Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs

Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, Torsten Hoefler
ETH Zurich, Switzerland

ABSTRACT

Determining I/O lower bounds is a crucial step in obtaining communication-efficient parallel algorithms, both across the memory hierarchy and between processors. Current approaches either study specific algorithms individually, disallow programmatic motifs such as recomputation, or produce asymptotic bounds that exclude important constants. We propose a novel approach for obtaining precise I/O lower bounds on a general class of programs, which we call Simple Overlap Access Programs (SOAP). SOAP analysis covers a wide variety of algorithms, from ubiquitous computational kernels to full scientific computing applications. Using the red-blue pebble game and combinatorial methods, we are able to bound the I/O of the SOAP-induced Computational Directed Acyclic Graph (CDAG), taking into account multiple statements, input/output reuse, and optimal tiling. To deal with programs that are outside of our representation (e.g., non-injective access functions), we describe methods to approximate them with SOAP. To demonstrate our method, we analyze 38 different applications, including kernels from the Polybench benchmark suite, deep learning operators, and — for the first time — applications in unstructured physics simulations, numerical weather prediction stencil compositions, and full deep neural networks. We derive tight I/O bounds for several linear algebra kernels, such as Cholesky decomposition, improving the existing reported bounds by a factor of two. For stencil applications, we improve the existing bounds by a factor of up to 14. We implement our method as an open-source tool, which can derive lower bounds directly from provided C code.

CCS CONCEPTS

• **Theory of computation → Communication complexity**; **Parallel computing models**; *Scheduling algorithms*;

KEYWORDS

I/O complexity, red-blue pebble game, parallel scheduling model

ACM Reference Format:
Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, Torsten Hoefler. 2021. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithmsand Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3409964.3461796

SPAA '21, July 6–8, 2021, VirtualEvent, USA

ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461796

1 INTRODUCTION

I/O operations, both across the memory hierarchy and between parallel processors, dominate time and energy costs in many scientific applications [1–4]. It is thus of key importance to design algorithms with communication-avoiding or *I/O-efficient* schedules [5, 6]. To inform, and occasionally inspire the development of such algorithms, one must first understand *the associated lower bounds on the amounts of communicated data.* Deriving these bounds has always been of theoretical interest [7, 8]. It is particularly relevant for dense linear algebra, as many important problems in scientific computing [9, 10] and machine learning [11] rely on linear algebra operations such as matrix factorization [12, 13] or tensor contractions [14].

Analyzing I/O bounds of linear algebra kernels dates back to the seminal work by Hong and Kung [8], who derived the first asymptotic bound for matrix-matrix multiplication (MMM) using the red-blue pebble game abstraction. This method was subsequently extended and used by other works to derive asymptotic [15] and tight [16] bounds for more complex programs. Despite the expressiveness of pebbling, it is prohibitively hard to solve for arbitrary programs, as it is PSPACE-complete in the general case [17].

Since analyzing programs with parametric sizes disallows the construction of an explicit Computation Directed Acyclic Graph (CDAG), some form of parameterization is often needed [18–20]. However, we argue that the widely-used approaches based on the Loomis-Whitney or the HBL inequalities [21–23] (a) are often too restrictive, requiring the programs to be expressed in the polyhedral model to count the points in the projection polytopes; (b) do not capture pebbling motifs such as recomputation [19]; or (c) are limited to single-statement programs [7, 21–23, 23, 24].

In our work, we take a different approach based on a combinatorial method. We directly map each elementary computation to a vertex in a parametric CDAG, which allows us not only to operate on unstructured iteration domains, but also to precisely count the sizes of dominator sets and model vertex recomputation. Furthermore, to handle complex data dependencies in programs that evaluate multiple arrays, we introduce the Symbolic Directed Graph (SDG) abstraction, which encapsulates the data flow between elementary computations. This allows us to cover a wider class of programs and handle more complex data flow.

To enable precisely mapping every data access to the parametric CDAG vertex, we introduce a class of ***Simple Overlap Access Programs*** (SOAP), and present a general method to derive *precise* I/O bounds of programs in this class. Specifically, SOAPs are defined as loop nests of statements, whose data access sets can be modeled as injective functions, and their per-statement data overlap

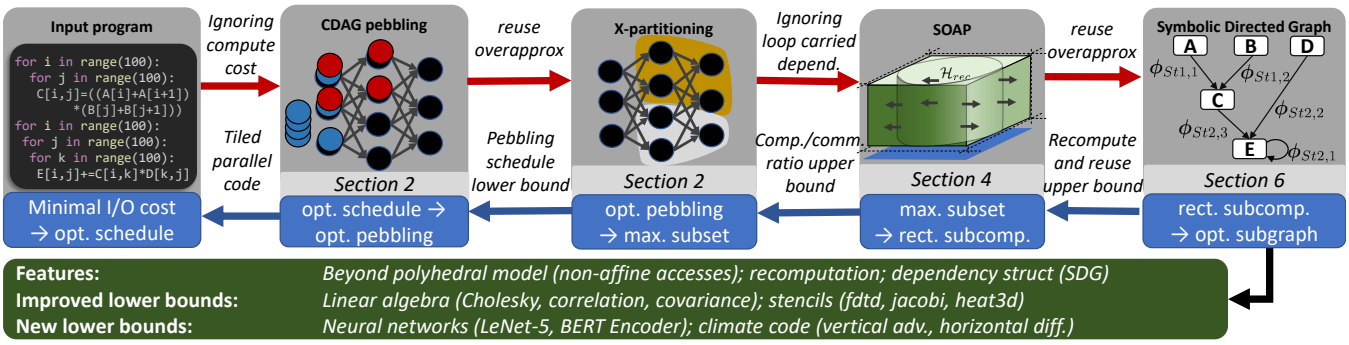

Figure 1: High level overview of the combinatorial SOAP analysis. An input program's schedule is modeled as the red-blue pebble game. The X-Partitioning abstraction relaxes the pebbling problem to the graph partition problem. The SOAP abstraction utilizes the static loop structure to upper-bound the size of the optimal X-partition. The Symbolic Directed Graph (SDG) models inter-statement data dependencies. Our method derives I/O lower bounds together with accompanying tile sizes and loop fusions that can be used by a compiler to generate an I/O optimal parallel code.

can be expressed with constant offsets. For programs that do not directly adhere to SOAP, with nontrivial overlaps and non-injective access functions, we show that under a set of assumptions, we can construct SOAP "projections" of those programs, which can be analyzed in the same way. Our method strictly contains the polyhedral model and associated analysis methods.

To show the breadth of our approach, we demonstrate SOAP analysis on a set of 38 applications, taking Python and C codes as input to create the SDG. This automated analysis procedure generates symbolic bounds, which match or improve upon previously-known results. Notably, we tighten the known I/O lower bounds for numerous programs, including stencils by up to a factor of 14, linear algebra kernels (e.g., Cholesky factorization by a factor of two), and the core convolution operation in deep learning by a factor of 8.

Since our derivation of the bounds is constructive — i.e., it provides loop tilings and fusions after relaxing loop-carried dependencies — the results can be used by a compiler to generate I/O optimal parallel codes. This can both improve existing schedules and possibly reveal new parallelization dimensions.
The paper makes the following contributions:

- A combinatorial method for precisely counting the number of data accesses in parametric CDAGs.
- A class of programs — SOAP — on which I/O lower bounds can be automatically derived.
- Symbolic dataflow analysis that extends SOAP to multiple-statement programs, capturing input and output reuse between statements, as well as data recomputation.
- I/O analysis of 38 scientific computing kernels, improving existing bounds [19, 20] by up to a factor of 14, and new lower bounds for applications in deep learning, unstructured physics simulation, and numerical weather prediction.

2 BACKGROUND

We first present several fundamental concepts used throughout the paper. We introduce program, memory, and execution models that are based on the work by Hong and Kung [8]. We then present a general approach for deriving I/O lower bounds based on graph partitioning abstractions. The bird's eye view of our method is presented in Figure 1.

2.1 General Approach of Modeling I/O Costs

Program model: CDAG. One of the most expressive ways to model executions of arbitrary programs is a Computation Directed Acyclic Graph (CDAG) [8, 16, 18, 20, 25] $G = (V, E)$, where vertices represent data (either inputs or results of computations) and edges represent data dependencies. That is, for $u, v \in V$, a directed edge $(u, v) \in E$ signifies that u is required to compute v. Given vertex v, vertices $\{u : (u, v) \in E\}$ are referred to as *parents* of v. Analogously, $\{u : (v, u) \in E\}$ are *children* of v. Vertices with in-degree (out-degree) zero are denoted *program inputs* (*program outputs*).
Memory model: red-blue pebble game [8]. Programs are executed on a sequential machine equipped with a two-level memory system, which consists of a fast memory of limited size and unlimited slow memory. The contents of the fast memory are represented by S red pebbles. A red pebble placed on a vertex indicates that the data associated with this vertex resides in the fast memory. Analogously, data residing in the slow memory is represented with blue pebbles (of unlimited number).
Execution model: graph pebbling. An execution of a program represented by a CDAG $G = (V, E)$ is modeled as a sequence of four allowed pebbling moves: 1) placing a red pebble on a vertex which has a blue pebble (load), 2) placing a blue pebble on a vertex which has a red pebble (store), 3) placing a red pebble on a vertex whose parents have red pebbles (compute) 4) removing any pebble from a vertex (discard). At the program start, all input vertices have blue pebbles placed on them. Execution finishes when all output vertices have blue pebbles on them. A sequence of moves leading from the start to the end is called a graph *pebbling* P. The number of load and store moves in P is called the *I/O cost of* P. ***The I/O cost*** Q ***of a program*** G ***is the minimum cost among all valid pebbling configurations***. A pebbling with cost Q is called optimal.

2.2 I/O Lower Bounds

Assume that the optimal pebbling P_{opt} is given. For any constant $X > S$ we can partition this sequence of moves into subsequences, such that in each subsequence except of the last one, exactly $X - S$ load/store moves are performed (the last subsequence contains at most $X - S$ load/store moves). Denote the number of these subsequences as h. Then observe that $(X - S)(h - 1) \leq Q \leq (X - S)h$.
Graph pebbling vs graph partitioning. Since finding P_{opt} is PSPACE complete [26], we seek to derive a lower bound of Q from

