH*. For XDs whose input and output buffers are both intermediate buffers, $\overline{\text{SUFFIX}}(l_i, l_{i+1})$ provides a lower bound for the intermediate buffer sizes on both sides. Therefore, the space constraints require that $\overline{\text{SUFFIX}}(l_i, l_{i+1}) \leq TH$ for $0 < i < n$.

With these definitions in hand, we can now describe the algorithm used by the path planner to select a path with maximum throughput. There are three components to the algorithm:

(1) The planner enumerates the set P of all possible paths between the source and destination memories. This step satisfies the connectivity constraints.
(2) For a given path $p \in P$ and a desired throughput TP, the planner solves the throughput and space constraints.
(3) For each path $p \in P$, the planner finds the maximum throughput by solving an optimization problem that uses part (2) above as a subroutine. The final answer is the path with the greatest maximum throughput over all paths.

For part (1), the planner considers all acyclic paths from the source to the destination memory, excepting that self-loops may be used at most once. Furthermore, note that the set of possible layouts is also finite, as each dimension can appear either once or twice (e.g., as x_{in} and x_{out}) in a layout. The total number of possible paths is therefore finite, but may be quite large. The following techniques are used to further reduce the number of paths that are considered:

- Paths that convert the layout of the innermost dimension are ignored for channels that behave poorly with small requests.
- Rather than enumerate all possible blocking factors for a layout component x_{in}, Isometry solves for the optimal blocking factor as part of component (3) above, treating x_{in} as another variable in the optimization problem.
- We compute an initial conservative upper bound U on the maximum throughput of all paths. If a path p is found that approaches U, p is returned as the solution and no further paths are considered. U is computed via a breadth-first search from the source to the destination considering only the peak bandwidth of each channel.

For part (2), given a path p and a potential throughput $\text{TP}_{potential}$, we determine if there is a solution to the throughput constraints where $\text{TP}_{overall} \geq \text{TP}_{potential}$. For each throughput constraint

$$\text{TP}_{overall} \leq tp(m_i, m_{i+1}, \text{PREFIX}(l_i, l_{i+1}))$$

We first find r_i' such that

$$tp(m_i, m_{i+1}, r_i') = \text{TP}_{potential}$$

We then replace the throughput constraint by

$$r_i' \leq \text{PREFIX}(l_i, l_{i+1})$$

This requires the reasonable assumption that the throughput function is monotonic[2]. Now we have a system of constraints

$$\begin{array}{lcll} \text{PREFIX}(l_i, l_{i+1}) & \geq & r_i' & \forall 0 \leq i \leq n \\ \overline{\text{SUFFIX}}(l_i, l_{i+1}) & \leq & TH & \forall 0 < i < n \end{array}$$

where for each constraint one side is a constant and the other is a product of variables and a coefficient (recall from above that we solve for the blocking factors). Thus, by taking the logarithm of both sides, this non-linear system can be transformed to a linear system, which is easily checked for satisfiability.

For part (3), the planner simply performs a binary search on possible throughputs for path p in the interval between 0 and the conservative upper bound U and records the maximum throughput found. This process is repeated for all paths and the path with the optimal throughput is selected.

The path planner involves a caching mechanism to eliminate the cost of planning for new transfers that closely match earlier transfers. After generating a path, the path planner caches the source/destination memory and layout, as well as the path selected. Any future transfers between the same memories with the same layouts will use the cached path, without rerunning the algorithm. The caching mechanism is very effective for iterative applications. Section 6.5 evaluates Isometry on three benchmarks and shows that while the path planner generates many new paths early in our benchmarks, it quickly converges to a state where very few or no new paths are generated.

4.3 An Example

We use a data transfer between FBMs on different nodes to demonstrate how the planning algorithm works. The transfer performs matrix transposition by moving data from a row-major 2-D instance to a column-major 2-D instance in the two-node cluster in Figure 4. For simplicity, we assume all fields for an entry are compactly stored (i.e., F is in the inner-most dimension in the layout).

