# Reduction Fusion for Optimized Distributed Data-Parallel Computations via Inverse Recomputation

Haoxiang Lin*
Microsoft Research
Beijing, China
haoxlin@microsoft.com

Yang Wang
Microsoft Research
Beijing, China
yang.wang92@microsoft.com

Yanjie Gao
Microsoft Research
Renmin University of China
Beijing, China
yanjga@microsoft.com

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Ming Wu
Zero Gravity Labs
San Francisco, USA
ming@0g.ai

Mao Yang
Microsoft Research
Beijing, China
maoyang@microsoft.com

## Abstract

Distributed data-parallel computations are critical for both traditional big data applications and emerging large language model tasks. The efficiency of these computations largely depends on reducer performance, particularly in handling extensive data access. This paper introduces a novel reduction fusion algorithm that optimizes distributed data-parallel programs by fusing dependent reducers and mappers into a single, unified reducer. Employing inverse recomputation, the algorithm preserves partial aggregation and reduces storage, network I/O, memory, and cache overheads. Our preliminary evaluation reveals performance improvements of up to 2.47×, demonstrating the practicality and effectiveness of this approach, while also highlighting its potential to address challenges posed by extensive data access in modern distributed computing environments.

**ACM Reference Format:**
Haoxiang Lin, Yang Wang, Yanjie Gao, Hongyu Zhang, Ming Wu, and Mao Yang. 2025. Reduction Fusion for Optimized Distributed Data-Parallel Computations via Inverse Recomputation. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728496

## 1 Introduction

Distributed data-parallel computations have been critical in many application areas for decades, including evidence-based medicine, fraud detection, personalized education, and large language model (LLM) training. MapReduce [10] is the pioneering programming model designed for processing big datasets in parallel across a cluster of machines. Subsequent general-purpose distributed computing systems, such as SCOPE [5], DryadLINQ [37], Pig Latin [30], Apache Hive [4], and Apache Spark [39], provide developers with more convenient, SQL-like programming models that greatly simplify various data-processing tasks. These models typically decompose a sophisticated computation into a series of *map* and *reduce* (also known as aggregate or fold in functional programming) procedures, where mappers independently manipulate individual data items and reducers aggregate groups of them.

A major performance challenge in executing a series of dependent reducers and mappers arises from *extensive data access* [15, 36]. These computations require continuous reading and writing of large amounts of data from the original input and intermediate results. Extensive data access imposes significant overhead on the storage subsystem, as all input and intermediate results must be retained until the corresponding computations are complete. Moreover, it substantially elevates network I/O across the entire cluster and intensifies memory traffic and cache invalidation on individual machines, resulting in severe runtime performance degradation. This issue is further amplified in LLM tasks due to the constrained capacity of GPU L1 cache and shared memory (e.g., up to 256 KB for NVIDIA H100 [29]). As a result, computations like `SOFTMAX` [14] in attention layers [7, 35] can become critical performance bottlenecks, primarily due to the extended latency introduced by frequent data transfers between GPU memory and cache.

This paper presents a novel reduction fusion algorithm that optimizes distributed data-parallel programs by minimizing overhead caused by extensive data access. The proposed algorithm identifies opportunities to fuse a series of dependent reducers and mappers into a single, unified reducer through inverse recomputation. Our approach hinges on the idea that the fused reducer can *speculatively execute* all associated reducers and mappers based on their dependencies, using only a portion of the initial input data. When new data items are introduced, the algorithm begins with the partial result and reconstructs the input data for each reducer and mapper by applying their inverse functions. Given that reducers are not necessarily injective (i.e., they may produce the same output for different inputs), the algorithm can potentially reproduce much less input data than was originally processed. Once the initial input data is reconstructed, the fused reducer appends the newly received items and reruns the computations in the forward direction. As a result, if the inverse reproduction of data items by the reducers is sufficiently efficient in terms of data size (as with the `MIN`, `MAX`, and `SUM` reducers in Figure 2), the reduction fusion can outperform the original computations. This improvement is achieved by trading

*Haoxiang Lin is the corresponding author.