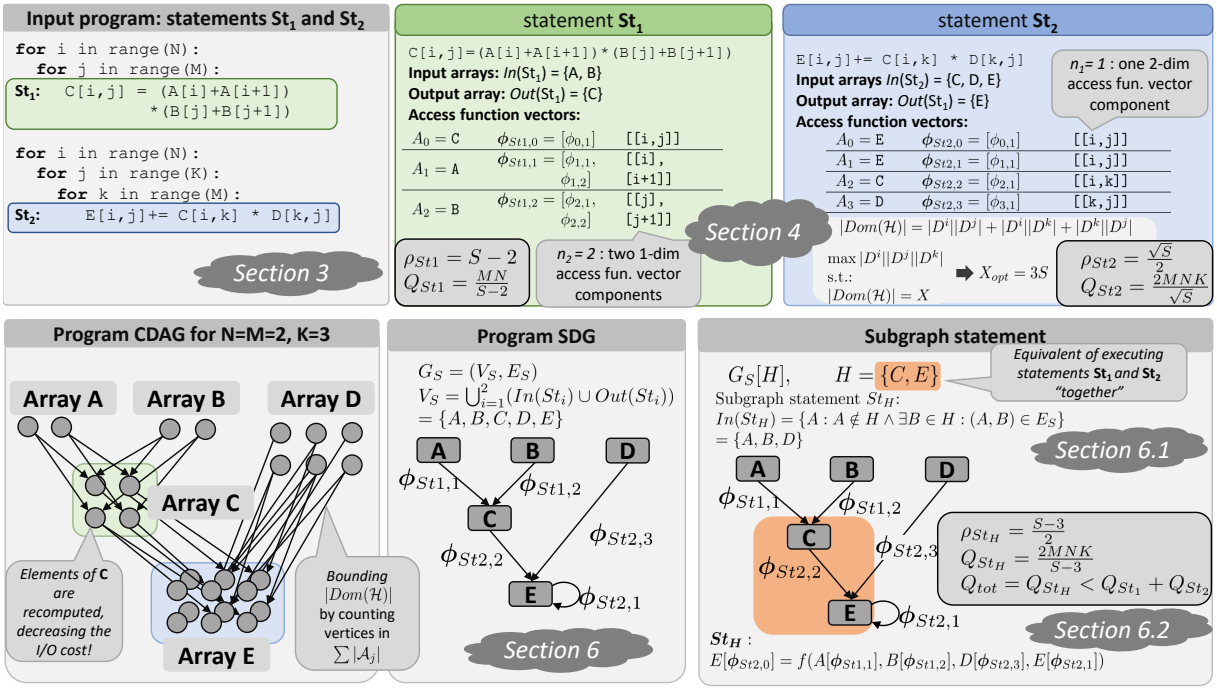

Figure 2: From the input code to the I/O lower bounds. First, for each statement, the access function vectors ϕ are extracted from the input program (green and blue fields). For each statement, the size of its dominator set is obtained using Lemma 3 (Section 4.2), and then, the I/O lower bound is obtained using inequality 9 (Section 4.5). For programs that contain multiple statements, the SDG is constructed (Section 6) and all valid subgraph statements are evaluated (Section 6.1). Lastly, the final I/O lower bound is obtained (Section 6.2).

the structure of G. Observe that the set of vertices which are computed in each subsequence defines a subgraph $\mathcal{H} \subseteq G$. By this construction, computing vertices in $\mathcal{H}$ requires $X - S$ load/store operations in the optimal schedule. The number of subsequences h may be bounded by a particular partitioning of G. To do this, we need to introduce two vertex sets defined for any subgraph of G.

Dominator and minimum sets [8]. Given $\mathcal{H} \subseteq G$, a *dominator set* $Dom(\mathcal{H})$ is a set of vertices such that every path from an input to any vertex in $\mathcal{H}$ must contain at least one vertex in $Dom(\mathcal{H})$. The *minimum set* $Min(\mathcal{H})$ is a set of all vertices in $\mathcal{H}$ that do not have any child in $\mathcal{H}$. To avoid the ambiguity of non-uniqueness of dominator set size, we denote a *minimum dominator set* $Dom_{min}(\mathcal{H})$ to be a dominator set with the smallest size.

X-Partitioning: bounding I/O cost. Introduced by Kwasniewski et al. [16], X-Partitioning generalizes the *S-partitioning* from Hong and Kung [8]. Given a constant X, an X-partition of $G = (V, E)$ is a collection of s mutually disjoint subsets $\mathcal{H}_i \subseteq V$ (referred to as *subcomputations*) $\mathcal{P}(X) = \{\mathcal{H}_1, \ldots, \mathcal{H}_s\} : \forall_{i \neq j} \mathcal{H}_i \cap \mathcal{H}_j = \emptyset \wedge \bigcup_i \mathcal{H}_i = V$ with two additional properties:

- there are no cyclic dependencies between subcomputations: $\forall \mathcal{H}_i \neq \mathcal{H}_j : (\exists (u_1, v_1) \in E \text{ s.t. } u_1 \in \mathcal{H}_i \wedge v_1 \in \mathcal{H}_j) \implies (\nexists (v_2, u_2) \in E \text{ s.t. } u_2 \in \mathcal{H}_i \wedge v_2 \in \mathcal{H}_j)$
- $\forall \mathcal{H} \in \mathcal{P}(X), |Dom_{min}(\mathcal{H})| \leq X$ and $|Min(\mathcal{H}_h)| \leq X$.

The authors prove that for any $X > S$, the optimal pebbling P_{opt} has an associated X-partition $\mathcal{P}_{opt}(X)$ s.t. $|\mathcal{P}_{opt}(X)| = h$.

Computational intensity. In previous works it was proven that (a) Q is lower bounded by the number of subsequences h in the optimal pebbling P_{opt} [8]; (b) h is lower bounded by the size of the smallest X-partition $|\mathcal{P}_{min}(X)|$ for any value of $X > S$ [16]; (c) $|\mathcal{P}_{min}(X)|$ is bounded by the maximum size of a single subcomputation $|\mathcal{H}_{X,max}|$ in any valid X-partition: $|\mathcal{P}_{min}(X)| \geq |V|/|\mathcal{H}_{X,max}|$ [16]; and (d) if $|\mathcal{H}_{X,max}|$ can be expressed as a function of X, that is, $\chi(X) \equiv |\mathcal{H}_{X,max}|$, then Q is bounded by

$$Q \geq |V| \frac{X_0 - S}{\chi(X_0)}, \tag{1}$$

where $X_0 = \arg\min_X \frac{\chi(X)}{X-S}$ (Lemma 2 in Kwasniewski et al. [27]). The expression $\rho = \frac{\chi(X)}{X-S}$ is called the *computational intensity*.

3 SIMPLE OVERLAP ACCESS PROGRAMS

In Section 2, we show how the I/O cost of a program can be bounded by the maximum size of a subcomputation $\mathcal{H}$ in any valid X-partition of program CDAG. We now introduce **Simple Overlap Access Programs (SOAP)**: a class of programs for which we can derive tight analytic bounds of $|\mathcal{H}|$. We leverage the SOAP structure and design an end-to-end method for deriving I/O lower bounds of input programs (summarized in Figure 2).

What is SOAP? Before introducing the formal definition, we start with an illustrative example, which we use in the following sections.

Example 1. *Consider the following 3-point stencil code (we use the Python syntax in code listings):*

```
for t in range(1,T):
  for i in range(t,N-t):
    A[i,t+1]=(A[i-1,t] + A[i,t] + A[i+1,t])/3 + B[i]
```

This is what we will refer to as a single-statement SOAP. The program consists of one statement St : `A[i,t+1]=(A[i,t] + ...` *which is placed in two nested loops. All accessed data comes from static, disjoint, multi-dimensional arrays (*`A` *and* `B`*). Furthermore, different accesses to the same array (array* `A` *is referenced by* `[i,t+1]`, `[i-1,t]`,

`[i,t]`,`[i+1,t]`*) are offset by a constant stride* `[0,1]`, `[-1,0]`,`[0,0]`, `[1,0]`. *We denote such access pattern as a* ***simple overlap*** *and it is a defining property of SOAP.*

Why SOAP? We use the restriction on the access pattern to precisely count the number of vertices in $Dom(\mathcal{H})$. If we allow arbitrary overlap of array accesses, we need to conservatively assume a maximum possible overlap of accessed vertices. This reduces the lower bound on $|Dom(\mathcal{H})|$, which, in turn, increases the upper bound on $|\mathcal{H}|$, providing less-tight I/O lower bound for a program.

This is not a fundamental limitation of our method. However, it allows a fully automatic derivation of tight I/O lower bounds for input programs. If the restriction is violated, additional assumptions on the access overlap are needed (Section 5).

SOAP definition. A program is a sequence of statements $St_1, \ldots, St_k$. Each such statement St is a constant time computable function f enclosed in a loop nest of the following form:

for $\psi^1 \in \mathcal{D}^1$:
  ...
  for $\psi^\ell \in \mathcal{D}^\ell(\psi^1, \ldots, \psi^{\ell-1})$:
    $St : A_0[\boldsymbol{\phi_0}(\boldsymbol{\psi})] \leftarrow f(A_1[\boldsymbol{\phi_1}(\boldsymbol{\psi})], A_2[\boldsymbol{\phi_2}(\boldsymbol{\psi})], \ldots, A_m[\boldsymbol{\phi_m}(\boldsymbol{\psi})])$

where:

(1) The *statement* St is nested in a loop nest of depth ℓ.
(2) Each loop in the tth level, $t = 1, \ldots, \ell$ is associated with its *iteration variable* ψ^t, which iterates over its domain $\mathcal{D}^t \subset \mathbb{N}$. Domain $\mathcal{D}^t$ may depend on iteration variables from outer loops $\psi^1, \ldots, \psi^{t-1}$ (denoted as $\mathcal{D}^t(\psi^1, \ldots, \psi^{t-1})$).
(3) All ℓ iteration variables form the *iteration vector* $\boldsymbol{\psi} = [\psi^1, \ldots, \psi^\ell]$ and we define the *iteration domain* $\boldsymbol{\mathcal{D}}$ as the set of all values the iteration vector iterates over during the entire execution of the program $\boldsymbol{\mathcal{D}} \subset \mathbb{N}^\ell$.
(4) The dimension of array A_j is denoted as $dim(A_j)$.
(5) Elements of A_j are referenced by an *access function vector* $\boldsymbol{\phi}_j$ which maps $dim(A_j)$ iteration variables $\boldsymbol{\psi}_j = [\psi_j^1, \ldots, \psi_j^{dim(A_j)}]$ to **a set of n_j elements** from A_j, that is $\boldsymbol{\phi}_j : \mathcal{D}_j^1 \times, \ldots, \times \mathcal{D}_j^{dim(A_j)} \rightarrow \left(\mathbb{N}^{dim(A_j)}\right)^{n_j}$. We then write $\boldsymbol{\phi}_j = [\boldsymbol{\phi}_{j,1}, \ldots, \boldsymbol{\phi}_{j,n_j}]$, where $\boldsymbol{\phi}_{j,k} : \mathcal{D}_j^1 \times, \ldots, \mathcal{D}_j^{dim(A_j)} \rightarrow \mathbb{N}^{dim(A_j)}, k = 1, \ldots, n_j$. Furthermore, all access function components $\boldsymbol{\phi}_{j,k}(\boldsymbol{\psi}_j)$ are injective.
(6) All n_j access function vector's components are equal up to a constant translation vector, that is, $\forall k = 1, \ldots, n : \boldsymbol{\phi}_{j,k}(\boldsymbol{\psi}) = \boldsymbol{\phi}_{j,1}(\boldsymbol{\psi}) + \boldsymbol{t}_k$, where $\boldsymbol{t}_k = [t_k^1, \ldots, t_k^{dim(A)}] \in \mathbb{N}^{dim(A_j)}$. We call $\boldsymbol{\phi}_j$ the **simple overlap access**.
(7) Arrays $A_1, \ldots A_m$ are disjoint. If the output array A_0 is also used as an input, that is, $A_0 \equiv A_j, j \geq 1$, then $\boldsymbol{\phi}_0 \cup \boldsymbol{\phi}_j$ is also the simple overlap access (c.f. Example 1).
(8) Each execution of statement St is an evaluation of f for a given value of iteration vector $\boldsymbol{\psi}$.

Iteration variables and iteration vectors. Formally, an iteration variable ψ^t is an iterator: an object which takes values from its iteration domain during the program execution. However, if it is clear from the context, we will refer to a particular *value* of the iteration variable simply as ψ^t (or a value of iteration vector as $\boldsymbol{\psi}$).

Vertices as iteration vectors. Since by definition of CDAG, each computation corresponds to a different vertex, and by definition of SOAP, every statement execution is associated with a single iteration vector $\boldsymbol{\psi}$, every non-input vertex in G is uniquely associated with an iteration vector $\boldsymbol{\psi}$. Input vertices are referred to by their access function vectors $u = A_j[\boldsymbol{\phi}_{j,k}(\boldsymbol{\psi})]$. **We further define CDAG edges as follows:** for every value of iteration vector $\boldsymbol{\psi}$, we add an edge from all accessed elements to the vertex associated with $\boldsymbol{\psi}$, that is: $E = \{(u, v) : u = A_j[\boldsymbol{\phi}_{j,k}(\boldsymbol{\psi})], v = \boldsymbol{\psi}, \boldsymbol{\psi} \in \boldsymbol{\mathcal{D}}\}$.