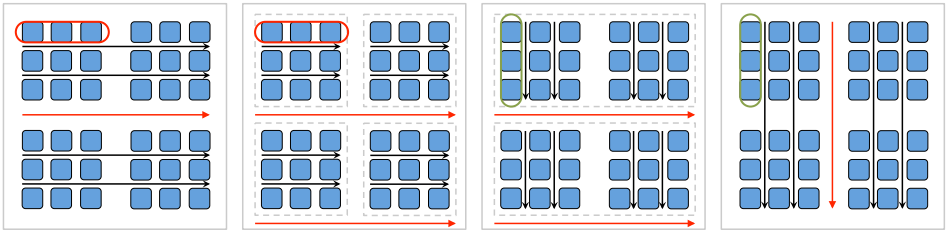
Table 3 shows the analysis of one potential path. We split every dimension into an inner and an outer dimension to emphasize the layout conversions. Figure 6 visualizes the linearization orders for the layouts in Table 3. Each blue box represents all fields for an entry in the 2-D instance. The path analysis proceeds as follows:

- $(m_0, l_0) \rightarrow (m_1, l_1)$: The data is moved from the initial FBM to the ZCM with direct GPU access. The longest common prefix between l_0 and l_1 is $\langle F, x_{in} \rangle$, which indicates that this direct transfer can merge $|F| \times |x_{in}|$ entries into a single request, shown as the red boxes in Figures 6a and 6b. A large x_{in} helps optimize the throughput for the GPURead channel, as shown in Figure 5b. Isometry omits the space constraint for this data transfer since m_0 is not an intermediate buffer.
- $(m_1, l_1) \rightarrow (m_2, l_2)$: The data is then transferred to REG. For the layout conversion, only the fields F for a single entry can be batched into a request. However, MemcpyChannel provides good performance with small requests, as shown in Figure 5d. Since both the input and output buffers for this direct transfer are intermediate buffers, the space constraint applies $|F| \times |x_{in}| \times |y_{in}| \leq TH$.

[2]When a throughput function is not monotonic, it can be made so by automatically splitting slow large requests into faster smaller requests.

Table 3: Path analysis on a path between GPU framebuffers on different nodes. $*$ indicates memories on the destination node.

Memory	Layout	PREFIX(l_i, l_{i+1})	$\overline{\text{SUFFIX}}(l_i, l_{i+1})$	Connectivity Constraints	Throughput Constraints	Space Constraints
m_0 = FBM	$l_0 = \langle F, x_{in}, x_{out}, y_{in}, y_{out} \rangle$	$\|F\| \times \|x_{in}\|$	TOTAL$(l)/\|y_{out}\|$	✓	$\text{TP}_{overall} \leq tp(m_0, m_1, \|F\| \times \|x_{in}\|)$	
m_1 = ZCM	$l_1 = \langle F, x_{in}, y_{in}, x_{out}, y_{out} \rangle$	$\|F\|$	$\|F\| \times \|x_{in}\| \times \|y_{in}\|$	✓	$\text{TP}_{overall} \leq tp(m_1, m_2, \|F\|)$	$32MB \geq \|F\| \times \|x_{in}\| \times \|y_{in}\|$
m_2 = REG	$l_2 = \langle F, y_{in}, x_{in}, x_{out}, y_{out} \rangle$	TOTAL(l_2)	1	✓	$\text{TP}_{overall} \leq tp(m_2, m_3, \text{TOTAL}(l_2))$	$32MB \geq 1$
m_3 = REG$*$	$l_3 = \langle F, y_{in}, x_{in}, x_{out}, y_{out} \rangle$	$\|F\| \times \|y_{in}\|$	TOTAL(l_3)	✓	$\text{TP}_{overall} \leq tp(m_3, m_4, \|F\| \times \|y_{in}\|)$	
m_4 = FBM$*$	$l_4 = \langle F, y_{in}, y_{out}, x_{in}, x_{out} \rangle$					



(a) l_0 (b) l_1 (c) l_2 and l_3 (d) l_4

Figure 6: Linearization orders of the layouts in Table 3.

- $(m_2, l_2) \rightarrow (m_3, l_3)$: The data is moved through the network. Because $l_2 = l_3$ it automatically satisfies the space constraint.
- $(m_3, l_3) \rightarrow (m_4, l_4)$: The data is moved to the destination memory. Similar to the first direct transfer, the XD can merge $|F| \times |y_{in}|$ entries into a single request, marked as the green boxes in Figure 6c and 6d. Again, the space constraint is eliminated because m_4 is not an intermediate buffer.

If we use the throughput functions in Figure 5 and assume $|F|$ = 16, TH =32MB, and a 3GB/s network bandwidth (the environment used in the evaluation), the planning algorithm finds that $\text{TP}_{overall}$ is 3GB/s with the blocking factors x_{in} = 2K and y_{in} = 1K.