*FSE Companion '25, Trondheim, Norway*

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728496

```
1 static V reduce(Func<T, V, V> g, V v0, List<T> input) {
2     var res = v0;
3     foreach (var item in input) {
4       res = g(item, res);
5     }
6     return res;
7 }
8 static List<U> map(Func<T, V, U> f,  V v0, List<T> input) {
9     var res = new List<U>();
10    foreach (var item in input) {
11      var out_item = f(v0, item);
12      if (out_item != null) {
13        res.Add(out_item);
14      }
15    }
16    return res;
17 }
```

**Figure 1: General interface and illustrative C# implementation of reducers and mappers. $T$, $V$, and $U$ represent generic types. The Func delegate encapsulates a function that accepts one or more parameters and returns a value.**

CPU/GPU recomputation for substantial reductions in overhead associated with storage, network I/O, memory, and cache usage.

Moreover, our approach retains the capability of *partial aggregation* [1, 18, 22, 27, 38], provided that each reducer supports it. Our preliminary evaluation shows that the approach achieves a substantial performance speedup of up to 2.47× (see Section 4).

In summary, this paper makes the following contributions:

(1) Identification of a significant, challenging research problem arising from extensive data access.
(2) Proposal of a novel algorithm consolidating multiple dependent reducers and mappers into a single reducer through inverse recomputation. The reduction fusion eliminates unnecessary overhead and enhances runtime performance.
(3) Demonstration of practicality and effectiveness through preliminary evaluation results.

## 2 Background and Motivation

Figure 1 presents a generalized interface [17] and an illustrative implementation of reducers and mappers in the C# programming language. The general `reduce` function takes an *accumulator* [22] $g$ of type $T \rightarrow V \rightarrow V$ for result aggregation, an initial value $v_0$ of type $V$ (i.e., the default result for empty lists), and a list of data items *input* of type $[T]$ as its parameters. Such a function sequentially processes the dataset item by item, using $g$ and the previous aggregated value to produce a new value; it finally returns a result of type $V$. Because of their sequential semantics, the runtime performance of reducers at scale is critical for distributed data-parallel computations—not only in traditional big data applications but also in emerging deep learning and LLM tasks.

Fusion techniques [2, 12, 19, 20, 24, 32] have shown promise in program optimization, particularly by consolidating multiple loops, functions, or program segments into a singular, efficient unit. However, prior research primarily focuses on code written in traditional programming languages such as C, C++, Java, or Haskell, thereby limiting its direct applicability to distributed data-parallel programs due to substantial differences in their structural representations. Partial aggregation [1, 18, 22, 27, 38] is a crucial optimization that decreases the volume of network I/O needed for data transmission across machines. Specifically, the system locally aggregates intermediate data grouped by a specific key on each worker machine in parallel. It then combines these partial results using hierarchical aggregation trees and performs a final reduction on the sequence of aggregated partial results, ultimately computing the end result.

```
1 static double softmax_test(List<double> input) {
2     // SOFTMAX Begin
3     double max = input.Max();
4     var y1 = input.Select(x => x - max);
5     var y2 = y1.Select(x => (double)Math.Exp(x));
6     double sum = y2.Sum();
7     var y = y2.Select(x => x / sum);
8     // SOFTMAX End
9     return y.Min();
10 }
```

**Figure 2: Distributed data-parallel example in C# LINQ that applies SOFTMAX to a list of floating-point numbers, and then retrieves the minimum value.**

Figure 2 presents an example of distributed data-parallel processing using C# LINQ [25], demonstrating the application of the SOFTMAX operation [14] to a list of double-precision floating-point numbers, followed by retrieving their minimum value. This illustrative implementation decomposes SOFTMAX into two reducers (i.e., MAX and SUM) and three mappers (lines 4, 5, and 7). Retrieving the minimum adds one additional data access, resulting in a total of six data processing steps. This example highlights the feasibility of efficiently reconstructing data items through inverse operations. For instance, the MIN and MAX reducers, which return the minimum and maximum values from a list of data items, respectively, have computationally equivalent yet simple inverse functions: $\mathtt{MIN}^{-1}(y) = [y], \mathtt{MAX}^{-1}(y) = [y]$. Another example is the SUM reducer, which performs sum calculations. Its inverse function uses uniform value replication, expressed as $\mathtt{SUM}^{-1}(y) = \left[\frac{y}{n}\right]^n$. The notation $\left[\frac{y}{n}\right]^n$ represents an abbreviation for an $n$-sized list in which each item has the same value $\frac{y}{n}$, with $n$ being the length of the original input. This format enables efficient storage by storing only the value and the length, rather than the entire list.