X-Partitioning on SOAP's CDAG. Recall that our objective is to bound the maximum size of any subcomputation $|\mathcal{H}|$. Given pebbling P and an associated X-partition $\mathcal{P}(X)$, *every subcomputation $\mathcal{H} \in \mathcal{P}(X)$ is therefore associated with the set of iteration vectors $\boldsymbol{\psi}$ of the vertices computed in $\mathcal{H}$*. In the following section we will derive it by counting how many non-input vertices (iteration vectors) can $\mathcal{H}$ contain by bounding its dominator set size $|Dom(\mathcal{H})|$ - again, by counting vertices corresponding to each access $A_j[\boldsymbol{\phi}_{j,k}(\boldsymbol{\psi})]$.

SOAP definition (§ 3)	A_0	Output array of statement St (may overlap with input arrays).
	A_j, $j = 1, \ldots, m$	Mutually disjoint input arrays of statement St.
	$\boldsymbol{\psi} = [\psi^1, \ldots, \psi^\ell]$	Iteration vector composed of ℓ iteration variables.
	$\mathcal{D} \subseteq \mathcal{D}^1 \times, \ldots, \times \mathcal{D}^\ell$	Iteration domain: a set of values that iteration vector $\boldsymbol{\psi}$ takes during the entire program execution.
	$\boldsymbol{\phi}_j = [\boldsymbol{\phi}_{j,1}, \ldots, \boldsymbol{\phi}_{j,n_j}]$	Access function vector that maps $dim(A_j)$ variables $[\psi_j^1, \ldots, \psi_j^{dim(A_j)}]$ to n_j elements in array A_j.
	$\boldsymbol{t}_{j,k} = [t_{j,k}^1, \ldots, t_{j,k}^{dim(A_j)}]$	Translation vector of k-th access function vector's component $\boldsymbol{\phi}_{j,k}$, that is $\boldsymbol{\phi}_{j,k} \equiv \boldsymbol{\phi}_{j,1}$, $k = 1, \ldots, n_j$
Single-statement subcomputation (§ 4)	$\mathcal{P}(X) = \{\mathcal{H}_1, \ldots, \mathcal{H}_s\}$	An X-partition of CDAG $G = (V, E)$ composed of s disjoint subcomputations.
	$\boldsymbol{D} = D^1 \times, \ldots, \times D^\ell$	Subcomputation domain: a Caresian product of ranges of ℓ iteration variables during $\mathcal{H}$.
	$\mathcal{H} \subseteq \boldsymbol{D} \subseteq V$	Subcomputation $\mathcal{H}$ uniquely defined by a set of $\lvert\mathcal{H}\rvert$ iteration vector's values $\boldsymbol{\psi} \in \boldsymbol{D}$ taken during $\mathcal{H}$. If $\mathcal{H} = \boldsymbol{D}$, we call it a *rectangular subcomputation* $\mathcal{H}_{rec}$.
	$\mathcal{A} = \boldsymbol{\phi}[\mathcal{H}]$	Access set: a set of vertices from array A that are accessed by $\boldsymbol{\phi}$ during $\mathcal{H}$.
	$\hat{\imath}^i = \{t_1^i, \ldots, t_n^i\} \setminus \{0\}$	Access offset set: set of all non-zero ith coordinates among n translation vectors $\boldsymbol{t}_k$, $k = 1, \ldots, n$.
	$Dom(\mathcal{H})$	Dominator set of subcomputation $\mathcal{H}$.
	ρ	The computational intensity of the X-partition.
	$Q \geq \lvert\mathcal{D}\rvert \frac{\sum_{j=1}^m \lvert\mathcal{A}_j(X_0)\rvert - S}{\prod_{t=1}^\ell \lvert D^t(X_0)\rvert}$	A number of I/O operations of a schedule.
SDG (§ 6)	$G_S = (V_S, E_S)$	Symbolic Directed Graph, where every array accessed in a program is a vertex, and edges represent data dependencies between them.
	$I \subset V_S$	Set of read-only arrays of the program.
	$G_S[H]$, $H \subset V_S \setminus I$	SDG subgraph that represents a subcomputation in which at least one vertex from every array in H is computed.
	St_H	Subgraph SOAP statement.

Table 1: Notation used in the paper.

4 I/O LOWER BOUNDS FOR SINGLE-STATEMENT SOAP

We now derive the I/O bounds for programs that contain only one SOAP statement. We start with introducing necessary definitions that allow us to bound the size of a *rectangular subcomputation*. The summary of the notation is presented in Table 1.

4.1 Definitions

Definition 1. ***Subcomputation domain.*** *Denote the set of all values which iteration variable ψ^t takes during subcomputation $\mathcal{H}$ as $D^t \subset \mathcal{D}^t, t = 1, \ldots, \ell$. Then, the* ***subcomputation domain***

$\boldsymbol{D}(\mathcal{H}) \subseteq \mathcal{D}$ *is a Cartesian product of ranges of all* ℓ *iteration variables which they take during* $\mathcal{H}$*, that is* $\boldsymbol{D}(\mathcal{H}) = D^1 \times, \ldots, \times D^\ell$*. We therefore have* $\mathcal{H} \subseteq \boldsymbol{D}(\mathcal{H}) \subset \mathbb{N}^\ell$*. If it is clear from the context, we will sometimes denote* $\boldsymbol{D}(\mathcal{H})$ *simply as* $\boldsymbol{D}$*.*

Example 2. *Recall the program from Example 1. Consider subcomputation* $\mathcal{H}$ *in which* $\mathtt{t} \in \{1, 2\}$ *and* $\mathtt{i} \in \{1, 2\}$*. Then, subcomputation domain* $\boldsymbol{D} = \{1, 2\} \times \{1, 2\} = \{[1, 1], [1, 2], [2, 1], [2, 2]\}$*, but computation itself can contain at most 3 elements* $\mathcal{H} \subseteq \{[1, 1], [1, 2], [2, 2]\}$*, since* $\boldsymbol{\psi} = [2, 1] \notin \mathcal{D}$ *does not belong to the iteration domain.*

Definition 2. ***Access set and access subdomain.*** *Consider input array A and its access function vector* $\boldsymbol{\phi}$*. Given* $\mathcal{H}$*, the* ***access set*** $\mathcal{A}$ *of A is the set of vertices belonging to A that are accessed during* $\mathcal{H}$*, that is* $\mathcal{A} = \boldsymbol{\phi}[\mathcal{H}] = \{A[\boldsymbol{\phi}(\boldsymbol{\psi})] : \boldsymbol{\psi} \in \mathcal{H}\}$*. If function* $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n]$ *accesses n vertices from A, we analogously define access sets for each access function component* $\boldsymbol{\phi}_k[\mathcal{H}], k = 1, \ldots, n$*. We then have* $\mathcal{A} = \bigcup_{k=1}^{n} \boldsymbol{\phi}_k[\mathcal{H}]$*. The* ***access subdomain*** $\boldsymbol{D}(\mathcal{A})$ *is minimum bounding box of the access set* $\mathcal{A}$*.*

Example 3. *For program in Example 1, consider subcomputation* $\mathcal{H}$ *evaluated on only one iteration vector* $\mathcal{H} = [i = 2, j = 2]$*. We have two accessed arrays* A *and* B*. Furthermore, we have* $\boldsymbol{\phi}_\mathsf{A} = [[i, t + 1], [i - 1, t], [i, t], [i - 1, t]]$*. Therefore,* $dim(\mathsf{A}) = 2$*, and* $\boldsymbol{\phi}_\mathsf{B} : \mathbb{N}^2 \to \left(\mathbb{N}^2\right)^4$*. We further have* $\boldsymbol{\phi}_\mathsf{B} = [[i]]$*,* $dim(\mathsf{B}) = 1$*, and* $\boldsymbol{\phi}_\mathsf{A} : \mathbb{N} \to \mathbb{N}$*. To evaluate St for* $\boldsymbol{\psi} = [2, 2]$*, we need to access four elements of* A *(three loads and one store), so its access set is* $\mathcal{A} = \boldsymbol{\phi}_\mathsf{A}[\mathcal{H}] = \{[2, 3], [1, 2], [2, 2], [2, 3]\}$*. Furthermore, we have the access subdomain* $\boldsymbol{D}(\mathcal{A}) = \{2, 3\} \times \{1, 2, 3\}$*.*

Definition 3. ***Access offset set****. Given a simple overlap access* $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n]$ *consider its n translation vectors* $\boldsymbol{t}_k = [t_k^1, \ldots, t_k^{dim(A_j)}] \in \mathbb{N}^{dim(A)}, k = 1, \ldots, n$*. For each dimension* $i = 1, \ldots, dim(A_j)$ *we denote* $\hat{t}^i = \{t_1^i, \ldots, t_n^i\} \setminus \{0\}$ *as the set of all unique non-zero* i*th coordinates among all n translation vectors.*

Definition 4. ***Rectangular subcomputation*** *For a given subcomputation domain* $\boldsymbol{D}$*, a subcomputation* $\mathcal{H}$ *is called* ***rectangular*** *if* $\mathcal{H} = \boldsymbol{D}$ *and is denoted* $\mathcal{H}_{rec}(\boldsymbol{D})$*. The size of rectangular computation is* $|\mathcal{H}_{rec}(\boldsymbol{D})| = \prod_{t=1}^{\ell} |D^t|$*. If it is clear from the context, we will denote* $\mathcal{H}_{rec}(\boldsymbol{D})$ *simply as* $\mathcal{H}_{rec}$*.*

Observation 1. *Consider a simple overlap access* $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n]$ *of array A and a rectangular subcomputation* $\mathcal{H}_{rec}(\boldsymbol{D})$*. Then since all* $\boldsymbol{\phi}_k$ *are equal up to translation, the ranges of iteration variables they access are also equal up to the same translation:* $\forall i = 1, \ldots, dim(A) : \forall j = 1, \ldots, n : \boldsymbol{\phi}_j[D^i] = \boldsymbol{\phi}_1[D^i] + t_j$*, which also implies that* $\forall i = 1, \ldots, dim(A) : \forall j = 1, \ldots, n : |\boldsymbol{\phi}_j[D^i]| = |\boldsymbol{\phi}_1[D^i]|$*.*

To bound the sizes of rectangular subcomputations, we use two lemmas given by Kwasniewski et al. [27]:

Lemma 1. *(Lemma 4 in [27]) For statement St, given* $\boldsymbol{D}$*, the size of subcomputation* $\mathcal{H}$ *(number of vertices of S computed during* $\mathcal{H}$*) is bounded by the sizes of the iteration variables' sets* $D^t, t = 1, \ldots, \ell$*:*

$$|\mathcal{H}| \leq \prod_{t=1}^{\ell} |D^t|. \tag{2}$$

Proof. Inequality 2 follows from a combinatorial argument: each computation in $\mathcal{H}$ is uniquely defined by its iteration vector $[\psi^1, \ldots, \psi^\ell]$. As each iteration variable ψ^t takes $|D^t|$ different values during $\mathcal{H}$, we have $|D^1| \cdot \cdots \cdot |D^t| = \prod_{t=1}^{\ell} |D^t|$ ways how to uniquely choose the iteration vector in $\mathcal{H}$. □

