# Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory

Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula
Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India 560012.
{roshan, chandan.g, thejas, uday}@csa.iisc.ernet.in

***Abstract*—Programming for parallel architectures that do not have a shared address space is extremely difficult due to the need for explicit communication between memories of different compute devices. A heterogeneous system with CPUs and multiple GPUs, or a distributed-memory cluster are examples of such systems. Past works that try to automate data movement for distributed-memory architectures can lead to excessive redundant communication. In this paper, we propose an automatic data movement scheme that minimizes the volume of communication between compute devices in heterogeneous and distributed-memory systems. We show that by partitioning data dependences in a particular non-trivial way, one can generate data movement code that results in the minimum volume for a vast majority of cases. The techniques are applicable to any sequence of affine loop nests and works on top of any choice of loop transformations, parallelization, and computation placement. The data movement code generated minimizes the volume of communication for a particular configuration of these. We use a combination of powerful static analyses relying on the polyhedral compiler framework and lightweight runtime routines they generate, to build a source-to-source transformation tool that automatically generates communication code. We demonstrate that the tool is scalable and leads to substantial gains in efficiency. On a heterogeneous system, the communication volume is reduced by a factor of 11× to 83× over state-of-the-art, translating into a mean execution time speedup of 1.53×. On a distributed-memory cluster, our scheme reduces the communication volume by a factor of 1.4× to 63.5× over state-of-the-art, resulting in a mean speedup of 1.55×. In addition, our scheme yields a mean speedup of 2.19× over hand-optimized UPC codes.**

***Keywords*—*communication optimization, data movement, polyhedral model, distributed memory, heterogeneous architectures.***

## I. Introduction

Different compute devices of a heterogeneous system with CPUs and multiple GPUs or a distributed-memory cluster do not typically share a global address space. Parallelizing code for such architectures is difficult due to the need for explicit communication between devices. If the compiler has to deal with partitioning and scheduling computation on such distributed-memory architectures, it has to address the problem of moving data between memories. The problem of generating efficient data movement code is common to both heterogeneous and distributed-memory systems. It is a key part of the larger parallelization problem for such systems, which involves other orthogonal sub-problems, such as computation and data transformations, computation placement and scheduling.

Most recent academic and industrial efforts in the area of compilation for heterogeneous architectures have been on building compilers for parallel programming models such as CUDA and OpenCL. Recent efforts such as OpenACC and C++ AMP [1] raise the level of abstraction and provide productivity, while programming accelerators, similar to that with OpenMP. Some proprietary compilers from PGI [2] and CAPS [3] also offer similar programming support. However, none of the known compilers for the above models support automatic distribution of loop computations across different devices in a heterogeneous system. They require programmers to explicitly move data in such a scenario. If any of the above programming models or compilers are to be extended to parallelize for synergistic execution on multiple devices while taking care of data movement automatically, they will have to deal with problems that we have addressed in this paper.

The techniques we develop are applicable to any sequence of arbitrarily nested loops with affine bounds and accesses, also known as *affine loop nests*. Affine loop nests form the compute-intensive core of computations like stencil-style computations, linear algebra kernels, alternating direction implicit (ADI) integrations. Past works on distributed-memory compilation deal with input more restrictive than affine loop nests [4], [5]. The schemes proposed in works which handle arbitrary affine loop nests while automating data movement for distributed-memory architectures [6]–[9] or for multi-device heterogeneous architectures [10] can lead to excessive redundant communication, as we explain later. Hence, there is no precise and efficient automatic data movement scheme at present which handles affine loop nests for distributed-memory or multi-device heterogeneous architectures.

The techniques we present rely on a combination of powerful static analyses employing the polyhedral compiler framework and lightweight runtime routines generated using them. Our approach statically determines data to be transferred between compute devices with a goal to move only those values that need to be moved in order to preserve program semantics. We show that by partitioning polyhedral data dependences in a particular non-trivial way, and determining communication sets and their receivers based on those partitions, one can determine communication data precisely, while avoiding both unnecessary and duplicate data from being communicated – a notoriously hard problem and a limitation of all previous polyhedral data movement works. Our scheme can handle any choice of loop transformations and parallelization. The code it generates

is parametric in problem size symbols and number of processors, and valid for any computation placement (static or dynamic). The data movement code generated minimizes the volume of communication, given a particular choice of all these.

Our contributions can be summarized as follows:

- We describe two new static analysis techniques to generate efficient data movement code between compute devices that do not share an address space.
- We implement these techniques in a source-level transformer to allow automatic distribution of loop computations on multiple CPUs and GPUs of a heterogeneous system, or on a distributed-memory cluster.
- We experimentally evaluate our techniques and compare it against existing schemes showing significant reduction in communication volume, translating into significantly better scaling and performance.

The rest of the paper is organized as follows. Section II provides background on the polyhedral model and existing communication schemes. Section III and Section IV describe our static analysis techniques to address the data movement problem. Section V describes our implementation. Experimental evaluation is presented in Section VI. Section VII discusses related work and conclusions are presented in Section VIII.

## II. Background

### A. Polyhedral Model

The polyhedral compiler framework provides a representation that captures the execution of a sequence of arbitrarily nested affine loop nests in a mathematical form suitable for analysis and transformation using machinery from linear algebra and linear programming. The iteration space of every statement can be represented as the set of integer points inside a (convex) polyhedron. The polyhedron is typically represented by a conjunction of affine inequalities. The dimensions of the polyhedron correspond to surrounding loop iterators as well as *program parameters*. Program parameters are symbols that do not vary in the portion of the code we are representing; they are typically the problem sizes. Each integer point in the polyhedron, also called an iteration vector, contains values for induction variables of loops surrounding the statement from outermost to innermost. The *data dependence graph*, $DDG = (\mathbf{S}, E)$ is a directed multi-graph with each vertex representing a statement in the program and edge $e$ from $S_i$ to $S_j$ representing a polyhedral dependence from a dynamic instance of $S_i$ to one of $S_j$. Every edge $e$ is characterized by a polyhedron $D_e$, called *dependence polyhedron* which precisely captures dependences between dynamic instances of $S_i$ and $S_j$. The dependence polyhedron is in the sum of the dimensionalities of the source and target iterations spaces, and the number of program parameters. The reader is referred to the Clan user guide [11] for a more detailed explanation of the polyhedral representation.

During analysis and transformation, one might end up with a union of convex polyhedra. Polylib [12], Omega [13], and ISL [14] are three libraries that provide operations to manipulate such a union of polyhedra. The latter two are precise and deal with integer points. All of the sets that we describe and compute in subsequent sections are unions of convex polyhedra, and any of the above libraries can be used to manipulate them.

```
for (t=1; t<=T-1; t++)
  for (i=1; i<=N-2; i++)
    a[t][i]=a[t-1][i-1]+a[t-1][i]+a[t-1][i+1];
```

Fig. 1: Jacobi-style stencil code

```
for (k=0; k<=N-1; k++)
  for (i=0; i<=N-1; i++)
    for (j=0; j<=N-1; j++)
      a[i][j]=((a[i][k]+a[k][j])<a[i][j])?(a[i][k]+a[k][j]): a[i][j];
```

Fig. 2: Floyd-Warshall code

### B. Existing communication schemes

It is known that anti (WAR) dependences and output (WAW) dependences do not necessitate communication; only the flow (RAW) dependences necessitate communication [9], [15]. Previous research [6] has also shown that it is essential for any communication scheme to perform *communication coalescing*, which reduces redundant communication by combining the data to be communicated due to multiple data accesses or dependences.

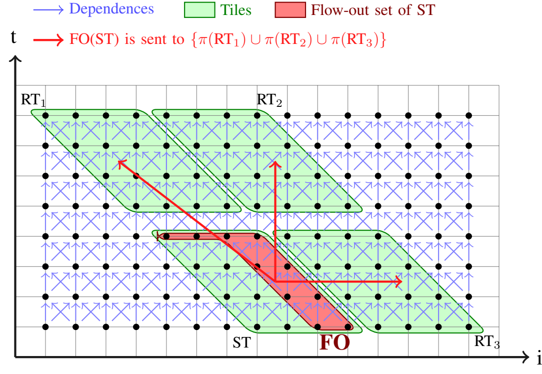
We provide a brief description of an existing communication scheme, termed **flow-out (FO) scheme** [9], that our schemes build upon. To illustrate the working of this scheme and our schemes, we use two examples – Jacobi-style stencil code in Fig. 1 and Floyd-Warshall code in Fig. 2, which are tiled and parallelized.

Bondhugula [9] describes static analysis techniques using the polyhedral compiler framework to determine data to be transferred between compute devices parametric in problem size symbols and number of processors, which is valid for any computation placement (static or dynamic). The key idea is that since code corresponding to a single iteration of the distributed loop will always be executed atomically by one compute device, communication parameterized on that iteration can be determined statically. So, an iteration of the distributed loop represents an atomic computation tile, on which communication is parameterized; the computation tile may or may not be a result of loop tiling.