4.4 A Simple Path Planning Algorithm

The path planning algorithm described in Section 4.2 generates efficient paths for data transfers. However, for small data transfers, the planning algorithm takes a relatively long time compared to the actual data transfer time. For example, the planning algorithm takes up to 70μs for 1-D transfers and 0.2ms for 2-D transfers in the experiments. Therefore, we also consider an alternative *simple* planning algorithm that is faster and usually generates efficient paths. The simple path planning algorithm selects the shortest path between the source and destination memories, provided the shortest path includes a Memcpy channel for layout conversions (if layout conversion is required).

In the absence of layout conversions, the shortest path is returned as the data transfer path. However, for data transfers involving layout conversions, the shortest path may not include a channel that is efficient for layout transformations. For the example in Section 4.3, the shortest path is FBM $\rightarrow$ REG $\rightarrow$ REG$*$ $\rightarrow$ FBM$*$, in which none of the GPURead, RemoteMemcpy, or GPUWrite channels have good performance to transform layouts. For transfers with layout conversions, the simple planning algorithm restricts the search space by requiring all intermediate buffers to use the same layouts as the source or destination, and only one channel on the path converts the layouts. In addition, the simple planning algorithm always uses the Memcpy channels for layout conversions. If the shortest path does not include a Memcpy channel, one is added after the first CPU memory on the path to improve layout conversion performance. For the transfer example in Section 4.3, the simple planning algorithm generates the path FBM $\rightarrow$ REG $\rightarrow$ REG $\rightarrow$ REG$*$ $\rightarrow$ FBM$*$ and uses the Memcpy channel REG $\rightarrow$ REG to transform the layout.

The simple algorithm runs faster because it reduces the set of considered paths and the possible layouts on each path. However, it may generate suboptimal paths both with and without layout conversions. Section 6.3 performs a comparison between the two path algorithms and shows that the simple path algorithm performs reasonably well for most data transfers but generates suboptimal paths in certain circumstances.

For large data transfers, generating efficient paths is critical to data transfer performance, and Isometry uses the full planning algorithm to find the best possible paths. For small data transfers, reducing planning time is important to avoid the possibility that planning time could dominate transfer time. Note that while caching paths is another way to reduce (amortized) path planning time, caching would still be less effective than reducing planning cost for programs that either run for a short period of time or have many small transfers along unique paths. We use a transfer size threshold to decide where to switch between the two algorithms. The threshold is set to 16MB in the experiments because transferring 16MB data takes around 2-5ms for all non-disk channels, which is 10 times longer than worst-case full path planning time.

Table 4: System configurations used for the experiments.

Cluster	Sapling	XStream	Moonlight
Nodes	4	16	308
CPUs/Node	2x Xeon 5680	2x E5-2680	2x E5-2670
GPUs/Node	2x Tesla C2070	2x Tesla K80	2x Tesla M2090
DRAM/Node	48 GB	256GB	32GB
Disk	Samsung SSD	1.4PB SAS hard drive	1.8PB Panasas [29]
Network	2x QDR Infiniband	FDR Infiniband	QDR Infiniband
Experiments	Sections 6.1 to 6.4	Section 6.5	Section 6.5

5 IMPLEMENTATION

Isometry is applicable to a variety of data parallel systems that meet the requirements listed in Section 3. We now describe an implementation of Isometry within Legion [7]. Legion is a parallel programming system for distributed heterogeneous architectures. Legion satisfies the three requirements: (1) Realm (Legion's low-level runtime component [26]) has a dedicated DMA subsystem; (2) the DMA subsystem can allocate buffers in specific memories; and (3) Legion's *physical instances* (the basic building block for Legion's data collections) can be expressed by Isometry's data model.

5.1 Realm DMA Subsystem

The Realm DMA subsystem supports moving data between memories with a direct hardware connection. For each transfer launched by the application, the DMA subsystem analyzes the source and the destination memory and generates a specialized *copier* that describes how to perform and optimize the transfer. All copiers are enqueued into a *priority queue* for execution.

The DMA subsystem initially launches a fixed number of *DMA threads* for parallelizing data transfers. Each DMA thread iteratively selects a copier with the highest priority from the priority queue and performs the corresponding data transfers.

Copiers in the Realm DMA subsystem include optimizations for each specific channel; however, coordination among concurrent copiers is missing. Multiple DMA threads competing for the same channel may slow down all transfers. Furthermore, some channels may stay idle if their transfers are low priority and are not selected by any DMA thread.