Lemma 2. *(Lemma 5 in [27]) For the given access function* $\boldsymbol{\phi} = [\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_n]$ *accessing array A,* $A[\boldsymbol{\phi}(\boldsymbol{\psi})]$*, the access set size of each of components* $|\boldsymbol{\phi}_k[\mathcal{H}]|$ *during subcomputation* $\mathcal{H}$ *is bounded by the sizes of* $dim(\boldsymbol{\phi}_A)$ *iteration variables' sets* $D^i, k = 1, \ldots, dim(\boldsymbol{\phi}_j)$*:*

$$|\boldsymbol{\phi}_k[\mathcal{H}]| \leq \prod_{i=1}^{dim(\boldsymbol{\phi}_A)} |D^i| \tag{3}$$

where $D^i \ni \psi^i$ *is the iteration domain of variable* ψ^i *during* $\mathcal{H}$*.*

Proof. We use the same combinatorial argument as in Lemma 1. Since access functions are injective, each vertex in A accessed by $\boldsymbol{\phi_k}$ is uniquely defined by $[\psi_k^1, \ldots, \psi_k^{dim(A)}]$. Knowing the number of different values each ψ_k^j takes in $\mathcal{H}$, we bound the number of different access vectors $\boldsymbol{\phi}_k[\mathcal{H}]$. □

4.2 Bounding SOAP Access Size

Recall that our goal is to find the maximum size of the subcomputation given its dominator size. We first do the converse: given the rectangular subcomputation $\mathcal{H}_{rec}$, we bound the minimum number of input vertices required to compute $\mathcal{H}_{rec}$. In Section 4.4 we prove that indeed $\mathcal{H}_{rec}$ is the subcomputation that upper-bounds the maximum computational intensity ρ. Since arrays $A_1, \ldots, A_m$ are disjoint, the total number of input vertices is the sum of their access set sizes: $|Dom_{min}(\mathcal{H}_{rec}) \geq \sum_{j=1}^{m} |\mathcal{A}_j|$. We now proceed to bound individual access set sizes $|\mathcal{A}_j|$.

Consider array A with $dim(A) = d$ and its access function $\boldsymbol{\phi}(\boldsymbol{\psi}) = [\boldsymbol{\phi}_1(\boldsymbol{\psi}), \ldots, \boldsymbol{\phi}_n(\boldsymbol{\psi})]$ that access n elements from A (to simplify the notation, we drop the subscript j, since we consider only one array). Observe that during $\mathcal{H}_{rec}$, all combinations of iteration variables $\psi^1 \in D^1, \ldots, \psi^\ell \in D^\ell$ are accessed, so $|\mathcal{H}_{rec}| = \prod_{t=1}^{\ell} |D^t|$ (Lemma 1). This also implies that each of $k = 1, \ldots, n$ accesses to A required $|\boldsymbol{\phi}_k[\mathcal{H}_{rec}]| = \prod_{t=1}^{d} |D^t|$ vertices from A (Lemma 2 and Observation 1). Therefore, the total number of accesses to array A during $\mathcal{H}_{rec}$ is $|\mathcal{A}| \geq \prod_{t=1}^{d} |D^t|$. However, the sets of vertices accessed by different $\boldsymbol{\phi}_k$ may overlap, that is, there may exist two accesses $\boldsymbol{\phi}_l$ and $\boldsymbol{\phi}_m$, for which $\boldsymbol{\phi}_l[\mathcal{H}_{rec}] \cap \boldsymbol{\phi}_m[\mathcal{H}_{rec}] \neq \emptyset$. Therefore, we also obtain the upper bound $|\mathcal{A}| \leq \sum_{j=1}^{n} \prod_{t=1}^{d} |D^t|$. We now want to narrow the gap between the upper and the lower bounds.

Lemma 3. *If a given input array A with* $dim(A) = d$ *is accessed by a simple overlap access* $\boldsymbol{\phi}(\boldsymbol{\psi}) = [\boldsymbol{\phi}_1(\boldsymbol{\psi}), \ldots, \boldsymbol{\phi}_n(\boldsymbol{\psi})]$*, its access set size* $|\mathcal{A}|$ *during rectangular computation* $\mathcal{H}_{rec}(\boldsymbol{D})$ *is bounded by*

$$|\mathcal{A}| = |\boldsymbol{\phi}[\mathcal{H}_{rec}(\boldsymbol{D})]| \geq 2 \prod_{i=1}^{d} |D^i| - \prod_{i=1}^{d} (|D^i| - |\hat{t}^i|), \tag{4}$$

where $|\hat{t}^i|$ *is the size of the access offsets set in the* i*th dimension.*

Proof. W.l.o.g., consider the first access function component $\boldsymbol{\phi}_1$ and its $\prod_{t=1}^{d} |D^t|$ accessed vertices $\boldsymbol{\phi}_1[\mathcal{H}_{rec}]$. We will lower bound the number of accesses to A from remaining $\boldsymbol{\phi}_k, k = 2, \ldots, n$, which

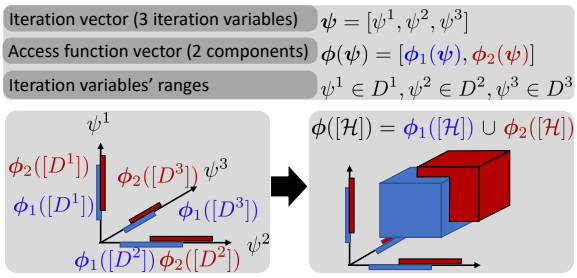

Figure 3: Intuition behind Lemma 3. Access sets $\boldsymbol{\phi}_i[\mathcal{H}_{rec}(D)]$ as 3-dimensional hyperrectangles. The union $|\bigcup_{k=1}^{n} \boldsymbol{\phi}_k[\mathcal{H}_{rec}]|$ (and therefore, the total number of accesses $|\mathcal{A}|$) is minimized when the hyperrectangles are placed in two antipodal locations of the subcomputation domain $\mathcal{D}$.

do not overlap with $\boldsymbol{\phi}_1[\mathcal{H}_{rec}]$, that is $|\bigcup_{k=2}^{n} \boldsymbol{\phi}_k[\mathcal{H}_{rec}] \setminus \boldsymbol{\phi}_1[\mathcal{H}_{rec}]|$. Since by construction of $\mathcal{H}_{rec}$, all $\boldsymbol{\phi}_k[\mathcal{H}_{rec}]$ are Cartesian products of iteration variables' ranges $\boldsymbol{\phi}_k[D^1]\times, \cdots, \times \boldsymbol{\phi}_k[D^d]$, there is a bijection between $\boldsymbol{\phi}_k[\mathcal{H}_{rec}]$ and an d-dimensional hyperrectangle $H_k \in \mathbb{N}^d$. To secure correctness of our lower bound on $|\mathcal{A}|$, we need to find the volume of the smallest union of these hyperrectangles.

Note that $|\hat{t}^i|$ is a lower bound on the maximum offset between any two $H_j \neq H_k$ in dimension i: the union of all hyperrectangles $\bigcup_{k=1}^{n} H_k$ "stretches" at least $|D^i|+|\hat{t}^i|$ elements in the ith dimension for all $i = 1, \ldots, d$ (see Figure 3). To see this, observe that since $D^i \subset N$, for each element in the access offset set $t_j^i \in \hat{t}^i$ there is at least one element in $D^i + t_j^i$ that is not in D^i, which implies that $|(D^i + t_j^i) \setminus D^i| \geq 1$. Since D^i is finite, there is a single well-defined maximum and a minimum element, which implies that $(\max\{D^i\} + t_j^i \notin D^i) \vee (\min\{D^i\} + t_j^i \notin D^i)$. Also, because by definition of $\hat{t}^i$ we have $\forall t_j^i, t_k^i \in \hat{t}^i : t_j^i \neq t_k^i$, then we also have that each t_j^i accesses at least one "non-overlapping" element independent of any other t_k^i, that is $\forall t_j^i, t_k^i \in \hat{t}^i : \max\{D^i\} + t_j^i \neq \max\{D^i\} + t_k^i$.

The arrangement of hyperrectangles $H_k, k = 1, \ldots, n$ in a $\mathbb{N}^d$ lattice s.t., their bounding box is $\boldsymbol{D} = (|D^1| + |\hat{t}^1|) \times \cdots \times (|D^d| + |\hat{t}^d|)$, which minimizes the size of their union $|\bigcup_k H_k|$ satisfies two properties:

(1) there exist two "extreme" H_p, H_q, such that $H_q = H_p + \boldsymbol{v}$, $\boldsymbol{u} = \mathbb{Z}^d, \forall_{i=1,\ldots,d} : |v^i| = |\hat{t}^i|$,
(2) all the remaining $H_k, k \neq p, q$ perfectly overlap with the "extreme" hyperrectangles $H_k \subseteq H_p \cup H_q$.

To see this, observe that for every non-zero $|\hat{t}^i|$ we need two hyperrectangles $H_p^i \neq H_q^i$, s.t., $H_q^i = H_p^i + [\cdot, \ldots, |\hat{t}^i|, \ldots, \cdot]$, that is, H_q^i is offset from H_p^i by $|\hat{t}^i|$ in ith dimension. We therefore have $\bigcup_{i,|\hat{t}^i|>0}(H_p^i \cup H_q^i) \subseteq \bigcup_k H_k$. Since H_p^i and H_q^i are pairwise non-equal, but there are no restrictions on $H_p^i, H_p^j, i \neq j$, we have that the union $\bigcup_{i,|\hat{t}^i|>0}(H_p^i \cup H_q^i)$ is minimized if $\forall_{i\neq j} H_p^i = H_p^j$.

Finally, observe the volume of $|\bigcup_{k=1}^{n} H_k|$ s.t. to the claimed arrangement is:

$$\left|\bigcup_{k=1}^{n} H_k\right| = |H_p \cup H_q| = 2\prod_{i=1}^{d}|D^i| - \prod_{i=1}^{d}(|D^i| - |\hat{t}^i|) \tag{5}$$

It shows that for any set of n hyperrectangles s.t. the given constraint, the volume of their union is no smaller than the one in Equation 5. Since the offset constraint is also a lower bound on the number of non-overlapping accesses in each dimension, it also forms the bound on $|\bigcup_{k=1}^{n} \boldsymbol{\phi}_k[\mathcal{H}_{rec}]| = |\boldsymbol{\phi}[\mathcal{H}_{rec}]| = |\mathcal{A}|$. □

4.3 Input-Output Simple Overlap

If one of the input arrays $A_i, i \geq 1$, is also the output array A_0, then their access function vectors $\boldsymbol{\phi}_0$ and $\boldsymbol{\phi}_i$ form together a simple overlap access (Section 3). In such cases, some vertices accessed by $\boldsymbol{\phi}_i$ during $\mathcal{H}_{rec}$ may be computed and do not need to be loaded. We formalize it in the following corollary, which follows directly from Lemma 3:

Corollary 1. *Consider statement St that computes array A, $dim(A) = d$ and simultaneously accesses it as an input $A[\boldsymbol{\phi}_0(\boldsymbol{\psi})] = f(A[\boldsymbol{\phi}_1(\boldsymbol{\psi})])$. If $\boldsymbol{\phi}_0 \cup \boldsymbol{\phi}_1$ is a simple overlap access, the access set size $|\mathcal{A}|$ during rectangular computation $\mathcal{H}_{rec}$ is bounded by*

$$|\mathcal{A}| \geq \prod_{i=1}^{d}|D^i| - \prod_{i=1}^{d}(|D^i| - |\hat{t}^i|), \tag{6}$$

where $\hat{t}$ is an access offset offset set of $\boldsymbol{\phi}_0 \cup \boldsymbol{\phi}_1$.