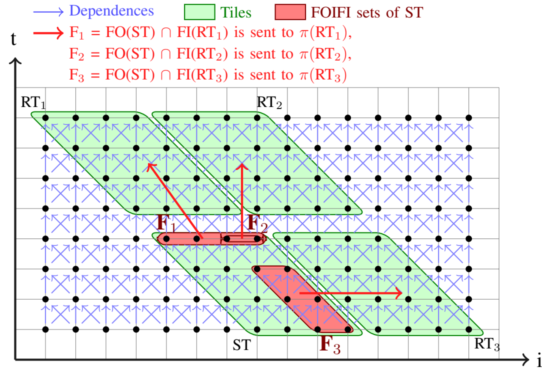
**Overview of FO scheme:** For each distributed loop, consider its iteration vector $\vec{i}$. For each data variable $x$, that can be a multidimensional array or a scalar, the following is determined parameterized on $\vec{i}$:

- Flow-out set, $FO_x(\vec{i})$: the set of elements that need to be communicated from iteration $\vec{i}$.
- Receiving iterations, $RI_x(\vec{i})$: the set of iterations of distributed loop(s) that require some element in $FO_x(\vec{i})$.

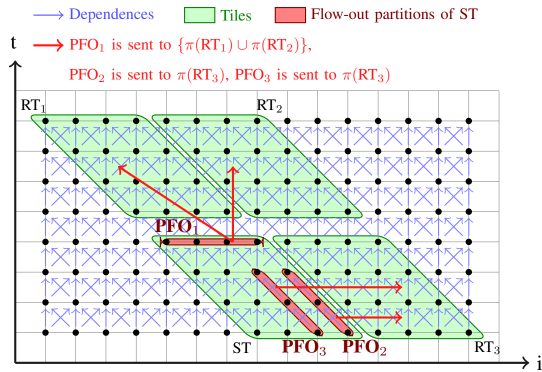
**Flow-out set:** The set of all values which flow from a write in an iteration to a read outside the iteration due to a



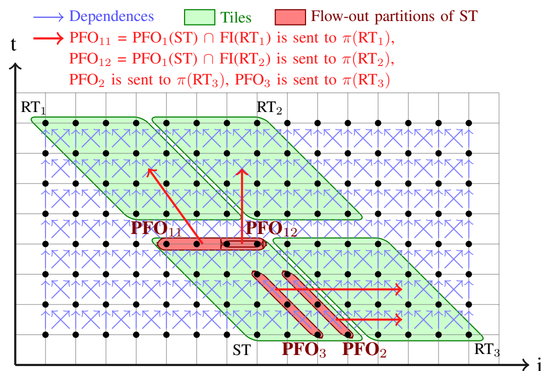

(a) FO scheme




(b) FOIFI scheme




(c) FOP scheme using multicast-pack




(d) FOP scheme using unicast-pack

Fig. 3: Jacobi-style stencil example

RAW dependence is termed as the **per-dependence flow-out set** corresponding to that iteration and dependence. For a RAW dependence polyhedron $D$ of data variable $x$ whose source statement is in $\vec{i}$, the per-dependence flow-out set $DFO_x(\vec{i}, D)$ is determined by computing the region of data $x$ written by those source iterations of $D$ whose writes are read outside $\vec{i}$. The flow-out set of an iteration is the set of all values written by that iteration, and then read outside the iteration. Therefore:

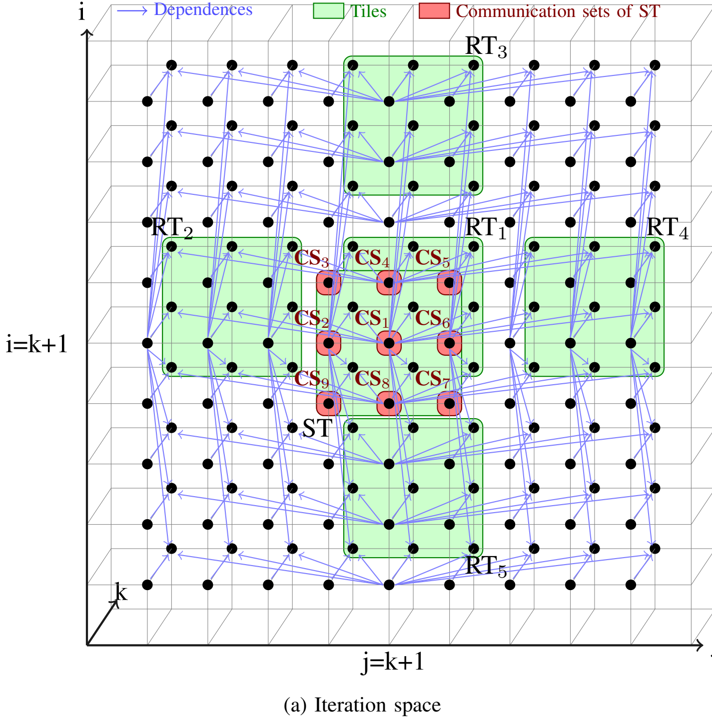
$$FO_x(\vec{i}) = \bigcup_{\forall D} DFO_x(\vec{i}, D) \qquad (1)$$

Since the flow-out set combines the data to be communicated due to multiple dependences, *communication coalescing* is implicitly achieved.

**Receiving iterations:** $RI_x(\vec{i})$ are the iterations $\vec{i'}$ of distributed loop(s) that read values written in $\vec{i}$ ($\vec{i'} \neq \vec{i}$). For each RAW dependence polyhedron $D$ of data variable $x$ whose source statement is in $\vec{i}$, $RI_x(\vec{i})$ is determined by projecting out dimensions inner to $\vec{i}$ in $D$ and scanning the target iterators while treating the source iterators as parameters. Since the goal is to determine the compute devices to communicate with, code is generated for a pair of helper functions $\pi(\vec{i})$ and $receivers_x(\vec{i})$. $\pi(\vec{i})$ returns the compute device that executes $\vec{i}$, while $receivers_x(\vec{i})$ returns the set of compute devices that require at least one element in $FO_x(\vec{i})$. $\pi$ is the placement function which maps an iteration of a distributed loop to a compute device. It is the inverse of the computation distribution function which maps a compute device to a set of iterations of the distributed loop(s) (which it executes). So, $\pi$ can be easily determined from the given computation distribution function. Since $\pi$ is evaluated only at runtime, the computation placement (or distribution) can be chosen dynamically. $receivers_x(\vec{i})$ enumerates the receiving iterations and makes use of $\pi$ on each receiving iteration to aggregate the set of distinct receivers ($\pi(\vec{i}) \notin receivers_x(\vec{i})$).

**Packing and unpacking:** The flow-out set of an iteration could be discontiguous in memory. So, at runtime, the generated code packs the flow-out set of each iteration executed by the compute device to a single buffer. The data is packed for an iteration $\vec{i}$ only if $receivers_x(\vec{i})$ is a non-empty set. The packed buffer is then sent to the set of receivers returned by $receivers_x(\vec{i})$ for all iterations $\vec{i}$ executed by it. After receiving data from other compute devices, the generated code unpacks the flow-out set of each iteration executed by every compute device other than itself from the respective received buffer. The data is unpacked for an iteration $\vec{i}$ only if $receivers_x(\vec{i})$ is a non-empty set, and if some data has been received from the compute device that executed $\vec{i}$. Both the packing code and the unpacking code traverse the iterations executed by a compute device, and the flow-out set of each iteration in the same order. Therefore, the offset of an element in the packed buffer of the sending compute device matches that in the received buffer of the receiving compute device. In order to allocate buffers for sending and receiving, reasonably tight upper bounds on the required size of buffers can be determined from the communication set constraints, but we do not present details on it due to space constraints.

**Communication volume:** All the elements in the flow-out set of an iteration might not be required by all its receiving




(a) Iteration space

FO(ST) = $\text{CS}_1 \cup \text{CS}_2 \cup \text{CS}_3 \cup \text{CS}_4 \cup \text{CS}_5 \cup \text{CS}_6 \cup \text{CS}_7 \cup \text{CS}_8 \cup \text{CS}_9$
FO(ST) is sent to $\{\pi(\text{RT}_1) \cup \pi(\text{RT}_2) \cup \pi(\text{RT}_3) \cup \pi(\text{RT}_4) \cup \pi(\text{RT}_5)\}$

(b) FO scheme

$\text{F}_1 = \text{CS}_1 \cup \text{CS}_2 \cup \text{CS}_3 \cup \text{CS}_4 \cup \text{CS}_5 \cup \text{CS}_6 \cup \text{CS}_7 \cup \text{CS}_8 \cup \text{CS}_9$
$\text{F}_1$ represents FO(ST) $\cap$ FI($\text{RT}_1$)
$\text{F}_1$ is sent to $\pi(\text{RT}_1)$
$\text{F}_2 = \text{CS}_1 \cup \text{CS}_4 \cup \text{CS}_8$
$\text{F}_2$ represents FO(ST) $\cap$ FI($\text{RT}_2$) and FO(ST) $\cap$ FI($\text{RT}_4$)
$\text{F}_2$ is sent to $\pi(\text{RT}_2)$ and $\pi(\text{RT}_4)$
$\text{F}_3 = \text{CS}_1 \cup \text{CS}_2 \cup \text{CS}_6$
$\text{F}_3$ represents FO(ST) $\cap$ FI($\text{RT}_3$) and FO(ST) $\cap$ FI($\text{RT}_5$)
$\text{F}_3$ is sent to $\pi(\text{RT}_3)$ and $\pi(\text{RT}_5)$