5.2 Integration into Realm

Our implementation of Isometry includes all memories and channels listed in Table 1 and 2 as well as all optimizations described in this paper. In our implementation, Isometry replaces the original DMA subsystem in Realm. No changes are required at the application level. With Isometry integrated into Realm, the Legion high-level runtime can launch transfers between any two memories, with the Isometry path planner automatically selecting efficient paths and the Isometry runtime optimizing data transfers.

6 EVALUATION

We first study the performance of the Isometry runtime, focusing on how effectively Isometry pipelines direct data movement within a data transfer (Section 6.1) and the overhead for prioritizing a data transfer (Section 6.2). Second, we evaluate the two planning algorithms (Section 6.3). Finally, we compare Isometry with a state-of-the-art multi-hop DMA engine (Section 6.4) and evaluate Isometry on three benchmark applications (Section 6.5). All experiments were run on three clusters (see Table 4).

Table 5: Channel performance. Numbers in bold indicate the best bandwidth for a channel type.

Source Memory	Destination Memory	Channel Type	Bandwidth (MB/s)
Intra-node channels			
ZCM/REG/SYS	ZCM/REG/SYS	Memcpy	**7740**
ZCM	FBM	GPUWrite	**5941**
REG/SYS	FBM	GPUWrite	5139
FBM	ZCM	GPURead	**6418**
FBM	REG/SYS	GPURead	4502
FBM	FBM	GPUCopy	**55366**
ZCM/REG/SYS	DSK	DiskWrite	**270**
DSK	ZCM/REG/SYS	DiskRead	**280**
Inter-node channels			
REG	ZCM/REG/SYS	RemoteMemcpy	**3180**
ZCM/SYS	ZCM/REG/SYS	RemoteMemcpy	2701

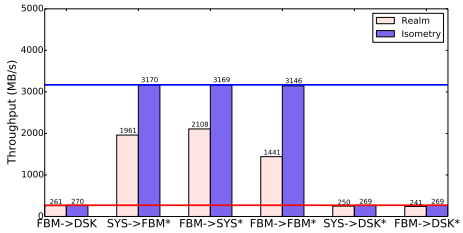
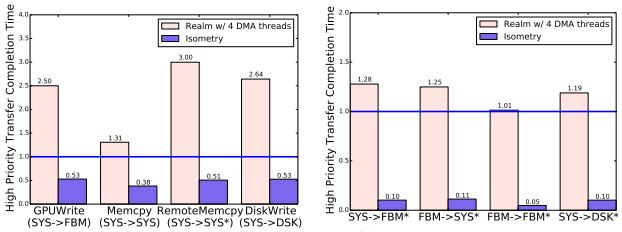


Figure 7: Multi-hop data transfer throughput (higher is better). ∗ indicates that the destination memory is on a remote node. The blue and the red horizontal lines show the network and the disk bandwidth, respectively.

Table 5 reports the bandwidth numbers for all Isometry channels in the Sapling cluster. Note that different instances of the same channel type can have different maximum bandwidths. For example, the bandwidth of the GPURead channel is 4502 MB/s if the destination memory is normal system memory and 6418 MB/s if the destination memory is registered with GPU access. This bandwidth heterogeneity greatly increases the number of potential paths for some data transfers. We have compared the channel bandwidth measured by Isometry with those measured by Realm [26]. Realm and Isometry achieve the same bandwidth for all channels listed in Table 5.

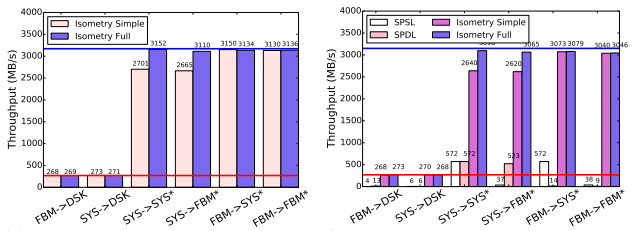
6.1 Pipelining Direct Transfers

To evaluate how effectively Isometry pipelines direct data transfers, we use a microbenchmark that moves 8GB of data by concurrently transferring 64 instances from the source memory to the destination memory. Each instance has a 1-D index space with 4 million entries and 8 fields. Figure 7 shows the throughput for various data transfer scenarios. Because the original Realm runtime only handles single-hop direct transfers, the Legion high-level runtime decomposes any multi-hop transfer into a series of single-hop transfers with no pipelining. As a result, Realm has suboptimal performance for some data transfers. For example, to transfer data between FBMs on different nodes, the Legion runtime creates two new instances



(a) Direct data transfers. (b) Multi-hop data transfers.