Proof. This result follows directly from Lemma 3. Since there are at least $2\prod_{i=1}^{d}|D^i| - \prod_{i=1}^{d}(|D^i| - |\hat{t}^i|)$ vertices accessed from A_i, and at most $\prod_{i=1}^{d}|D^i|$ of them can be computed during $\mathcal{H}_{rec}$ (Lemma 2) and therefore, do not have to be loaded, then at least $2\prod_{i=1}^{d}|D^i| - \prod_{i=1}^{d}(|D^i| - |\hat{t}^i|) - \prod_{i=1}^{d}|D^i|$ elements have to be accessed from the outside of $\mathcal{H}_{rec}$. □

4.4 Bounding Maximal Subcomputation

In Section 4.2 we lower-bounded the dominator set size of the rectangular subcomputation $|Dom_{min}(\mathcal{H}_{rec})| = \sum_{j=1}^{m}|\mathcal{A}_j|$ by bounding the sizes of simple overlap access sets sizes $|\mathcal{A}_j|$ (Lemma 3). Recall that to bound the I/O lower bound we need the size $\chi(X)$ of the *maximal* subcomputation $\mathcal{H}_{max}$ for given value of X (Inequality 1). We now prove that $\mathcal{H}_{rec}$ upper-bounds the size of $\mathcal{H}_{max}$.

Given $\mathcal{H}$, denote the the ratio of the size of the subcomputation to the dominator set size $\delta(\mathcal{H}) = \frac{|\mathcal{H}|}{\sum_{j=1}^{m}|\boldsymbol{\phi}_j[\mathcal{H}]|}$. By definition, $\mathcal{H}_{max}$ maximizes δ among all valid $\mathcal{H} \in \mathcal{P}$. We need to show that for a fixed subcomputation domain $\boldsymbol{D}_0$, among all subcomputations for which $\boldsymbol{D}(\mathcal{H}) = \boldsymbol{D}_0$, the rectangular subcomputation $\mathcal{H}_{rec}(\boldsymbol{D}_0)$ upper-bounds δ. *Note that an X-partition derived from the optimal pebbling schedule P_{opt} may not include $\mathcal{H}_{rec}$.* However, $\forall X : \chi_{rec}(X) \geq \chi(X)$, that is, given X, the size of $\mathcal{H}_{rec}$ s.t., $\sum_{j=1}^{m}|\boldsymbol{\phi}_j[\mathcal{H}_{rec}]| = X$ will always be no smaller than the size of $\mathcal{H}_{max}$. To show this, we first need to introduce some auxiliary definitions.

Iteration variables, their indices, and their values. To simplify the notation, throughout the paper we used the iteration variables ψ^i and the *values* they take for some iteration interchangeably. However, now we need to make this distinction explicit. The iteration vector consists of ℓ iteration variables $\boldsymbol{\psi} = [\psi^1, \ldots, \psi^\ell]$. Each access function $\boldsymbol{\phi}_j$ is defined on $dim(A_j) \leq \ell$ of them. Recall that $\boldsymbol{\psi}_j$ is the set of iteration variables accessed by $\boldsymbol{\phi}_j$ (Section 3, property (5)). To keep track of the indices of particular iteration variables, denote $\boldsymbol{\Psi} = [\ell] = \{1, \ldots, \ell\} \subset \mathbb{N}$, $\boldsymbol{\Psi}_j \subseteq \boldsymbol{\Psi}$, and $\boldsymbol{\Psi}'_j = \boldsymbol{\Psi} \setminus \boldsymbol{\Psi}_j$ as the sets of integers. If $i \in \boldsymbol{\Psi}_j$, then the ith

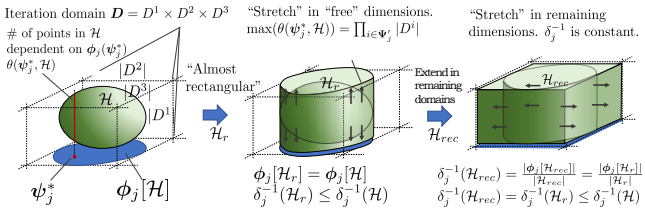

Figure 4: Intuition behind Lemma 4: extending the subcomputation in the free dimensions w.r.t $\boldsymbol{\phi}_j$ does not increase $|\boldsymbol{\phi}_j[\mathcal{H}]|$. Once the subcomputation is almost rectangular, extending H in the remaining dimensions keeps the ratio δ_j^{-1} constant.

iteration variable ψ^i is accessed by the access function $\boldsymbol{\phi}_j$. We further define $\boldsymbol{\psi}^* \in \mathbb{N}^\ell$ as a specific *value* of the iteration vector $\boldsymbol{\psi}$ that uniquely defines a single non-input vertex. We analogously define $\boldsymbol{\psi}_j^*$, $\psi^{i,*}$, and $\psi_j^{i,*}$ (the last one being a value of ith iteration variable of the jth access). We also define $\theta(\boldsymbol{\psi}_j^*, \mathcal{H})$ as the number of vertices in $\mathcal{H}$ that have all their $\boldsymbol{\Psi}_j$ coordinates equal to $\boldsymbol{\psi}_j^*$, that is $\theta(\boldsymbol{\psi}_j^*, \mathcal{H}) = |\{\boldsymbol{\psi}^* : \boldsymbol{\psi}^* \in \mathcal{H} \wedge (\forall i \in \boldsymbol{\Psi}_j : \psi^{i,*} = \psi_j^{i,*})\}|$.

We now formalize our claim in the following lemma:

Lemma 4. *Given the subcomputation domain $\boldsymbol{D}_0$, $\mathcal{H}_{rec}(\boldsymbol{D}_0)$ maximizes $\delta(\mathcal{H})$ for all $\mathcal{H}$ s.t. $\boldsymbol{D}(\mathcal{H}) = \boldsymbol{D}_0$.*

$$\forall \mathcal{H} : \delta(\mathcal{H}) \leq \delta(\mathcal{H}_{rec}) \tag{7}$$

Proof. Instead of maximizing $\delta(\mathcal{H})$, we will minimize $\delta^{-1}(\mathcal{H}) = (\sum_{j=1}^m |\boldsymbol{\phi}_j[\mathcal{H}]|)/|\mathcal{H}| = \sum_{j=1}^m |\boldsymbol{\phi}_j[\mathcal{H}]|/|\mathcal{H}|$ over all possible $\mathcal{H}$. Observe that $\delta^{-1}(\mathcal{H})$ is linear w.r.t. the ratios of individual access function sets sizes $|\boldsymbol{\phi}_j[\mathcal{H}]|$ and the size of subcomputation $|\mathcal{H}|$. Therefore, we can examine each access $\boldsymbol{\phi}_j[\mathcal{H}]$ separately and show that every $\delta_j^{-1} = |\boldsymbol{\phi}_j[\mathcal{H}]|/|\mathcal{H}|$ is minimized for $\mathcal{H} = \mathcal{H}_{rec}$. Then, if $\mathcal{H}_{rec}$ minimizes each of δ_j^{-1}, then $\delta^{-1} = \sum_{j=1}^m \delta_j^{-1}$ is minimized, so indeed $\mathcal{H}_{rec}$ maximizes the ratio of the subcomputation size to the dominator set size.

Observe now, that for any $\mathcal{H}$ we have that $\forall_j : \delta_j^{-1}$ is monotonically decreasing w.r.t. $\theta(\boldsymbol{\psi}_j^*, \mathcal{H})$ for all $\boldsymbol{\psi}_j^* \in \boldsymbol{\phi}_j[\mathcal{H}]$. That is - pick any input vertex $\boldsymbol{\psi}_j^*$ from the set of vertices accessed by $\boldsymbol{\phi}_j[\mathcal{H}]$. Adding compute vertices $\boldsymbol{\psi}^*$ to $\mathcal{H}$ that access $\boldsymbol{\psi}_j^*$ do not increase the access set size $\boldsymbol{\phi}_j[\mathcal{H}]$, since $\boldsymbol{\psi}_j^*$ is already accessed. However, it increases the size of $\mathcal{H}$. Clearly, δ_j^{-1} reaches its minimum if $\forall \boldsymbol{\psi}_j^* \in \boldsymbol{\phi}_j[\mathcal{H}] : \theta(\boldsymbol{\psi}_j^*, \mathcal{H}) = \prod_{i \in \boldsymbol{\Psi}_j'} |D^i|$, that is, $\mathcal{H}$ computes all vertices spanned by the access set $\boldsymbol{\phi}_j[\mathcal{H}]$ and all elements in the Cartesian product of "free" (independent of the access function ϕ_j) iteration domains $D^i, i \in \boldsymbol{\Psi}_j'$.

We showed that for all j, given its initial access set $\boldsymbol{\phi}_j[\mathcal{H}]$, the ratio δ_j^{-1} is minimized for the "almost-rectangular" subcomputation, that is, $\mathcal{H}$ which computes all vertices $\boldsymbol{\psi}^* \in \boldsymbol{\phi}_j[\mathcal{H}] \times \prod_{i \in \boldsymbol{\Psi}_j'} D^i$. We now need to show that also extending $\mathcal{H}$ over the "dependent" ranges $\boldsymbol{\Psi}_j$ won't increase the ratio δ^{-1}. When the access set size $\boldsymbol{\phi}_j[\mathcal{H}]$ increases by a factor x, $\mathcal{H}$ increases proportionally by x too, keeping the ratio constant (See Figure 4 for an example for $\ell = 3$).

Since our goal is to minimize each δ_j^{-1} *separately*, independently of other $\delta_i^{-1}, i \neq j$, assume that we have already extended $\mathcal{H}$ to the "almost-rectangular" subcomputation, that is, all combinations of $\prod_{i \in \boldsymbol{\Psi}_j'} D^i$ were accessed in $\mathcal{H}$. Observe now that $\theta(\boldsymbol{\psi}_j^*, \mathcal{H}) = \prod_{i \in \boldsymbol{\Psi}_j'} |D^i|$ for *any* vertex $\boldsymbol{\psi}_j^*$. Therefore, since $|\mathcal{H}| = \sum_{\boldsymbol{\psi}_j^* \in \boldsymbol{\phi}_j[\mathcal{H}]} \prod_{i \in \boldsymbol{\Psi}_j'} |D^i|$, we see that δ_j^{-1} is *constant* w.r.t., the size of the access set: $\delta_j^{-1} = \frac{|\boldsymbol{\phi}_j[\mathcal{H}]|}{|\mathcal{H}|} = \frac{1}{\prod_{i \in \boldsymbol{\Psi}_j'} |D^i|}$. Therefore, we can safely maximize $\boldsymbol{\phi}_j[\mathcal{H}]$ to the entire access set of the rectangular subcomputation $\mathcal{H}_{rec}$ without increasing δ_j^{-1}. We conclude that for every access function $\boldsymbol{\phi}_j$ and every iteration variable index i, evaluating all vertices $\boldsymbol{\psi}*$ s.t. ψ^i iterates over the entire domain D^i minimizes δ_j^{-1}.