(c) FOIFI scheme

$\text{PFO}_1 = \text{CS}_1$
$\text{PFO}_1$ is sent to $\{\pi(\text{RT}_1) \cup \pi(\text{RT}_2) \cup \pi(\text{RT}_3) \cup \pi(\text{RT}_4) \cup \pi(\text{RT}_5)\}$
$\text{PFO}_2 = \text{CS}_4 \cup \text{CS}_8$
$\text{PFO}_2$ is sent to $\{\pi(\text{RT}_2) \cup \pi(\text{RT}_4)\}$
$\text{PFO}_3 = \text{CS}_2 \cup \text{CS}_6$
$\text{PFO}_3$ is sent to $\{\pi(\text{RT}_3) \cup \pi(\text{RT}_5)\}$
$\text{PFO}_4 = \text{CS}_3 \cup \text{CS}_5 \cup \text{CS}_7 \cup \text{CS}_9$
$\text{PFO}_4$ is sent to $\pi(\text{RT}_1)$

(d) FOP scheme using multicast-pack

Fig. 4: Floyd-Warshall example ($\text{CS}_i$ sets are used only for illustration; communication sets are determined as described in text)

iterations. As illustrated in Fig. 3a, if $\text{RT}_1$ and $\text{RT}_3$ are executed by different compute devices, then unnecessary data is communicated to those compute devices. Similarly, in Fig. 4b, if $\text{RT}_1$ and $\text{RT}_2$ are executed by different compute devices, then unnecessary data is communicated to the compute device that executes $\text{RT}_2$. Thus, this scheme could communicate large volume of unnecessary data since every element in the packed buffer need not be communicated to every receiver compute device; different receivers might require different elements in the packed buffer.

## III. Flow-out intersection Flow-in (FOIFI) scheme

FO scheme [9] could send unnecessary data since it only ensures that at least one element in the communicated data is required by the receiver. The goal, however, is that all elements in the data sent from one compute device to another compute device should be required by the receiver compute device. The problem in determining this at compile-time is that placement of iterations to compute devices is not known, even for a static computation distribution (like block-wise), since problem sizes and number of processes are not known. Nevertheless, data that needs to be sent from one iteration to another, parameterized on a sending iteration and a receiving iteration, can be determined precisely at compile-time.

**Overview:** For each distributed loop, consider its iteration vector $\vec{i}$. For each data variable $x$, the set of elements that need to be communicated from iteration $\vec{i}$ to iteration $\vec{i'}$ of a distributed loop, termed flow set – $F_x(\vec{i} \rightarrow \vec{i'})$, is determined at compile-time. For each iteration $\vec{i}$ executed by the compute device $c$ and iteration $\vec{i'} \in RI_x(\vec{i})$ that will be executed by another compute device $c' = \pi(\vec{i'})$ ($c' \neq c$), the generated code packs $F_x(\vec{i} \rightarrow \vec{i'})$ into the local buffer associated with $c'$. The packed buffers are then sent to their respective compute devices, and data is received from other compute devices. For each iteration $\vec{i}$ executed by another compute device $c'$ ($c' \neq c$) and iteration $\vec{i'} \in RI_x(\vec{i})$ that will be executed by the compute device $c$ ($\pi(\vec{i'}) = c$), the generated code unpacks $F_x(\vec{i} \rightarrow \vec{i'})$ from the received buffer associated with $c'$. Both the packing code and the unpacking code traverse the iterations $\vec{i}$ executed by a compute device, the receiving iterations $\vec{i'} \in RI_x(\vec{i})$, and the elements in $F_x(\vec{i} \rightarrow \vec{i'})$ in the same order.

**Flow-in set:** The set of all values which flow to a read in an iteration from a write outside the iteration due to a RAW dependence is termed as the **per-dependence flow-in set** corresponding to that iteration and dependence. For a RAW dependence polyhedron $D$ of data variable $x$ whose target statement is in $\vec{i}$, the per-dependence flow-in set $DFI_x(\vec{i}, D)$ is determined by computing the region of data $x$ read by those target iterations of $D$ whose reads are written outside $\vec{i}$. The flow-in set of an iteration is the set of all values read by that iteration, and previously written outside the iteration. Therefore:

$$FI_x(\vec{i}) = \bigcup_{\forall D} DFI_x(\vec{i}, D) \tag{2}$$

**Flow set:** The flow set from an iteration $\vec{i}$ to an iteration $\vec{i'}$ ($\vec{i} \neq \vec{i'}$) is the set of all values written by $\vec{i}$, and then read by $\vec{i'}$. For each data variable $x$, the flow set $F_x$ from an iteration $\vec{i}$ to an iteration $\vec{i'}$ is determined at compile-time by intersecting the flow-out set of $\vec{i}$ with the flow-in set of $\vec{i'}$:

$$F_x(\vec{i} \rightarrow \vec{i'}) = FO_x(\vec{i}) \cap FI_x(\vec{i'}) \tag{3}$$

Hence, this communication scheme is termed as the flow-out intersection flow-in (FOIFI) scheme. Since the flow set combines the data to be communicated due to multiple dependences, *communication coalescing* is implicitly achieved.

**Communication volume:** When each receiving iteration is executed by a different compute device, FOIFI scheme ensures that every element of the communicated data is required by the receiver, unlike FO scheme. The placement of iterations to compute devices, however, cannot be assumed. Different iterations can receive the same elements from the same sending iteration. So, when different receiving iterations of an iteration will be executed by the same compute device, this scheme could lead to duplication of data since it accumulates the flow sets to the buffer associated with the receiver compute device. For example, in Fig. 3b, if $RT_1$ and $RT_2$ are executed by the same compute device, then $F_2$ is sent twice to that compute device. Similarly, in Fig. 4c, if $RT_1$, $RT_2$ and $RT_4$ are executed by the same compute device, then $F_2$ is sent thrice to that compute device; the amount of duplication depends on the number of iterations a compute device executes in $j$ dimension with the same $i$ and $k$. Thus, this scheme could communicate a significantly large volume of duplicate data. The amount of redundancy cannot be theoretically bounded, and can be more than that of a naive scheme in the worst case.

## IV. Flow-out partitioning (FOP) scheme

FO scheme does not communicate duplicate data, but ignores whether a receiver requires most of the communication set or not. On the other hand, FOIFI scheme precisely computes the communication set required by a receiving iteration, but could lead to huge duplication when multiple receiving iterations are executed by the same compute device. A better approach is one that avoids communication of both duplicate and unnecessary elements. We show that this can be achieved by partitioning the communication set in a particular non-trivial way, and sending each partition to only its receivers.

The motivation behind partitioning the communication set is that different receivers could require different elements in the communication set. So ideally, the goal should be to partition the communication set such that all elements within each partition are required by all receivers of that partition. However, the receivers are not known at compile-time and partitioning at runtime is expensive. RAW dependences determine the receiving iterations, and ultimately, the receivers. Hence, we partition the communication set at compile-time, based on RAW dependences. To this end, we introduce new classifications for RAW dependences below.

*Definition 1:* A set of dependences is said to be **source-identical** if the region of data that flows due to each dependence in the set is the same.

Consider a set of RAW dependence polyhedra $S_D$ of an iteration $\vec{i}$. If $S_D$ is *source-identical*, then:

$$DFO_x(\vec{i}, D_1) = DFO_x(\vec{i}, D_2) \quad \forall D_1, D_2 \in S_D \tag{4}$$

*Definition 2:* Two source-identical sets of dependences are said to be **source-distinct** if the regions of data that flow due to the dependences in different sets are disjoint.

If two *source-identical* sets of RAW dependence polyhedra $S_D^1$ and $S_D^2$ of an iteration $\vec{i}$ are *source-distinct*, then:

$$\begin{aligned} DFO_x(\vec{i}, D_1) \cap DFO_x(\vec{i}, D_2) = \emptyset \\ \forall D_1 \in S_D^1, D_2 \in S_D^2 \end{aligned} \tag{5}$$

*Definition 3:* A **source-distinct partitioning** of dependences partitions the dependences such that all dependences in a partition are source-identical and any two partitions are source-distinct.

**Overview:** For each distributed loop, consider its iteration vector $\vec{i}$. For each data variable $x$, a *source-distinct* partitioning of RAW dependence polyhedra, whose source statement is in $\vec{i}$, is determined at compile-time. For each *source-identical* set (partition) of RAW dependence polyhedra $S_D$, the following is determined parameterized on $\vec{i}$:

- Partitioned flow-out set, $PFO_x(\vec{i}, S_D)$: the set of elements that need to be communicated from iteration $\vec{i}$ due to $S_D$.
- Partitioned flow set, $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$: the set of elements that need to be communicated from iteration $\vec{i}$ to iteration $\vec{i'}$ due to $S_D$.
- Receiving iterations of the partition, $RI_x(\vec{i}, S_D)$: the set of iterations of distributed loop(s) that require some element in $PFO_x(\vec{i}, S_D)$.