## 3 Methodology

### 3.1 Problem Definition

Dependent reducers and mappers are formally modeled as a directed acyclic graph (DAG), also referred to as an execution plan or a computation graph: $G = \langle V = \{u_i\}, E = \{(u_i, u_j)\}_{i \neq j} \rangle$. In this representation, a source node represents either an initial input or an initial value, while a sink node denotes a final output. An internal node corresponds to a reducer or a mapper. A directed edge $(u_i, u_j)$ signifies that the output of node $u_i$—which may be a list of data items or a single value—is passed as input to node $u_j$, enforcing the constraint that $u_j$ can begin execution only after $u_i$ has completed its task. The entire graph $G$ can be viewed as a mathematical function denoted $\mathcal{F}_G$.

Assume that the initial input set of $G$ is given by $X = \{X_1 : [T_1], X_2 : [T_2], \cdots, X_m : [T_m]\}$, and the initial value set is $v_0 = \{v_{01} : V_1, v_{02} : V_2, \cdots, v_{0l} : V_l\}$. In this context, each $X_i$ represents a list of data items of type $[T_i]$, and each $v_{0i}$ is an initial value of type $V_i$. We assume that all $X_i$ have the same length; otherwise, data packing is employed. Similarly, let $y = \{y_1 : V_{l+1}, y_2 : V_{l+2}, \cdots, y_n : V_{l+n}\}$ represent the final outputs from all sink nodes. To enable the fused reducer to process both the initial input set and the final

output set, we make a few adjustments to $X$ and $y$. First, we merge all $X_i$ into a single list $\overline{X}$, where each item is a tuple combining elements from the respective $X_i$, yielding a type $[T]$ such that $T = T_1 \times T_2 \times \cdots \times T_m$. Second, we introduce a sum type [31] $U = V_1 + \cdots + V_l + V_{l+1} + \cdots + V_{l+n}$ to represent a $v_{0i}$ or a $y_j$. We then construct a list $\overline{v_0} = [v_{01}, v_{02}, \cdots, v_{0l}]$, with type $[U]$, to serve as the initial value for the fused reducer. Lastly, the final output of the fused reducer is expressed as a list $\overline{y} = [y_1, y_2, \cdots, y_n]$, also with type $[U]$. Using these notations, we define reduction fusion as follows [23].

**Definition 1.** An algorithm $\mathcal{A}$ qualifies as a reduction fusion algorithm if it meets the following two conditions:

(1) **Syntactic Legality**. For any given DAG $G$, algorithm $\mathcal{A}$ generates a valid reducer $r_{g,\overline{v_0}}$ (as illustrated in Figure 1):

$$G \vdash_{\mathcal{A}} r_{g,\overline{v_0}} :: [T] \to [U] \ \wedge \ g :: T \to [U] \to [U] \\ \wedge \ \overline{v_0} :: [U] \,.$$

(2) **Semantic Equivalence**. For any valid input, the outputs of $G$ and the reducer $r_{g,\overline{v_0}}$ must always match:

$$y = \mathcal{F}_G(X, v_0) \ \vdash_{\mathcal{A}} \ \overline{y} = r_{g,\overline{v_0}}(\overline{X}) \,.$$

The symbols $\mathcal{F}_G$, $T$, $U$, $y$, $X$, $\overline{y}$, $v_0$, $\overline{v_0}$, and $\overline{X}$ are defined in the previous section.

## 3.2 Inverse Function

Let $r_{g,v_0}$ be a valid reducer. We define $r^{-1}_{g,v_0}$, the inverse function of $r_{g,v_0}$, which takes an aggregated result of type $V$ as input and returns a list of data items of type $[T]$, as follows:

$$r^{-1}_{g,v_0} :: V \to [T]$$
$$[x_1, \cdots, x_n] = r^{-1}_{g,v_0}(y) \Longrightarrow y = r_{g,v_0}([x_1, \cdots, x_n])$$

Since the relation $H = \{\langle y, [x_1, \cdots, x_n]\rangle \mid y = r_{g,v_0}([x_1, \cdots, x_n])\}$ is non-empty, the inverse function $r^{-1}_{g,v_0} \subseteq H$ must exist by the Axiom of Choice [13]. However, $r^{-1}_{g,v_0}$ does not necessarily return the original data items. It is important to note that the accumulator $g$ and the initial value $v_0$ are crucial for defining $r^{-1}_{g,v_0}$. In subsequent sections, we may use $r^{-1}$ for simplicity, provided that omitting $g$ and $v_0$ does not affect the discussion. Below, we list the inverse functions of several common reducers:

$$\texttt{MIN}^{-1}(y) = [y], \qquad \texttt{MAX}^{-1}(y) = [y] \,,$$
$$\texttt{SUM}^{-1}(y) = \left[\frac{y}{n}\right]^n, \quad \texttt{AVG}^{-1}(y) = [y]^n, \quad \texttt{COUNT}^{-1}(y) = [\_]^y$$