□

4.5 I/O Lower Bounds and Optimal Tiling

We now proceed to the final step of finding the I/O lower bound. Recall from Section 2.2, that the last missing piece is $\chi(X)$; that is, we seek to express $|\mathcal{H}_{max}(\boldsymbol{D})| = \prod_{t=1}^\ell |D^t|$ as a function of X. Observe that by Lemma 4, $|Dom_{min}(\mathcal{H}_{max}(\boldsymbol{D})) \geq \sum_{j=1}^m (2 \prod_{i=1}^{dim(A_j)} |D_j^i| - \prod_{i=1}^{dim(A_j)} (|D_j^i| - |\hat{t}_j^i|)$. On the other hand, by definition of X-Partitioning, $|Dom_{min}(\mathcal{H}_{max}(\boldsymbol{D}))| \leq X$. Combining these inequalities, we solve for all $|D^t|$ as functions of X by formulating it as the optimization problem (see Section 3.2 in Kwasniewski et al. [27]):

$$\begin{aligned} &\max \prod_{t=1}^{\ell} |D^t| \qquad \text{s.t.} \\ &\sum_{j=1}^{m} |\mathcal{A}_j| \leq X \\ &\forall 1 \geq t \geq \ell : |D^t| \geq 1 \end{aligned} \tag{8}$$

Solving the above optimization problem yields $\chi(X) = |\mathcal{H}_{max}(\boldsymbol{D})|$. Since Lemma 4 gives a valid upper bound on computational intensity for *any* value of X, we seek to find the tightest (lowest) upper bound. One can obtain $X_0 = \arg\min_X \frac{\chi(X)}{X - S}$, since $\chi(X)$ is differentiable. Finally, combining Lemma 3, inequality 1, and the optimization problem 8, we obtain the I/O lower bound for the single-statement SOAP program:

$$Q \geq |\mathcal{D}| \frac{\sum_{j=1}^m |\mathcal{A}_j(X_0)| - S}{\prod_{t=1}^\ell |D^t(X_0)|}, \tag{9}$$

where $|\mathcal{A}_j(X_0)|$ are the access set sizes obtained from Lemma 3 for the optimal value of $|D^i|$ derived from the optimization problem 8.

Substituting X_0 back to $|D^t|(X)$ has a direct interpretation: they constitute optimal loop tilings for the maximal subcomputation. Note that such tiling might be invalid due to problem relaxations: e.g., we ignore loop-carried dependencies and we solve optimization problem 8 over real numbers, relaxing the integer constraint on $|D^t|$ set sizes. *However, this result can serve as a powerful guideline in code generation. Furthermore, if derived tiling sizes generate a valid code, it is provably I/O optimal.*

5 PROJECTING PROGRAMS ONTO SOAP

By the definition of SOAP, one input array may be accessed by different access function vector components, only if they form the simple overlap access — that is, the accesses are offset by a constant stride. However, our analysis may go beyond this constraint if additional assumptions are met.

5.1 Non-Overlapping Access Sets

Given input array A and its access function components $\boldsymbol{\phi}(\boldsymbol{\psi}) = [\boldsymbol{\phi}_1(\boldsymbol{\psi}_1), \ldots, \boldsymbol{\phi}_n(\boldsymbol{\psi}_n)]$, if all access sets are disjoint, that is: $\forall_{i \neq j} \boldsymbol{\phi}_i[\mathcal{D}] \cap \boldsymbol{\phi}_j[\mathcal{D}] = \emptyset$, then we represent it as n disjoint input arrays A_i accessed by single corresponding access function component $\boldsymbol{\phi}_i(\boldsymbol{\psi}_i)$.

Example 4. *Consider the following code fragment from LU decomposition:*

```
for k in range(N):
  for i in range(k+1,N):
    for j in range(k+1,N):
St:     A[i,j] = A[i,j] - A[i,k] * A[k,j]
```

The analysis of iteration variables' domains $\mathcal{D}^i, \mathcal{D}^j, \mathcal{D}^k$ shows that for fixed value of k_0, there are no two iteration vectors $\boldsymbol{\psi}_1 = [k_0, i_1, j_1]$ and $\boldsymbol{\psi}_2 = [k_0, i_2, j_2]$ such that $[i_1, k_0] = [k_0, j_2] \vee [i_1, j_1] = [k_0, j_2] \vee [i_1, j_1] = [i_1, k_0]$, therefore, their access sets are disjoint. Furthermore, for k_0, all elements from A in range $[(k_0, N), (k_0, N)]$ are updated. Therefore, all accesses of form $[i_1, k_1] = [k_2, j_2]$ access different *vertices. We model this as a SOAP statement with three disjoint arrays:*

$$St_2 : A_1[i,j] = f(A_1[i,j], A_2[i,k], A_3[k,j])$$

5.2 Equivalent Input-Output Accesses

If array A is updated by statement St — i.e., it is both input and output — then we require that the output access function $\boldsymbol{\phi}_0$ is different than the input access function $\boldsymbol{\phi}_i$. If the input program does not meet this requirement, we can add additional "version dimension" to access functions that is offset by a constant between input and output accesses.

Example 5. *Consider again Example 4. Observe that array A_1 is updated (it is both the input and the output of St_2. Furthermore, both access functions are equal: $\boldsymbol{\phi}_0 = \boldsymbol{\phi}_1 = [i, j]$. We can associate a unique version (and therefore, a vertex) of each element of A with a corresponding iteration of the k loop. We add the version dimension associated with k and offset it by constant 1 between input and output:*

$$St_3 : A_1[i,j,k+1] = f(A_1[i,j,k], A_2[i,k], A_3[k,j])$$

5.3 Non-Injective Access Functions

Given input array A and its access function vector $\boldsymbol{\phi}$, we require that $\forall \boldsymbol{\psi}_i \neq \boldsymbol{\psi}_j : A[\boldsymbol{\phi}(\boldsymbol{\psi}_i)] \neq A[\boldsymbol{\phi}(\boldsymbol{\psi}_j)]$. If this is not the case, then we seek to bound the size of such overlap, that is, given subcomputation domain $\boldsymbol{D}(\mathcal{H})$, how many different iteration vectors $\boldsymbol{\psi}_j$ map to the same array element $A[\boldsymbol{\phi}(\boldsymbol{\psi}_i)]$. We can solve this by analyzing the iteration domain $\mathcal{D}$ and the access function vector $\boldsymbol{\phi}$. If one array dimension is accessed by a function of multiple iteration variables $g(\phi^1, \ldots, \phi^k)$ and g is linear w.r.t. all ϕ^i, the number of different values g takes in $\boldsymbol{D}(\mathcal{H})$ is bounded by $\max_{i=1,\ldots,k}\{|D^i|\} \leq |g[\mathcal{H}]| \leq \prod_{i=1}^{k} |D^i|$, for $D^i \neq \{0\}, i = 1, \ldots, k$.

Example 6. *A single layer of the direct convolution used in neural networks may be written as seven nested loops with iteration variables b, c, k, w, h, r, s and statement (c.f. [23]):*

$$St : Out[k,h,w,b] += Image[r + \sigma_w w, s + \sigma_h h, c, b] \times Filter[k,r,s]$$

Depending on the value of σ_w and σ_h, the access function of Image, $\boldsymbol{\phi} = [r + \sigma_w w, s + \sigma_h h, c, b]$ may not be injective. Yet, observe that:

(1) $\sigma_w \geq |D^r| \wedge \sigma_h \geq |D^s| \implies \boldsymbol{\phi}$ is injective $\implies |\boldsymbol{\phi}[\mathcal{H}_{max}]| \geq |D^r| \cdot |D^w| \cdot |D^s| \cdot |D^h| \cdot |D^c| \cdot |D^b|$

(2) $\sigma_w = 1 \wedge \sigma_h = 1 \implies |\boldsymbol{\phi}[\mathcal{H}_{max}]| \geq \max(|D^r|, |D^w|) \cdot \max(|D^s|, |D^h|) \cdot |D^c| \cdot |D^b|$,

Our analysis provides a conditional *computational intensity: $\rho_{min} = \sqrt{S}/2$ in case (1) and $\rho_{max} = S/2$ in case (2). Observe that case (2) yields the maximum non-injective overlap (maximum number of different iteration vectors map to the same element in Image). For any other values of σ_w and σ_h, we have $\rho_{min} \leq \rho \leq \rho_{max}$.*

6 MULTI-STATEMENT SOAP

I/O lower bounds are not composable: the I/O cost of a program containing multiple statements may be lower than the sum of the I/O costs of each statement if evaluated in isolation. Data may be reused and merging of statements may lower the I/O cost.

Note that the number of vertices in the program's CDAG G depend on domain sizes D^i of each iteration variable. However, our derived upper bound of the computational intensity ρ is *independent* of the CDAG size, as it depends only on the access functions $\boldsymbol{\phi}_j$. This is also true for programs that contain multiple statements - to bound ρ for multi-statement SOAP, we only need to model dependencies between the arrays and how they are accessed - e.g., one statement may take as an input an array that is an output of a different statement.

We represent the data flow between the program statements with a *symbolic* directed graph $G_S = (V_S, E_S)$. For a given statement St_i, denote $In(St_i) = \{A_{i,1}, \ldots, A_{i,m}\}$ a set of input arrays of statement St_i. Analogously, denote $Out(St_i)$ the set containing the output array of St_i. Analogously to program CDAG G that captured dependencies between particular array elements, G_S models dependencies between whole arrays (Figure 2).

Definition 5. ***Symbolic Digraph: SDG*** *Given k-statement SOAP $St_1, \ldots, St_k$, its* symbolic digraph (SDG) *$G_S = (V_S, E_S)$ is a directed graph where $V_S = \bigcup_{i=1}^{k} (In(St_i) \cup Out(St_i))$ and $(A_u, A_v) \in E_S \iff \exists St_i : A_u \in In(St_i) \wedge A_v \in Out(St_i)$.*

G_S is a directed graph, where vertices represent arrays accessed by a program, and edges represent data dependencies between them. Two arrays A_u and A_v are connected if there is a statement that accesses A_u and computes A_v. Each edge is annotated with the corresponding access function vector of the statement that generates it.

Example 7. *Consider the example in Figure 2. We have two statements St_1 and St_2, with $In(St_1) = \{A, B\}$, $Out(St_1) = \{C\}$, $In(St_2) = \{C, D, E\}$, $Out(St_2) = \{E\}$. We then construct the SDG $G_S = (V_S, E_S)$, with $V_S = In(St_1) \cup Out(St_1) \cup In(St_2) \cup Out(St_2) = \{A, B, C, D, E\}$. Furthermore, we have edges $E_S = \{(A, C), (B, C), (C, E), (D, E), (E, E)\}$. The edges are annotated with the corresponding access function vectors $\boldsymbol{\phi}_{St1,1}, \ldots, \boldsymbol{\phi}_{St2,3}$.*

Note: While the "explicit" program CDAG $G = (V, E)$, where every vertex represents a single computation is indeed acyclic, the SDG $G_S = (V_S, E_S)$ may contain self-edges when a statement updates the loaded array ((E, E) in the example above). In G, one vertex corresponds to *one version* of a single array element, while in G_S, one vertex encapsulates *all versions* of all array elements.

6.1 SDG Subgraphs

Denote $I \subset V_S$ set of input vertices of G_S ($\forall A \in I : \text{indegree}(A) = 0$). Let $H \subset V_S \setminus I$ be a subset of the vertices of SDG $G_S = (V_S, E_S)$. The SDG subgraph $G_S[H]$ is a subgraph of G_S induced by the vertex set H. It corresponds to some subcomputation in which at least one vertex from each array in H was computed. We now use the analogous strategy to the X-Partitioning abstraction: since the optimal pebbling has an associated X-partition with certain properties (the dominator set constraint), we bound the cost of any pebbling by finding the maximum subcomputation among *all* valid X-partitions. We now show that every subcomputation in the optimal X-partition has a corresponding SDG subgraph $G_S[H]$. Therefore, finding $G_S[H_{opt}]$ that maximizes the computational intensity among *all* subgraphs bounds the size of the maximal subcomputation (which, in turn, bounds the I/O cost of any pebbling).

Recall that an optimal pebbling P has an associated X-partition $\mathcal{P}(X)$, where each $\mathcal{H} \in \mathcal{P}(X)$ represents a sequence of operations that are not interleaved with other subcomputations. Given G_S, each $\mathcal{H} \in \mathcal{P}(X)$ has an associated subgraph $G_S[H]$ s.t. every array vertex $A_i \in H$ represents an array from which at least one vertex was computed in $\mathcal{H}$.