Figure 8: Performance of prioritizing a transfer. The y-axis is normalized (lower is better), with 1 indicating the run time of a high priority transfer in Realm w/ 1 DMA thread.



(a) Transfers w/o layout conversions. (b) Transfers w/ layout conversions.

Figure 9: Performance with different path algorithms.

in the CPU memories on the two nodes. The new instances are the same size as the initial instance. Three sequential data transfers are launched to Realm, with the start of each transfer dependent on the completion of the previous data transfer before it can start. This disables the opportunities to potentially overlap data transfers in the GPU channels and the network channels; in addition, the large allocations for holding the temporary instances waste system resources and may prevent concurrent transfers.

With Isometry, the Legion runtime simply specifies the source and destination of the transfer. The path planner in Isometry automatically selects a path from the source to the destination and decomposes the path into XDs that are issued to the Isometry runtime for parallel execution. For all transfer scenarios in Figure 7, Isometry is able to pipeline sequential transfers and achieves near-optimal throughput.

6.2 Priority Scheduling

We evaluate the completion time of a high priority data transfer while a number of low priority transfers are being performed concurrently. We measure the time from when the high priority transfer is launched to when the runtime acknowledges the completion.

The original Realm DMA subsystem supports prioritizing a transfer as described in Section 5.1; however, a high priority transfer cannot be performed until a DMA thread has completed its current transfer (regardless of the priority) and selects the high priority transfer from the priority queue. This delays the start time of a high priority transfer. Moreover, Realm does not support coordination among different DMA threads; as a result, multiple DMA threads share the hardware resources if they are performing the same type of transfer, regardless of the priorities of the transfers, which can further prolong the completion time of a high priority transfer.

Isometry allows priority scheduling for each channel. Since each XD results in a series of requests with small granularity, the channel quickly finishes the current requests and lets the DMA worker select new requests from its XDPool. As a result, when a high-priority XD is enqueued, the DMA worker quickly stops selecting requests from low-priority XDs and allows the high-priority XD to start immediately and monopolize the channel bandwidth.

Figure 8a and 8b show the completion time for a high priority transfer with the presence of low priority transfers in the Realm DMA subsystem and Isometry. Compared to the performance of Realm with 1 DMA thread (the blue line), starting multiple DMA threads in Realm prolongs the completion time for high priority transfers because multiple DMA threads share the channel bandwidth. Isometry is able to reduce the completion time for direct and multi-hop high priority transfers by up to 62% and 95%, respectively.

6.3 Path Planning Algorithms

We evaluate the two path planning algorithms by comparing the transfer throughput of the generated paths, both with and without layout conversions. We use the terms full and simple to refer to the algorithms described in Section 4.2 and Section 4.4, respectively.

For each transfer scenario, the microbenchmark transfers 8GB of data by concurrently moving 64 instances from the source to the destination memory. Each instance has an index space with 4M entries and a field space with 8 integers. To measure the performance with layout conversions, the instance has the AOS layout in the source memory and the SOA layout in the destination memory.

Figure 9a shows the performance of the full and simple algorithms in the absence of layout conversions. Both algorithms select the same paths for some transfers (e.g., FBM→DSK); however, the simple algorithm may generate suboptimal paths in certain circumstances. For example, to transfer data between the system memories on different nodes, the simple algorithm chooses the shortest path that directly links the source and the destination memories (shown as the red path in Figure 10). The full algorithm selects the blue path in Figure 10 that uses the REG on the source node to exploit direct access by NICs and achieves a 17% speedup.

For transfers with layout conversions, we also use two intuitive algorithms *shortest-path-source-layout* (SPSL) and *shortest-path-destination-layout* (SPDL) as baselines. Both algorithms use the shortest path algorithm to select a path. To select layouts for intermediate buffer, SPSL requires all intermediate buffers to use the same layout as the source instance (i.e., the layout transformation is done in the last hop), while SPDL requires all intermediate buffers to use the destination layout (i.e., the first hop transforms the layout). Both algorithms are reasonable for maintaining small intermediate buffers, since all but one hop perform no layout conversion.

In our experiments, the SPSL, SPDL, and simple algorithms take 2-9μs to generate a path that transforms the layout from AOS to SOA, while the full algorithm takes up to 100μs.