Using these parameterized sets, code is generated to execute the following in each compute device $c$ at runtime:

- For each *source-identical* set of RAW dependence polyhedra $S_D$ and iteration $\vec{i}$ executed by $c$, execute one of these:
  - `multicast-pack`: for each other compute device $c'$ ($c' \neq c$) that will execute some $\vec{i'} \in RI_x(\vec{i}, S_D)$, i.e., $c' = \pi(\vec{i'})$, pack $PFO_x(\vec{i}, S_D)$ into the local buffer associated with $c'$,
  - `unicast-pack`: for each iteration $\vec{i'} \in RI_x(\vec{i}, S_D)$ that will be executed by another compute device $c' = \pi(\vec{i'})$ ($c' \neq c$), pack $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ into the local buffer associated with $c'$,
- Send the packed buffers to the respective compute devices, and receive data from other compute devices,
- For each *source-identical* set of RAW dependence polyhedra $S_D$ and iteration $\vec{i}$ executed by another compute device $c'$ ($c' \neq c$), execute one of these:
  - unpack corresponding to `multicast-pack`: if $c$ will execute some $\vec{i'} \in RI_x(\vec{i}, S_D)$, i.e., $\pi(\vec{i'}) = c$, unpack $PFO_x(\vec{i}, S_D)$ from the received buffer associated with $c'$,
  - unpack corresponding to `unicast-pack`: for each iteration $\vec{i'} \in RI_x(\vec{i}, S_D)$ that will be executed by $c$, i.e., $\pi(\vec{i'}) = c$, unpack $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ from the received buffer associated with $c'$.

Both the packing code and the unpacking code traverse the sets of RAW dependence polyhedra $S_D$, the iterations $\vec{i}$ executed by

a compute device, the receiving iterations $\vec{i'} \in RI_x(\vec{i}, S_D)$, and the elements in $PFO_x(\vec{i}, S_D)$ or $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ in the same order. Therefore, the offset of an element in the packed buffer of the sending compute device matches that in the received buffer of the receiving compute device.

**Algorithm 1**: *source-distinct* partitioning of dependences

**Input**: RAW dependence polyhedra $D_i$ and $D_j$
1 $(I_S, A_S) \leftarrow$ source (iterations, access) of $D_i$
2 $(I_T, A_T) \leftarrow$ source (iterations, access) of $D_j$
3 $D \leftarrow$ dependence from $(I_S, A_S)$ to $(I_T, A_T)$
4 **if** $D$ *is empty* **then**
5 $\quad D_S \leftarrow D_T \leftarrow$ empty
6 $\quad$ return
7 $(I'_S, I'_T) \leftarrow$ (source, target) iterations of $D$
8 $D_S \leftarrow$ source $I'_S$ and target unconstrained
9 $D_T \leftarrow$ source $I'_T$ and target unconstrained
**Output**: *source-distinct* partitions
$\{D_i - D_S\}, \{D_j - D_T\}, \{D_i \cap D_S, D_j \cap D_T\}$

**Partitioning of dependences:** In order to partition dependences, it is necessary to determine whether the regions of data that flow due to two dependences overlap, i.e., whether the region of data written by the source iterations of one dependence overlaps with that of the other. This can be determined by an explicit dependence test between the source iterations of one dependence and the source iterations of another dependence. Such a dependence might not be semantically valid (e.g., when there is overlap in the regions of data that flow due to dependences with the same source statement). It is just a virtual dependence between two dependences, that captures the overlap in the regions of data that flow due to those dependences.

Algorithm 1 partitions two dependence polyhedra using this 'dependence between dependences' concept. If a virtual dependence does not exist between the two dependences, then they are *source-distinct*. Otherwise, the virtual dependence polyhedron contains the source iterations of each dependence polyhedron that access the same region of data. A new dependence polyhedron is formed from each dependence polyhedron by restricting the source iterations to their corresponding source iterations in the virtual dependence polyhedron. These two new dependences are *source-identical*. From the original dependence polyhedra, their corresponding source iterations in the virtual dependence polyhedron are subtracted out. These modified original dependences and the *source-identical* set of the new dependences are *source-distinct*.

Before partitioning dependences whose source statement is in $\vec{i}$, each RAW dependence polyhedron $D$ is restricted to those source iterations of $D$ whose writes are read outside $\vec{i}$. Initially, each dependence is in a separate partition. For any two partitions, Algorithm 1 is used as a subroutine for each pair of dependences in different partitions; the *source-identical* set of new dependences, if any, formed by all of these pairs is a new partition. This is repeated until no new partitions can be formed, i.e., until all partitions are *source-distinct*. The number of dependences in each new partition is the sum of those in the two partitions, and cannot be more than the number of initial dependences. The source iterations in each new dependence polyhedron keep monotonically decreasing. So, the partitioning should terminate, and a *source-distinct* partitioning always exists for any set of dependences. This simple approach can be improved upon and optimized; it is presented as is for clarity of exposition.

**Partitioned communication sets:** If $S_P$ is the set of *source-distinct* partitions of RAW dependence polyhedra whose source statement is in $\vec{i}$, then $\forall S_D \in S_P$:

$$\begin{aligned} PFO_x(\vec{i}, S_D) &= \bigcup_{\forall D \in S_D} DFO_x(\vec{i}, D) \\ &= DFO_x(\vec{i}, D) \quad \forall D \in S_D \quad \text{(from (4))} \end{aligned} \tag{6}$$

$$PF_x(\vec{i} \rightarrow \vec{i'}, S_D) = PFO_x(\vec{i}, S_D) \cap FI_x(\vec{i'}) \tag{7}$$

From Equation (5) and (6):

$$\begin{aligned} PFO_x(\vec{i}, S_D^1) \cap PFO_x(\vec{i}, S_D^2) = \emptyset \\ \forall S_D^1, S_D^2 \in S_P \mid S_D^1 \neq S_D^2 \end{aligned} \tag{8}$$

From Equation (1) and Definition 3:

$$FO_x(\vec{i}) = \bigcup_{\forall S_D \in S_P} PFO_x(\vec{i}, S_D) \tag{9}$$

Hence, this communication scheme is termed as flow-out partitioning (FOP) scheme. Since the flow-out partitions are disjoint and each of them combines the data to be communicated due to multiple dependences, this scheme reduces duplication, thereby achieving *communication coalescing*.

**Receiving iterations of the partition:** For iteration $\vec{i}$ and each RAW dependence polyhedron $D$ in the set of RAW dependence polyhedra $S_D$, $RI_x(\vec{i}, S_D)$ is determined by projecting out dimensions inner to $\vec{i}$ in $D$ and scanning the target iterators while treating the source iterators as parameters. The generated code makes use of $\pi$ on each receiving iteration of this partition to aggregate the set of distinct receivers that require at least one element in $PFO_x(\vec{i}, S_D)$.

**Packing and unpacking:** For each iteration and partition, either `multicast-pack` or `unicast-pack` is executed, the choice of which could be determined either at compile-time or at runtime. Since the method of packing determines the communication volume, the goal of choosing the method of packing is to minimize redundant communication. We choose:

- `unicast-pack` at *compile-time* if the set of dependence polyhedra determining the partition contains only one dependence polyhedron such that each source iteration in it has at most one target iteration dependent on it: each element in the partition is required by only one receiving iteration of the partition, and therefore, there is no duplication in the communicated data for any placement of iterations to compute devices.
- `multicast-pack` at *compile-time* if the flow-in of the partition is independent of the parallel dimension(s): each element in the partition is required by all receiving iterations of the partition, and therefore, unnecessary data is not communicated for any placement of iterations to compute devices.
- `unicast-pack` at *runtime* if each receiving iteration of the executed iteration and partition is allocated to a different compute device: there is no duplication in the communicated data since each element of the

communicated data is required by only one iteration that will executed by the receiving compute device.

- `multicast-pack` at *runtime* if all the receiving iterations of the executed iteration and partition are allocated to the same compute device: unnecessary data is not communicated since each element of the communicated data is required by some iteration that will be executed by that compute device.

These conditions ensure no redundancy in communication. In the absence or failure of these non-redundancy conditions, we choose `multicast-pack` since `unicast-pack` could lead to more redundant communication in the worst case.