Note that both the pebbling P and the partition $\mathcal{P}(X)$ depend on the size of the CDAG that is determined by the sizes of the iteration domains D^i. However, the SDG does not depend on them. Thus, by finding the subgraph that maximizes the computational intensity, we bound ρ for *any* combination of input parameters.

Definition 6. *The **subgraph SOAP statement** St_H of subgraph $G_S[H]$ is a single SOAP statement with the input $In(St_H) = \{A : A \notin H \wedge \exists B \in H : (A, B) \in E_S\}$. Additionally, for each vertex $B \in H$ that is not computed in H, that is $\nexists A \in H : (A, B) \in E_S$, self-edges $(B, B) \in E$ are preserved ($B \in In(St_H)$).*

Intuition. The subgraph statement St_H is a "virtual" SOAP statement that encapsulates multiple statements $St_1, \ldots, St_k$. Given H, its subgraph statement's inputs $In(St_H)$ are formed by merging inputs $\bigcup_{i=1}^{k} In(St_i) \setminus V(H)$ from all statements that form H, but are not in H. By the construction of the SDG, this is equivalent to the definition above: take all vertices $A \in V_s \setminus V(H)$ that have a child in $V(H)$, that is $\exists B \in V(H) : (A, B) \in E_S$ (see Figure 2).

This forms the lower bound on the number of inputs for a corresponding subcomputation $\mathcal{H}$: all the vertices from arrays $A_i \in V(H)$ could potentially be computed during $\mathcal{H}$ and do not need to be loaded, but at least vertices from arrays $In(St_H)$ have to be accessed.

Example 8. *Consider again the example from Figure 2. The set of input nodes is $I = \{A, B, D\}$. There are three possible subgraph statements: $H_1 = \{C\}$, with $In(St_{H_1}) = \{A, B\}$, $H_2 = \{C\}$ with $In(St_{H_2}) = \{C, D, E\}$, and $H_3 = \{C, E\}$ with $In(St_{H_3}) = \{A, B, D\}$. Note that by definition, the self-edge (C, C) is preserved in H_2, but not in H_3. Subgraphs H_1 and H_2 correspond to the input statements St_1 and St_2. Subgraph H_3 encapsulates a subcomputation $\mathcal{H}$ that computes some vertices from both arrays C and E, merging subcomputations St_1 and St_2 and reusing outputs from St_1 to compute E.*

Then, we establish the following lemma:

Lemma 5. *Given an X-partition $\mathcal{P}(X) = \{\mathcal{H}_1, \ldots, \mathcal{H}_s\}$ of the k-statement SOAP, with its corresponding $G_S = (V_S, E_S)$, each subcomputation $\mathcal{H}$ has an associated intensity $\rho_{\mathcal{H}} = \frac{|\mathcal{H}|}{|Dom_{min}(\mathcal{H})|-S}$ that is upper-bounded by the computational intensity of the subgraph statement St_H (Lemma 4).*

Proof. Recall that given the subcomputation $\mathcal{H}$, its corresponding SDG subgraph H is constructed as follows: for each vertex $v \in V$ computed during $\mathcal{H}$ belonging to some array A_i, add the corresponding array vertex s_i to H. Note that we allow a vertex recomputation: if some vertex is (re)computed during the optimal schedule of $\mathcal{H}$, its array vertex will belong to H.

Observe that by this construction and by definition of the subgraph statement, all arrays from which at least one vertex is loaded during $\mathcal{H}$ are in $In(St_H)$. Furthermore, $In(St_H)$ is a subset of these arrays: during $\mathcal{H}$, there might be some loaded vertex from array $A_j \in H$, but, by definition of St_H, this array will not be in $In(St_H)$. Therefore, St_H lower bounds the input size of $\mathcal{H}$.

The last step of the proof is to observe that by Lemma 4, the computational intensity of St_H bounds the maximum number of computed vertices for any $\mathcal{H}' \in \mathcal{P}(X)$ that belong to H, that is, the union of all arrays in H. But since all vertices that are computed in $\mathcal{H}$ belong to one of these arrays, $\mathcal{H}$ cannot have higher computational intensity. □

6.2 SDG I/O Lower Bounds

We now proceed to establish a method to derive the I/O lower bounds of the multi-statement SOAP given its SDG $G_S = (V_S, E_S)$.

For each array vertex $A \in V_S$, denote $|A|$ as the total number of vertices in the CDAG that belong to array A. Denote further $\mathcal{S}(A)$ the set of all subgraphs of G_S that contain A. Then we prove the following theorem:

Theorem 1. *The I/O cost Q of a k-statement SOAP represented by the SDG $G_S = (V_S, E_S)$ is bounded by*

$$Q \geq \sum_{A \in V_S} \frac{|A|}{\max_{H \in \mathcal{S}(A)} \rho_H} \tag{10}$$

where $\max_{H \in \mathcal{S}(A)} \rho_H$ is the maximum computational intensity over all subgraph statements of subgraphs H that contain vertex A.

Proof. This theorem is a direct consequence of Lemma 5 and the fact that all vertices in CDAG G are associated with some array vertex in SDG G_S. Lemma 5, together with the definition of $\mathcal{S}(a)$, states that $\max_{H \in \mathcal{S}(a)} \rho_H$ is the upper bound on any subcomputation $\mathcal{H}$ that contains any vertex from array a. Since there are $|a|$ vertices associated with a, at least $\frac{|a|}{\max_{H \in \mathcal{S}(a)} \rho_H}$ I/O operations must be performed to compute these vertices. Since the computational intensity expresses the average cost *per vertex*, even if some subcomputation in an optimal X-partition spans more than one array, this is already modeled by the set $\mathcal{S}(a)$. Therefore, we can sum the I/O costs per arrays a, yielding inequality 10. □

Note that applying Theorem 1 requires iterating over all possible subgraphs. In the worst case, this yields exponential complexity, prohibiting scaling our method to large programs. However, many scientific applications contain a limited number of kernels with simple dependencies. In practice we observed that our approach scales well to programs containing up to 35 statements.

7 EVALUATION

We evaluate our lower bound analysis on a wide range of applications, ranging from fundamental computational kernels and solvers to full workloads in hydrodynamics, numerical weather prediction, and deep learning. The set of applications covers both the previously analyzed kernels (the Polybench suite [28], direct convolution), and kernels that were never analyzed before due to complicated dependency structures (multiple NN layers, diffusion, advection). Not only our tool covers broader class of programs than state-of-the-art approaches, but also it improves bounds generated by methods dedicated to specific narrower classes [19]. Improving I/O lower bounds has not only theoretical implications: loose bounds may not be applicable for generating corresponding parallel codes, as too many overapproximations may yield an invalid schedule.

In our experiments we use DaCe [29] to extract SOAP statements from Python and C code, and use MATLAB for symbolic analysis.

Polybench. As our first case study, we analyze Polybench [28], a polyhedral application benchmark suite composed of 30 programs from several domains, including linear algebra kernels, linear solvers, data mining, and computational biology. Prior best results were obtained by IOLB [19], a tool specifically designed for analyzing I/O lower bounds of affine programs. We summarize the results in Table 2, listing the leading order term for brevity.

We find that SOAP analysis derives tight I/O lower bounds for all Polybench kernels. Analyzing these programs as multi-statement SOAP either reproduces existing tight bounds, or improves them by constant factors (e.g., in Cholesky decomposition) on 14 out of 30 applications (Table 2). Of particular note is `adi` (Alternating Direction Implicit solver). Our algorithm detected a possible tiling in the time dimension, yielding the lower bound $(12N^2T)/\sqrt{S}$, compared to N^2T reported by Olivry et al. [19]. However, due to dependency chains incurred by alternating directions, such tiling may violate loop-carried dependency constraints, which our algorithm relaxes. A parallel machine could potentially take advantage of this tiling scheme, possibly providing super-linear communication reduction. However, this is outside of the scope of this paper.

Neural Networks. Analyzing I/O lower bounds of neural networks is a nascent field, and so far only single-layer convolution was analyzed [20, 23]. We improve the previously-reported bound reported by Zhang et al. [20] by a factor of 8.

7.1 New Lower Bounds

Analyzing SOAP and the SDG representation enables capturing complex data dependencies in programs with a large number of statements. To demonstrate this, we study larger programs in three fields, where no previous I/O bounds are known. If an application contains both SOAP and data-dependent kernels, we find a SOAP representation that bounds the access sizes from below.

	Kernel	SOAP I/O Bound	Improv. over SotA
Polybench[19]	adi	$\frac{12N^2T}{\sqrt{S}}$	$\frac{12}{\sqrt{S}}$
	atax	MN	1
	bicg	MN	1
	cholesky	$\frac{N^3}{3\sqrt{S}}$	2
	correlation	$\frac{M^2N}{\sqrt{S}}$	2
	covariance	$\frac{M^2N}{\sqrt{S}}$	2
	deriche	$3HW$	3
	doitgen	$\frac{2N_P^2N_QN_R}{\sqrt{S}}$	1
	durbin	$\frac{3N^2}{2}$	3
	fdtd2d	$\frac{2\sqrt{3}N_XN_YT}{\sqrt{S}}$	$6\sqrt{6}$
	floyd-warshall	$\frac{2N^3}{\sqrt{S}}$	2
	gemm	$\frac{2N^2}{\sqrt{S}}$	1
	gemver	N^2	1
	gesummv	$2N^2$	1
	gramschmidt	$\frac{MN^2}{\sqrt{S}}$	1
	heat3d	$\frac{6N^3T}{\sqrt[3]{S}}$	$\frac{32}{3\sqrt[3]{3}}$
	jacobi1d	$\frac{2NT}{S}$	8
	jacobi2d	$\frac{4N^2T}{\sqrt{S}}$	$6\sqrt{3}$
	2mm	$\frac{4N^3}{\sqrt{S}}$	1
	3mm	$\frac{6N^3}{\sqrt{S}}$	1
	lu	$\frac{2N^3}{3\sqrt{S}}$	1
	ludcmp	$\frac{2N^3}{3\sqrt{S}}$	1
	mvt	N^2	1
	nussinov	$\frac{N^3}{3\sqrt{S}}$	2
	seidel2d	$\frac{4N^2T}{\sqrt{S}}$	$6\sqrt{3}$
	symm	$\frac{2M^2N}{\sqrt{S}}$	1
	syr2k	$\frac{2MN^2}{\sqrt{S}}$	2
	syrk	$\frac{MN^2}{\sqrt{S}}$	2
	trisolv	$\frac{N^2}{2}$	1
	trmm	$\frac{M^2N}{\sqrt{S}}$	1
Neural Networks	Direct conv.	$\frac{2C_{in}C_{out}H_{out}NW_{out}W_{ker}H_{ker}}{\sqrt{S}}$	8
	Softmax	$4BHMN$	—
	MLP	$\frac{2N(fc_1fc_2+fc_1inp+fc_2out)}{\sqrt{S}}$	—
	LeNet-5	$\frac{300\sqrt{2}CHNW}{\sqrt{S}}$	—
	BERT Encoder	$\frac{4BHPL(L+2HP)}{\sqrt{S}}$	—
Various	LULESH	$22 \cdot$ numElem	—
	horizontal diff.	$2IJK$	—
	vertical adv.	$5IJK$	—

Table 2: Simplified leading-order terms of the I/O lower bounds extracted from multi-statement SOAP and previous state-of-the-art. For the direct convolution layer, the best previously known bound was published by Zhang et al. [20].