In $\texttt{SUM}^{-1}$ and $\texttt{AVG}^{-1}$, we retain $n$, the number of original data items. As noted earlier, $[y]^n$ represents an abbreviation for an $n$-sized list where each item has the same value $y$. This format enables efficient storage of only the value and length, simplifying inverse recomputation. The underscore "_" in $\texttt{COUNT}^{-1}$ indicates that any value can be used in its place.

Similarly, for a mapper $m_{f,v_0}$, we define its inverse function $m^{-1}_{f,v_0}$. We observe that $m_{f,v_0}$ processes each data item independently; therefore we must first define the inverse function $f^{-1}$ for the functor $f$. Since the computation of $f$ depends on $v_0$, we adjust the function type of $f^{-1}$ to accept $v_0$ as an additional parameter.

$$f^{-1} :: V \to U \to T, \quad m^{-1} :: [U] \to [T]$$
$$x = f^{-1}(v_0, y) \Longrightarrow y = f(v_0, x)$$
$$m^{-1}_{f,v_0}([y_1, \cdots, y_n]) = [f^{-1}(v_0, y_1), \cdots, f^{-1}(v_0, y_n)]$$

As stated earlier, $f^{-1}$ must exist; therefore, $m^{-1}_{f,v_0}$ also exists.

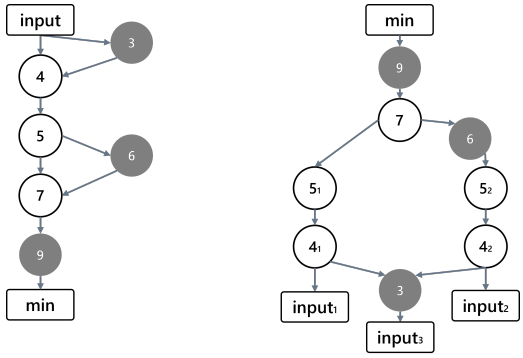


**Figure 3: Execution graph (left) and augmented reverse execution graph (right) for the SOFTMAX program in Figure 2. The internal numbers represent line numbers in the code.**

## 3.3 Reduction Fusion

This section introduces an algorithm designed to fuse a series of reducers and mappers into a single, unified reducer. Consider a DAG $G$ with $K$ sink nodes $su_i$ where $1 \le i \le K$. Assume that the inverse functions corresponding to the reducers and mappers are already known. In the preparation phase, we first compute a backward slice $G_i$ for each $su_i$ by tracing backward along the dependency edges of $G$ starting from $su_i$. In other words, $G_i$ is a subgraph of $G$, consisting of nodes (including $su_i$ itself) that influence the computation of $su_i$. The left panel of Figure 3 illustrates the execution graph for the SOFTMAX program shown in Figure 2. The numbers inside the circles represent line numbers in the code, with solid circles denoting reducers and hollow circles indicating mappers.

Next, we compute $G_i^{-1}$, the reverse graph of $G_i$, by substituting the internal nodes with their respective inverse functions and reversing the direction of the edges in $G_i$. A challenge arises when mappers or initial input nodes have multiple incoming edges in $G_i^{-1}$, complicating reverse execution due to the difficulty of meeting computational constraints across multiple paths. To address this, we apply the *node splitting transformation* [11], where a mapper or initial input node $u$ with $k$ incoming edges is replaced by $k$ clones $u_1, \cdots, u_k$, and each incoming edge to the original node is redirected to exactly one of the new nodes. The original outgoing edges of $u$ are then replicated to each of $u_1, \cdots, u_k$. The result is an augmented graph, denoted as $\overline{G_i^{-1}}$. As an example, the right panel of Figure 3 presents the augmented reverse execution graph for the SOFTMAX program, where certain original nodes are divided into two or three nodes (e.g., node ⑤ is split into $5_1$ and $5_2$). The reverse graph of $\overline{G_i^{-1}}$, referred to as $\overline{G_i}$, is then computed and used for forward computation after incorporating new data items. Additionally, we aggregate the original initial values into a list, denoted

as $\overline{v_0}$, which serves as the new initial value for the fused reducer $r$, as discussed in Section 3.1.