**Communication volume:** The flow (RAW) dependence polyhedra for the Floyd-Warshall example in Fig. 2, where $(k, i, j)$ is the source iteration and $(k^{'}, i^{'}, j^{'})$ the target iteration, are: (the bounds on $i$, $j$, $k$ are omitted for brevity)

$$\begin{aligned} D_1 &= \{k^{'} = k+1, i^{'} = i, j^{'} = j\} \\ D_2 &= \{k^{'} = k+1, i^{'} = i, j = k+1, 0 <= j^{'} <= N-1\} \\ D_3 &= \{k^{'} = k+1, i = k+1, j^{'} = j, 0 <= i^{'} <= N-1\} \end{aligned}$$

The *source-distinct* partitioning of these dependence polyhedra yields four partitions: $P_1$ containing subsets of $D_1$, $D_2$ and $D_3$; $P_2$ containing subsets of $D_1$ and $D_2$; $P_3$ containing subsets of $D_1$ and $D_3$; $P_4$ containing a subset of $D_1$. As shown in Fig. 4d, $PFO_1$, $PFO_2$, $PFO_3$ and $PFO_4$ are the partitioned flow-out sets of $P_1$, $P_2$, $P_3$ and $P_4$ respectively, which are sent to their respective receivers using `multicast-pack`. There is no redundancy in communication for any placement of iterations to compute devices.

For the Jacobi-style stencil example in Fig. 1, where $(t, i)$ is the source iteration and $(t^{'}, i^{'})$ the target iteration, the flow (RAW) dependence polyhedra are:
(the bounds on $t$, $i$ are omitted for brevity)

$$\begin{aligned} D_1 &= \{t^{'} = t+1, i^{'} = i+1\} \\ D_2 &= \{t^{'} = t+1, i^{'} = i\} \\ D_3 &= \{t^{'} = t+1, i^{'} = i-1\} \end{aligned}$$

The *source-distinct* partitioning of these dependence polyhedra yields three partitions: $P_1$ containing subsets of $D_1$, $D_2$ and $D_3$; $P_2$ containing subsets of $D_1$ and $D_2$; $P_3$ containing a subset of $D_1$. As shown in Fig. 3c, $PFO_1$, $PFO_2$ and $PFO_3$ are the partitioned flow-out sets of $P_1$, $P_2$ and $P_3$ respectively. If `unicast-pack` is chosen for $PFO_1$ as shown in Fig. 3d when $RT_1$ and $RT_2$ are allocated to different compute devices, and `multicast-pack` is chosen otherwise (Fig. 3c), then there is no redundancy in communication for any placement of iterations to compute devices.

In general, if `unicast-pack` is used for all partitions, then FOP scheme behaves similar to FOIFI scheme. If `multicast-pack` is used for all partitions, then the communication volume of FOP scheme cannot be more than that of FO scheme. Depending on the method of packing, FOP scheme is at least as good as FO and FOIFI schemes. FOP scheme is effective in minimizing redundant communication since the partitions of the communication set reduce the granularity at which receivers are determined. To further minimize redundant data movement, FOP schemes uses the non-redundancy conditions to choose between `unicast-pack` and `multicast-pack`; the communication set partitions also reduce the granularity at which these conditions are applied. Hence, FOP scheme minimizes communication volume better than FO and FOIFI schemes.

## V. Implementation

Our framework is fully implemented as part of a publicly available source-to-source polyhedral tool chain. Clan [11], ISL [14], Pluto [16], and Cloog-isl [17] are used to perform polyhedral extraction, dependence testing, automatic transformation, and code generation, respectively. Polylib [12] is used to implement the polyhedral operations in Sections II-B, III and IV. ISL [14] is used to eliminate transitive dependences and compute *last writers* or the *exact dataflow*. This ensures that when there are multiple writes to a location before a subsequent read due to transitively covered RAW dependences, only the *last write* to the location is communicated to the compute device that reads it.

The input to our framework is sequential code containing arbitrarily nested affine loop nests, which is tiled and parallelized using the Pluto algorithm [18], [19]; loop tiling helps reduce the bookkeeping overhead at runtime while being precise in communication. Our framework then automatically generates code for distributed-memory systems from this transformed code using techniques described in the work of Bondhugula [9]. The entire data is initially made available in every compute device, and the final result is collected at the master compute device, without violating sequential semantics. FO, FOIFI and FOP schemes take the parallelized code as input and insert data movement code soon after each parallelized loop nest. So, at runtime, data movement code is executed after each distributed phase. In FOP, partitions with the same receiving tile constraints are merged and partitions with constant volume are merged at compile-time so as to minimize the bookkeeping overhead at runtime without sacrificing communication volume. Asynchronous MPI primitives are used to communicate between nodes in the distributed-memory system.

For heterogeneous systems, the host CPU acts both as a compute device and as the orchestrator of data movement between compute devices, while the GPU acts only as a compute device. The data movement code is automatically generated in terms of OpenCL calls invoked from the host CPU. Each compute device has an associated management thread on the host CPU. This thread is responsible for launching computation kernels and handling data movement to and from the compute device it is managing. The computations are distributed onto each device at the granularity of tiles. Once all the tiles are executed on a compute device, the management thread for that device calls the `copyOut` function which copies the communication set for each computed tile from the source device onto the host CPU. Once `copyOut()` completes, the management thread issues the `copyIn` function which copies the communication set now residing on the host CPU onto the destination compute devices. On heterogeneous systems, packing functionally corresponds to `copyOut()` and unpacking corresponds to `copyIn()`. We notice that, for most of the cases, the data to be communicated is in a rectangular region of memory.

OpenCL 1.1 provides `clEnqueueReadBufferRect()` and `clEnqueueWriteBufferRect()` to copy such rectangular regions of data in a single call. We make use of these functions everywhere to minimize the number of OpenCL calls and thereby minimize the associated call overheads.

## VI. Experimental Evaluation

We compare FOP, FOIFI and FO schemes using the same parallelizing transformation. This implies that the frequency of communication is the same for them. The schemes differ only in the communication volume, and in the way the data is packed and unpacked. Since everything else is the same, comparing the total execution times of these communication schemes directly compares their efficiency.

### *A. Distributed-memory architectures*

**Setup:** The experiments were run on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node on the cluster consists of two quad-core Intel Xeon E5430 2.66 GHz processors with 12 MB L2 cache and 16 GB RAM. The InfiniBand host adapter is a Mellanox MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8 as the MPI implementation. It provides a point-to-point latency of 3.36 $\mu$s, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively. All codes were compiled with Intel C compiler (ICC) version 11.1 with flags '-O3 -fp-model precise'. All Unified Parallel C (UPC) codes were compiled with Berkeley Unified Parallel C compiler [20] version 2.16.0.

**Benchmarks:** We present results for Floyd Warshall (`floyd`), LU Decomposition (`lu`), Alternating Direction Implicit solver (`adi`), 2-D Finite Different Time Domain Kernel (`fdtd-2d`), Heat 2D equation (`heat-2d`) and Heat 3D equation (`heat-3d`) benchmarks. The first four are from the publicly available Polybench/C 3.2 suite [21]; `heat-2d` and `heat-3d` are widely used stencil computations [22]. All benchmarks were manually ported to UPC, while sharing data only if it may be accessed remotely and incorporating UPC-specific optimizations like localized array accesses, block copy, one-sided communication, where applicable. The outermost parallel loop in `heat-2d`, `heat-3d` and `adi` was marked as parallel using OpenMP, and given as input to OMPD [4]; since the data to be communicated in `floyd` and `lu` is dependent on the outer serial loop, OMPD does not handle them. In our tool which implements FOP, FOIFI and FO, the benchmark itself was the input and tile sizes were chosen such that the performance on a single node is optimized, as listed in Table I. All benchmarks use double-precision floating-point operations. Problem sizes used are listed in Table I.

**Evaluation:** Though our tool generates MPI+OpenMP code, we ran all the benchmarks with one OpenMP thread per process and one MPI process per node to focus on the distributed-memory part. Table I compares the total communication volume of FO, FOIFI and FOP. Table II compares the total execution time of hand-optimized UPC codes with that of OMPD, FO, FOIFI and FOP. `seq` – sequential time, is the time taken to run the serial code compiled with ICC. Execution time of FO, FOIFI and FOP on one node is different from `seq` time due to tiling and other loop transformations performed by our tool on the sequential code.

Across all benchmarks and number of nodes, FOP reduced communication volume by a factor of 1.4$\times$ to 63.5$\times$ over FO. This translates to a huge improvement in execution time, except for `heat-2d`, where the communication time is a minor component of the total execution time (less than 2% in most cases). For `adi` and `floyd` which are communication intensive, the reduction in communication volume for FOP gives up to 15.9$\times$ speedup over FO. For `floyd` and `lu`, FOIFI communicates 1.5$\times$ to 31.8$\times$ more volume of data than FOP due to duplication in communicated data when multiple receiving iterations are placed on the same node. This yields 1.1$\times$ to 1.84$\times$ speedup of FOP over FOIFI. In summary, FOP gives a mean speedup of 1.55$\times$ and 1.11$\times$ over FO and FOIFI respectively.

OMPD communicates the same volume of data as FOP for `adi`. However, OMPD incurs some additional overhead since it determines the communication sets at runtime. Moreover, our tool automatically tiles the parallel loop. For `heat-2d` and `heat-3d`, our tool transforms and tiles the code to yield both locality and load balance [19]. OMPD cannot handle such transformed code since the communication set is dependent on the outer serial loop. Even though OMPD is communicating the minimum volume of data for the non-transformed codes, FOP gives a mean speedup of 3.06$\times$ over OMPD since it handles transformed codes with lesser runtime overhead.