Figure 9b shows the performance with layout conversions in different cases. The paths generated by SPSL and SPDL may have very different throughput, since the layout conversions are performed on different channels. Note that for some common data transfers (e.g., moving data from system memory to disk), SPSL and SPDL have poor throughput because both of them transform the layouts in the DiskWrite channel. Both the simple and full algorithms generate

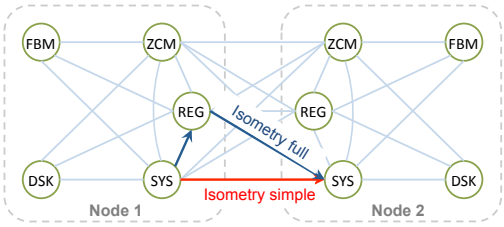
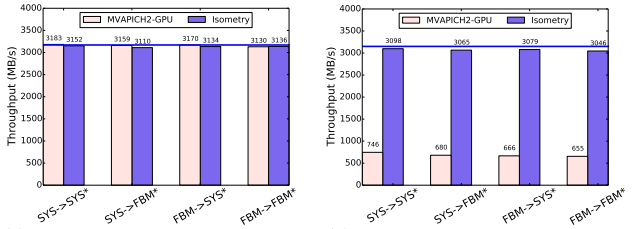


Figure 10: Different paths for data movement between system memories on different nodes.



(a) Transfers w/o layout conversions. **(b) Transfers w/ layout conversions.**

Figure 11: Comparison with MVAPICH2-GPU.

paths that use an in-memory intermediate buffer to transpose the layouts and submit large requests to the DiskWrite channel. This is a common, but tedious, optimization performed manually in many applications, and one that happens automatically in Isometry with both path planning algorithms.

In summary, both the simple and full planning algorithms are better than more straightforward baseline path selection algorithms. The simple algorithm provides relatively efficient paths by selecting (possibly modified) shortest paths and restricting layout conversions to Memcpy channels. However, the simple algorithm can also generate suboptimal paths. Therefore, for large transfers, using the full algorithm to generate more efficient paths is worthwhile, especially given that the results are cached for use in future transfers.

6.4 Comparison with Multi-hop DMA Engine

We compare the performance of Isometry with MVAPICH2-GPU [27], a state-of-the-art MPI implementation optimized for data transfers from/to GPU framebuffers across multiple nodes. We use the same benchmarks as Section 6.3 for this experiment. To obtain the best performance for MVAPICH2-GPU, we explicitly aggregated strided copies into a single `MPI_ISend` call to minimize runtime overhead.

Figure 11 shows the comparison results. For transfers without layout conversions, MVAPICH2-GPU and Isometry achieve the same transfer throughput (approaching network bandwidth). For transfers with layout conversions, Isometry achieves around 4× speedup compared to MVAPICH2-GPU. Note that Isometry supports transfers between all memory pairs, not just GPU framebuffers.

6.5 Benchmark Evaluation

We evaluate Isometry on three benchmarks. PageRank and Circuit are distributed implementations of PageRank [13, 16] and an electrical circuit simulation. VGG-16 performs distributed training of a deep convolutional neural network using data parallelism [23].

Table 6: Benchmark statistics.

Benchmark	Circuit	PageRank	VGG-16
Data transfer pattern	varying sizes	large transfers	varying sizes
Layout conversions	No	No	Yes
Full path generated	96 (master) / 5 (slaves)	16 (per node)	16 (per node)
Simple path generated	50 (master) / 2 (slaves)	0	4 (per node)
Total planning time (ms)	5.5 (master) / 0.3 (slaves)	0.8 (per node)	0.9 (per node)
	(The above numbers are for executions on 16 nodes)		

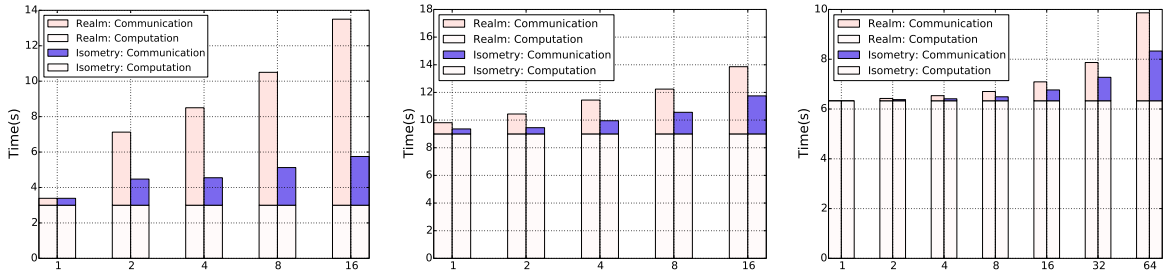
Table 6 summaries the data transfer patterns. The three benchmarks are written in Legion, and we use Isometry to replace the original DMA subsystem in Realm. All benchmarks involve both direct transfers (e.g., ZCM → local FBM) and multi-hop transfers (e.g., FBM → remote FBM). To achieve optimal performance, the VGG-16 benchmark uses *NHWC* layout for CPU tasks and *NCHW* layout for GPU tasks as suggested in [5]. As a result, it requires layout conversions for data transfers between SYS and local FBM.