LULESH. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [30] application is an unstructured physics simulation. We analyze the main computational kernel, totaling over 60% of runtime within one time-step of the simulation from the full C++ source code. As LULESH falls outside the purview of affine programs, this result is the first reported I/O lower bound.

Numerical Weather Prediction. We select two benchmark stencil applications from the COSMO Weather Model [31] — horizontal diffusion and vertical advection — representatives of the two major workload types in the model's dynamical core.

Deep Neural Networks. For deep learning, we choose both individual representative operators (Convolution and Softmax) and network-scale benchmarks. Previous approaches only study data movement empirically [32]. To the best of our knowledge, we are the first to obtain I/O lower bounds for full networks, including a Multi-Layer Perceptron (MLP), the LeNet-5 CNN [33], and a BERT Transformer encoder [34].

8 RELATED WORK

I/O analysis spans almost the entire history of general-purpose computer architectures, and graph pebbling abstractions were among the first methods to model memory requirements. Dating back to challenges with the register allocation problem [35], pebbles were also used to prove space-time tradeoffs [36] and maximum parallel speedups by investigating circuit depths [37]. Arguably the most influential pebbling abstraction work is the red-blue pebble game by Hong and Kung [8] that explicitly models load and store operations in a two-level-deep memory hierarchy. This work was extended numerous times, by: adding blocked access [38], multiple memory hierarchies [25], or introducing additional pebbles to allow CDAG compositions [18]. Demaine and Liu proved that finding the optimal pebbling in a standard and no-deletion red-blue pebble game is PSPACE-complete [17]. Papp and Wattenhofer introduced a game variant with a non-zero computation cost and investigated pebbling approximation algorithms [39].

Although the importance of data movement minimization is beyond doubt, the general solution for arbitrary algorithms is still an open problem. Therefore, many works were dedicated to investigate lower bounds only for single algorithms (often with accompanying implementations), like matrix-matrix multiplication [16, 40–42], LU [41] and Cholesky decompositions [43, 44]. Ballard et al. [45] present an extensive collection of linear algebra algorithms. Moreover, a large body of work exists for minimizing communication in irregular algorithms [46, 47], such as Betweenness Centrality [5], min cuts [48], BFS [49], matchings [50], vertex similarity coefficients [51], or general graph computations [52, 53, 53]. Many of them use linear algebra based formulations [54]. Recently, convolution networks gained high attention. The first asymptotic I/O lower bound for single-layer direct convolution was proved by Demmel et al. [23]. Chen et al. [55] propose a matching implementation, and Zhang et al. [20] present the first non-asymptotic I/O lower bound for Winograd convolution.

In parallel with the development of I/O minimizing implementations for particular algorithms, several works investigated I/O lower bounds for whole classes of programs. Christ et al. [7] use a discrete version of Loomis-Whitney inequality to derive asymptotic lower bounds for single-statement programs nested in affine loops. Demmel and Rusciano [22] extended this work and use discrete Hölder-Brascamp-Lieb inequalities to find optimal tilings for such programs. The polyhedral model [56] is widely used in practice by many compilers [57, 58]. However, polyhedral methods have their own limitations: 1) they cannot capture non-affine loops [59]; 2) while the representation of a program is polynomial, finding optimal transformations is still NP-hard [60]; 3) they are inapplicable for many neural network architectures, e.g., the Winograd algorithm for convolution [20].

Recently, Olivry et al. [19] presented IOLB — a tool for automatic derivation of non-parametric I/O lower bounds for programs that can be modeled by the polyhedral framework. IOLB employs both "geometric" projection-based bounds based on the HBL inequality [21], as well as the wavefront-based approach from Elango [61]. To the best of our knowledge, this is the only method that can handle multiple-statement programs. However, the IOLB model explicitly disallows recomputation that may be used to decrease the I/O cost, e.g., in the Winograd convolution algorithm, backpropagation, or vertical advection. Furthermore, the framework is strictly limited to affine access programs. Even then, our method is able to improve those bounds by up to a factor of $6\sqrt{6}$ (`fdtd2d`) using a single, general method without the need to use application-specific techniques, such as wavefront-based reasoning.

9 CONCLUSIONS

In this work we introduce SOAP — a broad class of statically analyzable programs. Using the explicit assumptions on the allowed overlap between arrays, we are able to precisely count the number of accessed vertices on the induced parametric CDAG. This stands in contrast with many state-of-the art approaches that are based on bounding projection sizes, as they need to underapproximate their union size, often resulting in a significant slack in constant factors of their bounds. Our single method is able to reproduce or improve existing lower bounds for many important scientific kernels from various domains, ranging from 2× increase in the lower bound for linear algebra (`cholesky`, `syrk`), to more than 10× for stencil applications (`fdtd2d`, `heat3d`).

Our SDG abstraction precisely models data dependencies in multiple-statement programs. It directly captures input and output reuse, and allows data recomputation. Armed with these tools, we are the first to establish I/O lower bounds for entire neural networks, as well as core components of the popular Transformer architecture.

We believe that our work will be further extended to handle data-dependent accesses (e.g., sparse matrices), as well as scale better with input program size. The derived maximum subcomputation sizes can guide compiler optimizations and development of new communication-optimal algorithms through tiling, parallelization, or loop fusion transformations.

10 ACKNOWLEDGEMENTS

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880). Tal Ben-Nun is supported by the Swiss National Science Foundation (Ambizione Project #185778). The authors wish to thank the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure and support.

REFERENCES

[1] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, , and M. Pericas, "Trends in Data Locality Abstractions for HPC Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 10, Oct. 2017.

[2] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2013, pp. 56–65.

[3] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat *et al.*, "Programming abstractions for data locality." PADAL Workshop 2014.

[4] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 3007–3020, 2017.

[5] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication," in *SC*, 2017.

[6] E. Solomonik, E. Carson, N. Knight, and J. Demmel, "Trade-offs between synchronization, communication, and computation in parallel linear algebra computations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 1, pp. 1–47, 2017.

[7] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick, "Communication lower bounds and optimal algorithms for programs that reference arrays–part 1," *arXiv preprint arXiv:1308.0068*, 2013.

[8] J. Hong and H. Kung, "I/O complexity: The red-blue pebble game," in *STOC*, 1981, pp. 326–333.

[9] M. Del Ben *et al.*, "Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution," *Comp. Phys. Comm.*, pp. 120–129, 2015.

[10] Q. Zheng and J. D. Lafferty, "Convergence analysis for rectangular matrix completion using burer-monteiro factorization and gradient descent," *CoRR*, 2016.

[11] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, 2019.

[12] C. D. Meyer, *Matrix analysis and applied linear algebra*. SIAM, 2000.

[13] A. Krishnamoorthy and D. Menon, "Matrix inversion using Cholesky decomposition," in *2013 signal processing: Algorithms, architectures, arrangements, and applications (SPA)*. IEEE, 2013, pp. 70–72.

[14] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, 2014.

[15] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "On characterizing the data access complexity of programs," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015.

[16] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.

[17] E. D. Demaine and Q. C. Liu, "Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 195–204.

[18] V. Elango *et al.*, "Data access complexity: The red/blue pebble game revisited," Tech. Rep., 2013.

[19] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, and F. Rastello, "Automated derivation of parametric data movement lower bounds for affine programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 808–822.

[20] X. Zhang, J. Xiao, and G. Tan, "I/O lower bounds for auto-tuning of convolutions in CNNs," 2020.

[21] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick, "Communication lower bounds and optimal algorithms for programs that reference arrays–part 1," *arXiv preprint arXiv:1308.0068*, 2013.

[22] J. Demmel and A. Rusciano, "Parallelepipeds obtaining HBL lower bounds," *arXiv preprint arXiv:1611.05944*, 2016.

[23] J. Demmel and G. Dinh, "Communication-optimal convolutional neural nets," *arXiv preprint arXiv:1802.06905*, 2018.

[24] G. Dinh and J. Demmel, "Communication-optimal tilings for projective nested loops with arbitrary bounds," *arXiv preprint arXiv:2003.00119*, 2020.

[25] J. E. Savage, "Extending the hong-kung model to memory hierarchies," in *International Computing and Combinatorics Conference*. Springer, 1995, pp. 270–281.

[26] Q. Liu, "Red-blue and standard pebble games : Complexity and applications in the sequential and parallel models," 2018.

[27] G. Kwasniewski, T. Ben-Nun, A. N. Ziogas, T. Schneider, M. Besta, and T. Hoefler, "On the parallel I/O optimality of linear algebra kernels: Near-optimal LU factorization," 2020.

[28] L. N. Pouchet, "PolyBench: The Polyhedral Benchmark suite," 2016. [Online]. Available: https://sourceforge.net/projects/polybench

[29] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[30] J. Keasler and USDOE, "Livermore unstructured lagrange explicit shock hydrodynamics," 9 2010. [Online]. Available: https://www.osti.gov//servlets/purl/1231396

[31] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer, "Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities." *Monthly Weather Review, 139:3387–3905*, 2011.

[32] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," 2020.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[35] R. Sethi, "Complete register allocation problems," in *STOC*, 1973.

[36] W. J. Paul and R. E. Tarjan, "Time-space trade-offs in a pebble game," *Acta Informatica*, vol. 10, no. 2, pp. 111–115, Jun 1978.

[37] P. W. Dymond and M. Tompa, "Speedups of deterministic machines by synchronous parallel machines," *Journal of Computer and System Sciences*, vol. 30, no. 2, pp. 149–161, 1985.

[38] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *CACM*, Sep. 1988.

[39] P. A. Papp and R. Wattenhofer, "On the hardness of red-blue pebble games," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 419–429.

[40] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017 – 1026, 2004.

[41] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6853, pp. 90–109. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23397-5_10

[42] J. Demmel *et al.*, "Communication-optimal parallel recursive rectangular matrix multiplication," in *IPDPS*, 2013, pp. 261–272.

[43] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication-optimal parallel and sequential Cholesky decomposition," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010.

[44] E. Hutter and E. Solomonik, "Communication-avoiding Cholesky-QR2 for rectangular matrices," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 89–100.

[45] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, "Communication lower bounds and optimal algorithms for numerical linear algebra," *Acta Numerica*, vol. 23, p. 1, 2014.

[46] M. Besta and T. Hoefler, "Accelerating irregular computations with hardware transactional memory and active messages," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 161–172.

[47] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz *et al.*, "The future is big graphs! a community view on graph processing systems," *arXiv preprint arXiv:2012.06171*, 2020.

[48] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler, "Communication-avoiding parallel minimum cuts and connected components," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 219–232, 2018.

[49] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "Slimsell: A vectorizable graph representation for breadth-first search," in *IPDPS*, 2017.

[50] M. Besta, M. Fischer, T. Ben-Nun, D. Stanojevic, J. D. F. Licht, and T. Hoefler, "Substream-centric maximum matchings on fpga," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–33, 2020.

[51] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 1122–1132.

[52] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[53] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger *et al.*, "Graphminesuite: Enabling high-performance and programmable graph mining algorithms with set algebra," *arXiv preprint arXiv:2103.03653*, 2021.

[54] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS," *arXiv:1606.05790*, 2016.

[55] X. Chen, Y. Han, and Y. Wang, "Communication lower bound in convolution accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 529–541.
[56] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 132–146. [Online]. Available: https://doi.org/10.1007/978-3-540-78791-4_9
[57] "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.
[58] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.
[59] T. Hoefler and G. Kwasniewski, "Automatic complexity analysis of explicitly parallel programs," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, 2014, pp. 226–235.
[60] A. Darte, "On the complexity of loop fusion," in *PACT*, 1999, pp. 149–157.
[61] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "On characterizing the data movement complexity of computational DAGs for parallel execution," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, 2014, pp. 296–306.