Manually developed UPC codes communicate the same volume of data for `fdtd-2d` and `floyd` as FOP. Since the data to be communicated is contiguous in global memory, UPC code has no additional overhead, and so, performs slightly better than FOP. On the other hand, the data to be communicated for `adi` is not contiguous in global memory. FOP packs and unpacks such non-contiguous data with very little runtime overhead while UPC incurs significant runtime overhead to handle such multiple shared memory requests to non-contiguous data. So, even though the data to be communicated is the same, FOP outperforms UPC code. For `lu`, `heat-2d` and `heat-3d`, writing UPC code incorporating the transformations automatically used by our tool is not trivial; without such transformations, UPC code performs poorly. Due to these limitations of UPC, FOP gives a mean speedup of 2.19$\times$ over hand-optimized UPC codes.

For the transformations and placement chosen in the benchmarks, we manually verified that FOP was achieving the minimum communication volume, resulting in the best performance and facilitating the benchmarks to scale well. As shown in Fig. 5, execution times of FOP decrease as the number of nodes are increased for all benchmarks, except for `lu` going from 16 to 32 nodes. In this case, performance does not improve by much due to the large volume of data that is required to be communicated. Nevertheless, `floyd` with the existing FO could not scale beyond 4 nodes, while FOP enables scaling similar to hand-optimized UPC codes as shown in Fig. 6.

### *B. Heterogeneous architectures*

**Intel-NVIDIA system setup:** The Intel-NVIDIA system consists of an Intel Xeon multicore server consisting of 12 Xeon E5645 cores running at 2.4 GHz. The server has 4 NVIDIA Tesla C2050 graphics processors connected on the PCI express bus, each having 2.5 GB of global memory. NVIDIA driver version 304.64 supporting OpenCL 1.1 was

TABLE I: Total communication volume on distributed-memory cluster – FO and FOIFI normalized to FOP

| Benchmark | Problem sizes | Tile sizes | 4 nodes | | | 8 nodes | | | 16 nodes | | | 32 nodes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | FOP | FOIFI | FO | FOP | FOIFI | FO | FOP | FOIFI | FO | FOP | FOIFI | FO |
| floyd | $8192^2$ | $64^2$ | 1.51GB | 31.8× | 63.5× | 3.53GB | 15.9× | 63.5× | 7.56GB | 7.9× | 63.5× | 15.62GB | 4.0× | 63.5× |
| lu | $4096^2$ | $64^2$ | 0.45GB | 5.3× | 1.4× | 0.99GB | 3.0× | 1.4× | 1.88GB | 1.9× | 1.4× | 2.59GB | 1.5× | 1.5× |
| fdtd-2d | $1024x4096^2$ | $16^2$ | 0.21GB | 1.0× | 14.3× | 0.47GB | 1.0× | 15.1× | 0.97GB | 1.0× | 15.5× | 1.97GB | 1.0× | 15.7× |
| heat-2d | $1024x8192^2$ | $256^3$ | 0.75GB | 1.0× | 2.0× | 1.74GB | 1.0× | 2.0× | 3.73GB | 1.0× | 2.0× | 7.72GB | 1.0× | 2.0× |
| heat-3d | $256x512^3$ | $16^4$ | 5.61GB | 1.0× | 2.0× | 13.09GB | 1.0× | 2.0× | 28.07GB | 1.0× | 2.0× | 58.01GB | 1.0× | 2.0× |
| adi | $128x8192^2$ | $256^2$ | 191.24GB | 1.0× | 4.0× | 223.11GB | 1.0× | 8.0× | 239.05GB | 1.0× | 16.0× | 247.02GB | 1.0× | 32.0× |

TABLE II: Total execution time on distributed-memory cluster – FOIFI, FO, OMPD and UPC normalized to FOP

(a) floyd – `seq` time is 2012s

| Nodes | FOP | FOIFI | FO | UPC |
|---|---|---|---|---|
| 1 | 2065.2s | 1.01× | 1.00× | 0.97× |
| 4 | 521.4s | 1.10× | 1.20× | 0.97× |
| 8 | 263.9s | 1.18× | 1.75× | 0.97× |
| 16 | 137.6s | 1.33× | 3.93× | 0.97× |
| 32 | 81.1s | 1.46× | 11.18× | 0.93× |

(b) lu – `seq` time is 82.9s

| Nodes | FOP | FOIFI | FO | UPC |
|---|---|---|---|---|
| 1 | 29.5s | 1.00× | 1.00× | 2.86× |
| 4 | 9.1s | 1.42× | 1.02× | 2.42× |
| 8 | 5.4s | 1.70× | 1.05× | 2.30× |
| 16 | 4.1s | 1.84× | 1.05× | 1.50× |
| 32 | 3.9s | 1.58× | 1.00× | 1.25× |

(c) fdtd-2d – `seq` time is 351.7s

| Nodes | FOP | FOIFI | FO | UPC |
|---|---|---|---|---|
| 1 | 359.5s | 1.00× | 1.00× | 0.98× |
| 4 | 90.8s | 1.00× | 1.03× | 1.26× |
| 8 | 66.9s | 1.00× | 1.04× | 1.01× |
| 16 | 33.8s | 1.00× | 1.09× | 1.01× |
| 32 | 16.8s | 1.00× | 1.24× | 0.99× |

(d) heat-2d – `seq` time is 796.4s

| Nodes | FOP | FOIFI | FO | OMPD | UPC |
|---|---|---|---|---|---|
| 1 | 228.3s | 1.00× | 1.00× | 3.42× | 5.33× |
| 4 | 59.8s | 1.00× | 1.01× | 3.29× | 5.11× |
| 8 | 31.4s | 1.00× | 1.02× | 3.92× | 5.47× |
| 16 | 17.3s | 1.00× | 1.03× | 3.58× | 5.00× |
| 32 | 10.2s | 1.00× | 1.04× | 3.06× | 4.25× |

(e) heat-3d – `seq` time is 590.6s

| Nodes | FOP | FOIFI | FO | OMPD | UPC |
|---|---|---|---|---|---|
| 1 | 235.5s | 1.00× | 1.00× | 2.51× | 2.68× |
| 4 | 65.4s | 1.00× | 1.05× | 2.39× | 2.46× |
| 8 | 36.1s | 1.00× | 1.15× | 2.82× | 2.54× |
| 16 | 21.4s | 1.00× | 1.23× | 2.58× | 2.21× |
| 32 | 14.1s | 1.00× | 1.33× | 2.29× | 1.78× |

(f) adi – `seq` time is 2717s

| Nodes | FOP | FOIFI | FO | OMPD | UPC |
|---|---|---|---|---|---|
| 1 | 422.7s | 1.00× | 0.95× | 6.27× | 7.90× |
| 4 | 231.7s | 1.00× | 2.11× | 3.55× | 4.68× |
| 8 | 143.6s | 1.00× | 4.00× | 3.43× | 4.29× |
| 16 | 78.6s | 1.00× | 7.87× | 2.88× | 4.47× |
| 32 | 41.0s | 1.00× | 15.9× | 2.95× | 5.22× |

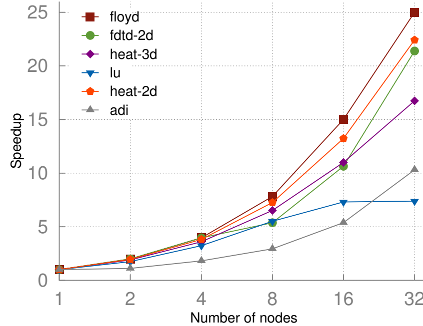



Fig. 5: FOP – strong scaling on distributed-memory cluster

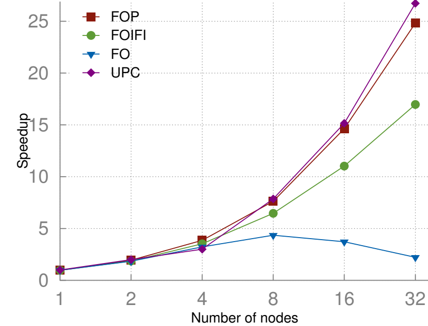



Fig. 6: `floyd` – speedup of FOP, FOIFI, FO and hand-optimized UPC code over `seq` on distributed-memory cluster

used as the OpenCL runtime. Double-precision floating-point operations were used in all benchmarks. The host codes were compiled with gcc version 4.4 with -O3.

**AMD system setup:** The AMD system consists of a AMD A8-3850 Fusion APU, consisting of 4 CPU cores running at 2.9 GHz and an integrated GPU based on the AMD Radeon HD 6550D architecture. The system has two ATI FirePro V4800 discrete graphics processors connected on the PCI express bus, each having 512 MB of global memory. Since these GPUs do not support double-precision floating-point operations, we use single-precision floating-point operations in all benchmarks. AMD driver version 9.82 supporting OpenCL 1.2 was used as the OpenCL runtime. The host codes were compiled with g++ version 4.6.1 with -O3.