We ran PageRank and VGG-16 on the XStream cluster and Circuit on the Moonlight cluster. The configurations for XStream and Moonlight are listed in Table 4. We used all compute nodes for the experiments on XStream and limited the runs to 64 nodes on Moonlight to get sufficient cluster time. In the experiments, the full and simple planning algorithms take 12-102μs and 2-9μs to generate a new path, respectively. The planning time for a previously cached path is negligible. Table 6 shows the number of full and simple paths that are generated by the path planner on each node due to cache misses. (Circuit has a master node that communicates with all other nodes. Therefore, the master node performs more transfers.) In all benchmarks, no new path is generated after the first iteration because of the cache mechanism, and so overall planning time is an insignificant fraction of wall clock execution time.

Figure 12 shows the weak-scaling performance of the three benchmarks. The computation time is the average elapsed time to compute a single partition of the application on one node, where the problem size of each partition is held constant (i.e., weak scaling). We compute the communication overhead by subtracting the computation time from the benchmark's overall execution time. For PageRank and VGG-16, Isometry reduces the communication overhead by 40-80% and the overall execution time by up to 60%. It is worth noting that for the VGG-16 execution on a single node, which includes only direct transfers, the layout conversion optimization reduces the communication overhead by 44%.

For Circuit, communication takes a relatively small proportion of overall execution time. However, communication overhead increases as we scale the number of nodes and reaches 36% on 64 nodes. For executions on 64 nodes, Isometry reduces the communication time by 43% and the overall execution time by 16%.

We found that two main optimizations in Isometry achieve most of the performance benefit over Realm: Isometry's path planner and Isometry's automatic pipelining of the direct transfers in a multi-hop transfer. Because Realm only supports direct transfers, Legion decomposes any multi-hop transfer into a series of direct transfers, which disables any pipelining opportunities. Moreover, Realm needs to allocate large intermediate buffers to hold the entire instance for a multi-hop transfer, which may preclude using smaller registered memories with better transfer performance. Figure 13 shows the transfer performance degradation for PageRank by cumulatively disabling the path and pipeline optimizations. To disable



(a) PageRank (10 iterations). (b) VGG-16 (10 iterations). (c) Circuit (10 iterations).

Figure 12: Benchmark weak scaling for Isometry and Realm.

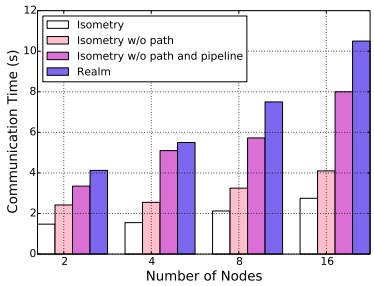


Figure 13: Transfer Performance for PageRank.

the path optimization, we force Isometry to use the same path as Realm. The results show that the path and pipeline optimizations speedup the transfers by 1.6× and 2.2×, respectively. The remainder of the speedup of Isometry over Realm for PageRank is most likely due to one other optimization we have not discussed: when possible Isometry actively manages the number of in-flight requests to guarantee high utilization of each channel, which Realm does not.

7 RELATED WORK

Direct data transfers. There has been considerable research on optimizing direct data movement, such as disk I/O [14, 24, 28], network traffic [21, 30], and communications between CPU memory and GPU framebuffers [12, 31]. Isometry builds on such work by incorporating methods for direct transfers into a framework for planning and executing multi-hop transfers.

Multi-hop data transfers. There are many approaches taken by prior work to support multi-hop data transfers. For example, distributed file systems (e.g., [3, 9]) allow accesses to disk on remote nodes. As another example, GPUfs [22] supports GPU and disk communication by providing a POSIX-like API to GPU programs for making the host's file system directly accessible to GPU programs.