**Benchmarks:** We evaluate FO and FOP for `floyd`, `lu`, `fdtd-2d`, `heat-2d` and `heat-3d` benchmarks. All these benchmarks have an outer serial loop containing a set of inner parallel loops. Wherever multiple nested parallel loops existed, the outermost among them was distributed across devices. The OpenCL kernels were manually written by mapping the parallel loops in a DOALL manner onto the OpenCL work groups and work items.

**Evaluation:** We consider the following combination of compute devices: (1) 1 CPU, (2) 1 GPU, (3) 1 CPU + 1 GPU, (4) 2 GPUs, (5) 4 GPUs. We evaluate FO and FOP on the Intel-NVIDIA system for all these cases. On the AMD system, we evaluate FO and FOP for (1), (2) and (4) cases, using only the discrete GPUs. In the first two cases, the devices run the entire OpenCL kernel. For cases (3), (4) and (5), kernel execution is partitioned across devices. For (4) and (5), the computation is equally distributed (block-wise). Since the CPU and GPUs have different compute powers, the computation distributions

TABLE III: Results on the Intel-NVIDIA system

| Benchmark | Problem sizes | Tile sizes | Device combination | Total execution time | | | | Total communication volume | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | - | FOP | FO | Speedup | FOP | FO | Reduction |
| floyd | 10240x10240 | 32x32 | 1 CPU (12 cores) | 890s | – | – | – | – | – | – |
| | | | 1 GPU | 113s | – | – | – | – | – | – |
| | | | 1 CPU + 1 GPU | – | 148s | 180s | 1.22 | 0.8 GB | 25.0 GB | 32 |
| | | | 2 GPUs | – | 65s | 104s | 1.60 | 1.6 GB | 51.0 GB | 32 |
| | | | 4 GPUs | – | 43s | 107s | 2.49 | 3.1 GB | 102.0 GB | 32 |
| lu | 11264x11264 | 256x256 | 1 CPU (12 cores) | 412s | – | – | – | – | – | – |
| | | | 1 GPU | 77s | – | – | – | – | – | – |
| | | | 1 CPU + 1 GPU | – | 92s | 132s | 1.43 | 0.9 GB | 63 GB | 70 |
| | | | 2 GPUs | – | 64s | 147s | 2.30 | 0.7 GB | 62.0 GB | 83 |
| | | | 4 GPUs | – | 60s | 208s | 3.47 | 1.2 GB | 63.0 GB | 51 |
| fdtd-2d | 4096x10240x10240 | 32x32 | 1 CPU (12 cores) | 1915s | – | – | – | – | – | – |
| | | | 1 GPU | 397s | – | – | – | – | – | – |
| | | | 1 CPU + 1 GPU | – | 580s | 603s | 1.03 | 0.9 GB | 11.0 GB | 11 |
| | | | 2 GPUs | – | 207s | 236s | 1.14 | 0.9 GB | 22.0 GB | 22 |
| | | | 4 GPUs | – | 117s | 164s | 1.40 | 2.2 GB | 62.0 GB | 28 |
| heat-2d | 4096x10240x10240 | 32x32 | 1 CPU (12 cores) | 1112s | – | – | – | – | – | – |
| | | | 1 GPU | 266s | – | – | – | – | – | – |
| | | | 1 CPU + 1 GPU | – | 242s | 255s | 1.05 | 0.6 GB | 21.0 GB | 32 |
| | | | 2 GPUs | – | 138s | 157s | 1.14 | 0.6 GB | 21.0 GB | 32 |
| | | | 4 GPUs | – | 80s | 124s | 1.55 | 1.9 GB | 62.0 GB | 32 |
| heat-3d | 4096x512x512x512 | 32x32x32 | 1 CPU (12 cores) | 3080s | – | – | – | – | – | – |
| | | | 1 GPU | 1932s | – | – | – | – | – | – |
| | | | 1 CPU + 1 GPU | – | 1718s | 2018s | 1.17 | 16.0 GB | 512.0 GB | 32 |
| | | | 2 GPUs | – | 1086s | 1379s | 1.26 | 16.0 GB | 512.0 GB | 32 |
| | | | 4 GPUs | – | 670s | 1658s | 2.47 | 49.0 GB | 1535.4 GB | 32 |

TABLE IV: Results on the AMD system

| Benchmark | Problem sizes | Tile sizes | Device combination | Total execution time | | | | Total communication volume | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | - | FOP | FO | Speedup | FOP | FO | Reduction |
| floyd | 10240x10240 | 32x32 | 1 CPU (4 cores) | 1084s | – | – | – | – | – | – |
| | | | 1 GPU | 512s | – | – | – | – | – | – |
| | | | 2 GPUs | – | 286s | 305s | 1.07 | 0.8 GB | 25.0 GB | 32 |
| fdtd-2d | 4096x5120x5120 | 32x32 | 1 CPU (4 cores) | 1529s | – | – | – | – | – | – |
| | | | 1 GPU | 241s | – | – | – | – | – | – |
| | | | 2 GPUs | – | 133s | 242s | 1.82 | 0.2 GB | 2.15 GB | 17 |
| heat-2d | 4096x8192x8192 | 32x32 | 1 CPU (4 cores) | 3654s | – | – | – | – | – | – |
| | | | 1 GPU | 256s | – | – | – | – | – | – |
| | | | 2 GPUs | – | 142s | 353s | 2.49 | 0.25 GB | 8.0 GB | 32 |

were chosen to be asymmetric for case (3). For all benchmarks, case (3) had 10% of computation distributed onto the CPU and 90% onto the GPU.

**Results:** Table III shows results obtained on the Intel-NVIDIA system. For all benchmarks, the running time on 1 GPU is much lower than that on the 12-core CPU. This running time is further improved by distributing the computation onto 2 and 4 GPUs. For all benchmarks, we see that FOP significantly reduces communication volume over FO. The computation tile sizes directly affects the communication volume (e.g., 32$\times$ for `floyd`). For the transformations and placement chosen for these benchmarks, we manually verified that FOP achieved the minimum communication volume. This reduction in communication volume results in a corresponding reduction in execution time facilitating strong scaling of these benchmarks, as shown in Fig. 7 – this was not possible with the existing FO. For example, FO for `heat-3d` has very high communication overhead and does not scale beyond two GPUs. For `floyd` and `lu`, FO scales up to 2 GPUs, but not beyond it. However, FOP easily scales up to 4 GPUs for all benchmarks. For `floyd`, `lu` and `fdtd-2d`, CPU's performance becomes the bottleneck, even when it only executed 10% of the computation. Hence, we observe 1 CPU + 1 GPU performance to be worse than 1 GPU performance for these benchmarks. On the other hand, 1 CPU + 1 GPU gives 9% and 11% improvement over 1 GPU for `heat-2d` and `heat-3d` respectively.

Table IV shows results obtained on the AMD system. The OpenCL functions used to transfer rectangular regions of memory are crucial for copying non-contiguous (strided) data efficiently. We found these functions to have a prohibitively high overhead on this system. This compelled us to use only those functions which could copy contiguous regions of memory. Hence, we present results only for `floyd`, `heat-2d` and `fdtd-2d` since the data to be moved for these benchmarks is contiguous. For all benchmarks, the running time on 1 GPU is much lower than that on the 4-core CPU. We could not evaluate them on 1 CPU + 1 GPU since the OpenCL data transfer functions crashed when CPU was used as an OpenCL device. FO does not perform well on 2 GPUs for `heat-2d` and `fdtd-2d` since these benchmarks have a low compute-to-copy ratio and the high volume of communication in FO leads to a slowdown. The FOP scheme, on the other hand, performs very well on 2 GPUs, yielding a near-ideal speedup of 1.8$\times$ over 1 GPU for all benchmarks.

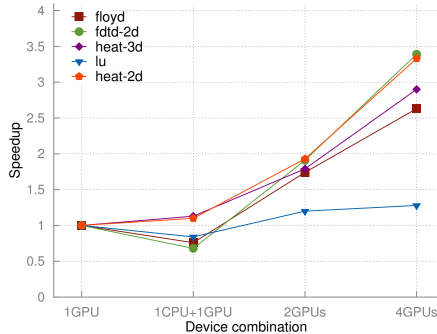



Fig. 7: FOP – strong scaling on the Intel-NVIDIA system

## VII. Related work

Works from literature closely related to communication code generation for distributed-memory architectures are: LWT – Amarasinghe and Lam [5]; dHPF – Adve and Mellor-Crummey [6], and Chavarría-Miranda and Mellor-Crummey [7]; CLGR – Claßen and Griebl [8]; FO – Bondhugula [9]; OMPD – Kwon et al. [4]. These abbreviations will be used to refer these works. LWT, dHPF, CLGR and FO statically determine the data to be communicated and generate code for it, whereas OMPD determines the data to be communicated primarily using a runtime dataflow analysis technique. LWT handles only perfectly nested loops; OMPD handles only those affine loop nests which have a repetitive communication pattern (i.e., those which transfer the same set of data on every invocation of the parallel loop); dHPF, CLGR and FO are based on the polyhedral framework, like our schemes, and can handle any sequence of affine loop nests. Our framework builds upon and subsumes the state-of-the-art automatic distributed-memory code generation framework [9], while generalizing it to target heterogeneous architectures.

LWT, dHPF and CLGR use a virtual processor to physical processor mapping to handle symbolic problem sizes and number of processors. If iterations of the distributed loop(s) are treated as virtual processors, then FOIFI statically determines the communication set between two virtual processors, and uses $\pi$ at runtime as a mapping function from virtual to physical processors. Thus, FOIFI also uses a virtual processor model. For all these schemes, the communication code is generated such that virtual processors communicate with each other only when they are mapped to different physical processors. In spite of this, when the data being sent to different virtual processors is not disjoint and when some of those virtual processors are mapped to the same physical processor, the common portion of the data is sent multiple times to that physical processor. dHPF [7] overcomes some of this redundancy by statically coalescing data required by multiple loop nests. FOP and FO also achieve *communication coalescing* across multiple loop nests by determining the communication set for all dependences, i.e., for all dependent loop nests. Moreover, instead of a virtual processor approach, FOP and FO determine the set of receivers precisely so as to not communicate duplicate data.

dHPF determines communication code by analyzing data accesses as opposed to dependences in a way that lacks *exact dataflow* information. dHPF could be either pulling the data just before it is consumed or pushing the data soon after it is produced. In the former scenario, when there are multiple reads to the same location that are spread across distributed phases, the read in each distributed phase gets the data in that location from the owner processor, though only the *first read* is required to get it. In the latter scenario, when there are multiple writes to the same location that are spread across distributed phases, the write in each distributed phase is sent to all processors which read that location, though only the *last write* is required to be sent. In contrast, FOP, FOIFI and FO use the *last writer* property of flow dependences to communicate only the *last write*.

As for *communication coalescing*, LWT and CLGR do not perform it for arbitrary affine accesses, unlike FOP, FOIFI and FO. So, they could communicate duplicate data when there are multiple references to the same data.

FO unnecessarily communicates the entire communication set when different receivers require different elements in the communication set. FOP reduces such unnecessary communication by partitioning the communication set and precisely determining the set of receivers for each partition.

Thus, LWT, dHPF, CLGR, FO and FOIFI schemes could lead to substantially larger volume of redundant communication than FOP scheme. Since FOIFI precisely determines the data which needs to be communicated between virtual processors by analyzing multiple dependences simultaneously, it is theoretically at least as good as schemes that use the virtual processor model, like LWT, dHPF and CLGR. Our evaluation shows that FOP clearly outperforms OMPD, FO, FOIFI, and consequently, LWT, dHPF and CLGR.

Among existing works that support distributing computation on multiple devices of a heterogeneous system [2], [3], [10], [23], [24], the work of Kim et al. [10] is the only one which completely automates data movement. Their input is an OpenCL program for a single device, which is distributed across multiple compute devices. The kernels in the program can only have affine array accesses. They determine the first and last memory location accessed by a computation partition and then send the entire data in that range to the compute device which would execute that partition. This could lead to false sharing since there could be many memory locations within the range that are not required by the associated partition. Thus, their scheme communicates significantly large volume of redundant data. To ensure consistency, they maintain a separate *virtual* buffer in the host, which is ‘diff’ed and ‘merge’d with the GPUs’ buffers at runtime when required. This introduces additional runtime overhead. FOP, on the other hand, precisely determines memory locations that need to be communicated through static analyses and with minimal runtime overhead. However, we are unable to present an experimental comparison with their scheme as it is not available. Leung et al. [23] describe an automatic source-level transformer in the RSTREAM compiler [25] which generates CUDA code from serial C code. Their work targets systems with multiple GPUs, but no details on inter-device data movement or results on multiple GPUs are provided. Song and Dongarra [24] execute linear algebra kernels on heterogeneous GPU-based clusters. They develop a multi-level partitioning and distribution method, which is orthogonal to the problem we address. Their communication scheme is not automatic, but

specific to the kernels addressed. Communication is driven by data dependences between atomic tasks. Since they send the entire output data of a task to any task that reads at least one value in that output data, they could send unnecessary data like FO. In contrast, FOP minimizes such redundant communication for the chosen distribution. Among production compilers, PGI [26] and CAPS [3] have a proprietary directive based accelerator programming model, and also support OpenACC. However, to the best of our knowledge, they do not automatically distribute loop computations across different devices of a heterogeneous system. So, the issues of automatic data movement or synchronization between different devices do not arise.

CGCM [27], DyManD [28], and AMM [29] are recent works that support only a single GPU device, but automate data movement between CPU and GPU. They allocate the entire data on every device. Data is transferred at the granularity of an allocation unit in CGCM and DyManD, and at the granularity of a CUDA X10 Rail in AMM, which could lead to redundant communication. FOP, by contrast, is precise in determining data to be transferred at the granularity of array elements. However, our approach is for affine array accesses and is thus complementary to CGCM and DyManD that are designed to also handle pointers and recursive data structures respectively.

## VIII. Conclusions

We proposed compilation techniques to free programmers from the burden of moving data on architectures that do not have a shared address space. We were able to generate efficient data movement code statically using a *source-distinct* partitioning of dependences. Minimum communication volume was achieved with a majority of dependence patterns and for all benchmarks considered. To the best of our knowledge, our tool is the first one to parallelize affine loop nests for a combination of CPUs and GPUs while providing precision of data movement at the granularity of array elements. On a heterogeneous system, we showed that our scheme (FOP) reduces the communication volume by a factor of $11\times$ to $83\times$, resulting in a mean execution time speedup of $1.53\times$ over the best existing scheme (FO). For communication-intensive benchmarks like Floyd-Warshall, when running on 4 GPUs, we demonstrated that our data movement scheme yields a mean speedup of $2.56\times$ over 1 GPU. On a distributed-memory cluster, we showed that our scheme (FOP) reduces the communication volume by a factor of $1.4\times$ to $63.5\times$, resulting in a mean speedup of $1.72\times$ over the best existing scheme (FO). Our scheme gave a mean speedup of $3.06\times$ over another existing scheme (OMPD) and a mean speedup of $2.19\times$ over hand-optimized UPC versions of these codes. We believe that our techniques will be able to provide OpenMP-like programmer productivity for distributed-memory and heterogeneous architectures if implemented in compilers.

## Acknowledgments

This work was supported in part by a research gift from AMD. We would like to acknowledge the Department of Science and Technology, India for a grant under the FIST program. We would also like to thank NVIDIA for donating us GPUs under the NVIDIA CUDA research center program.

## References

[1] "C++ AMP," http://msdn.microsoft.com/en-us/library/hh265137.aspx.

[2] "Portland Group Inc. Application Programming Interface," http://www.pgroup.com.

[3] R. Dolbeau, S. Bihan and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *GPGPU*, 2007.

[4] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff, "A hybrid approach of OpenMP for clusters," in *PPoPP*, 2012, pp. 75–84.

[5] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *PLDI*, 1993, pp. 126–138.

[6] V. Adve and J. Mellor-Crummey, "Using integer sets for data-parallel program analysis and optimization," in *PLDI*, 1998, pp. 186–198.

[7] D. Chavarría-Miranda and J. Mellor-Crummey, "Effective communication coalescing for data-parallel applications," in *PPoPP*, 2005, pp. 14–25.

[8] M. Claßen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," in *IPDPS*, 2006.

[9] U. Bondhugula, "Automatic distributed memory code generation using the polyhedral framework," Indian Institute of Science, Tech. Rep. IISc-CSA-TR-2011-3, 2011.

[10] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *PPoPP*, 2011, pp. 277–288.

[11] C. Bastoul, "Clan: The Chunky Loop Analyzer," the Clan User guide.

[12] "PolyLib - A library of polyhedral functions," http://icps.u-strasbg.fr/polylib/.

[13] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, vol. 8, pp. 102–114, Aug. 1992.

[14] S. Verdoolaege, "Integer Set Library," an integer set library for program analysis.

[15] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004, habilitation thesis.

[16] "PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores," http://pluto-compiler.sourceforge.net.

[17] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT*, 2004, pp. 7–16.

[18] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008, pp. 101–113.

[19] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *SC*, 2012, pp. 40:1–40:11.

[20] "Berkeley UPC - Unified Parallel C," http://upc.lbl.gov.

[21] "Polybench," http://polybench.sourceforge.net.

[22] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *SPAA*, 2011, pp. 117–128.

[23] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *GPGPU*, 2010, pp. 51–61.

[24] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *SPAA*, 2012, pp. 91–100.

[25] B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, A. Leung, and R. Lethin, "R-Stream Compiler," in *Encyclopedia of Parallel Computing*, 2011, pp. 1756–1765.

[26] M. Wolfe, "Implementing the PGI Accelerator model," in *GPGPU*, 2010, pp. 43–50.

[27] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *PLDI*, 2011, pp. 142–151.

[28] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for CPU-GPU architectures," in *CGO*, 2012, pp. 165–174.

[29] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for GPUs using compiler-assisted run-time coherence scheme," in *PACT*, 2012, pp. 33–42.