MVAPICH2-GPU [27] is a MPI implementation optimized for data transfers from/to GPU framebuffers across multiple nodes. GPUDirect [20] provides a new interface between the GPU and the InfiniBand that allows both devices to directly access the same pinned system memory. Although GPUDirect eliminates the host CPU involvement by exploiting direct memory access, it requires software modifications in both the operating system and the device drivers. GPUDirect RDMA [2] supports direct communication between GPU framebuffers on different nodes, but it is only available in specific system configurations. Although offering low latency, GPUDirect RDMA has suboptimal bandwidth [18] and does not support layout transformation.

The difference between Isometry and other approaches is that Isometry seeks not to optimize specific multi-hop transfers for particular hardware configurations, but to provide a framework in which any multi-hop transfer in any complex machine can be optimized. Thus, Isometry both relieves upper levels of the software stack of reasoning about how to optimize data transfers while at the same time having a sufficiently global view to carry out significant data movement optimizations.

Parallel systems for heterogeneous architectures. Legion [7] is a parallel programming system for distributed heterogeneous architectures. The state-of-the-art DMA bsystem in Realm, Legion's low-level runtime, only supports direct transfers as described in Section 5.1. Isometry is a replacement for Realm's DMA subsystem that supports efficient multi-hop transfers. Dandelion [19] is a compiler and runtime for heterogeneous systems. Communication between DRAM and GPU framebuffers is performed in the Dandelion tasks, and the Dandelion DMA subsystem only supports data movement between DRAM and disks on different nodes. This design results in limited performance for multi-hop data transfers since optimizations such as pipelining direct transfers are not possible.

8 CONCLUSION

We have presented Isometry, a path-based distributed data transfer system for optimizing both direct and multi-hop data transfers with layout conversions in parallel systems. Key to our approach is a path planner that can select near-optimal paths for data transfers in complex systems. Our evaluation shows that Isometry is able to achieve high throughput, often approaching the hardware limits, while also minimizing the space used for intermediate buffers.

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1 as well as the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work used the XStream computational resource, supported by the National Science Foundation Major Research Instrumentation program (ACI-1429830), as well as the Moonlight cluster at Los Alamos National Laboratory through the ASC (Advanced Simulation and Computing) program.

REFERENCES

[1] [n. d.]. CUDA Programming Guide Version 5.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. ([n. d.]).

[2] [n. d.]. Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf. ([n. d.]).

[3] [n. d.]. Lustre File System. http://www.lustre.org. ([n. d.]).

[4] [n. d.]. The Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. ([n. d.]).

[5] [n. d.]. TensorFlow Performance Guide. https://www.tensorflow.org/performance/performance_guide. ([n. d.]).

[6] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre; Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* (2011).

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*.

[8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*.

[10] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short-vector SIMD Architectures *(CC'11/ETAPS'11)*.

[11] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, and Pat Hanrahan. 2008. A Portable Runtime Interface for Multi-level Memory Hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*.

[12] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[13] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 297–310.

[14] Zhihao Jia, Sean Treichler, Galen Shipman, Mike Bauer, Noah Watkins, Carlos Maltzahn, Pat McCormick, and Alex Aiken. 2017. Integrating External Resources with a Task-Based Programming Model. In *IEEE 24th International Conference on High Performance Computing (HiPC'17)*.

[15] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs *(SC '16)*.

[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report.

[17] Matt Pharr and William R Mark. 2012. ISPC: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar '12)*.

[18] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing (ICPP '13)*.

[19] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*.

[20] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. 2011. The Development of Mellanox/NVIDIA GPUDirect over InfiniBand–a New Model for GPU to GPU Communications. *Comput. Sci.* 26, 3-4 (2011).

[21] Galen M. Shipman, Stephen Poole, Pavel Shamis, and Ishai Rabinovitz. 2008. X-SRQ - Improving Scalability and Performance of Multi-core InfiniBand Clusters. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI '08)*.

[22] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.

[23] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[24] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '99)*.

[25] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic Algorithm Transformation for Efficient Multi-snapshot Analytics on Temporal Graphs. *Proc. VLDB Endow.* (2017).

[26] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[27] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. 2011. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Comput. Sci.* 26, 3-4 (2011).

[28] Noah Watkins, Zhihao Jia, Galen Shipman, Carlos Maltzahn, Alex Aiken, and Pat McCormick. 2015. Automatic and Transparent I/O Optimization with Storage Integrated Application Runtime Support. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW '15)*.

[29] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brain Mueller, Jason Samll, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*.

[30] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. 2006. High Performance RDMA Protocols in HPC. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI '06)*.

[31] S. Xiao and W. c. Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS '10)*.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2008. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI'12*. San Jose, CA.