We now describe the functioning of the accumulator $g$ within the fused reducer $r$. Our analysis reveals that all sink nodes in the DAG $G$ operate independently of each other. This independence allows $g$ to perform inverse recomputations for each sink node separately, ensuring that the process is both efficient and logically coherent. For a given sink node $su_i$, the accumulator $g$ initiates the process by retrieving the corresponding partial result that has been previously computed. This partial result serves as input to the augmented reverse graph $\overline{G_i^{-1}}$. Subsequently, $\overline{G_i^{-1}}$ is executed starting from $su_i$ and proceeding along the reversed dependency edges. Through this reverse execution, $g$ reconstructs the partial initial inputs that are computationally equivalent to those originally provided. Afterward, $g$ augments such reconstructed inputs by appending the corresponding new data items. These inputs are then fed into $\overline{G_i}$, triggering a forward computation that mirrors the original processing sequence. This forward computation ensures that the new output produced by $\overline{G_i}$ accurately reflects the combined effects of both the original initial inputs and any subsequent new data. Note that the updated values of all $v_0$ for the internal reducers and mappers (e.g., the `max` and `sum` variables in Figure 2) are saved for the next round of inverse recomputation. Once all $su_i$ have been individually processed in this manner, the accumulator $g$ proceeds to consolidate the new outputs generated by each $\overline{G_i}$ into a new partial result.

This comprehensive approach ensures that the final result produced by the fused reducer $r$ remains consistent with that of the original DAG $G$. If the inverse reproduction of data items during the execution of each $\overline{G_i^{-1}}$ is sufficiently efficient in terms of data size, the accumulator $g$ can significantly optimize the utilization of storage, network I/O, memory, and cache, leading to substantial performance gains.

### 3.4 Partial Aggregation

As established earlier, if all reducers in a DAG $G$ support partial aggregation, then the fused reducer $r_{g,\overline{v_0}}$ also exhibits this property. The combiner and FinalReduce functions operate in a manner closely analogous to the accumulator $g$. Building on the work of Liu et al. [22], we derive the following key result:

**Theorem 1.** For any two valid initial inputs $\overline{X}$ and $\overline{X}'$, if

$$r_{g,\overline{v_0}}(\overline{X} \oplus \overline{X}') = r_{g,\overline{v_0}}(\overline{X}' \oplus \overline{X}),$$

then $r_{g,\overline{v_0}}$ supports partial aggregation. Here, $r_{g,\overline{v_0}}$ is computationally equivalent to $G$, and $\oplus$ denotes the concatenation operation on two lists.

Due to page constraints, we omit the detailed proof here; it will be included in an extended version of this paper.

## 4 Preliminary Evaluation

We conducted a preliminary evaluation of the `SOFTMAX` program (Figure 2) on a production distributed computing platform. The raw input dataset comprised $2^{40}$ double-precision floating-point numbers, totaling 8 TiB of data in memory. The fused version of the program was implemented following the algorithm described in Section 3.3, incorporating the inverse functions outlined in Section 3.2. The correctness of such an approach can be rigorously proven through induction [7, 26]. We executed both the original and fused versions of the `SOFTMAX` program as standard jobs using 256, 512, 1,024, and 2,048 CPU cores, with only partial reduction enabled. Our approach achieved substantial end-to-end speedups of 1.58×, 1.76×, 1.81×, and 2.47× across these configurations, highlighting its efficiency and potential for performance optimization.

## 5 Future Plan

The primary challenge of our reduction fusion algorithm lies in constructing inverse functions for the relevant reducers and mappers. This task is essentially a nondeterministic program inversion problem [6, 16, 22, 33, 34], which is inherently difficult to solve. To address this challenge, we will pursue three key strategies. First, we have prebuilt inverse functions for several commonly used operators in data-parallel computations, including aggregate (e.g., `MIN`, `MAX`, `AVG`, `SUM`, and `COUNT`), arithmetic (e.g., addition, subtraction, multiplication, division, and exponentiation), and bitwise (e.g., `AND` and `OR`) operators. We plan to enhance this library by predefining inverse functions for a broader set of common reducers and mappers. Second, we aim to design an interface that enables developers to write their own inverse functions and seamlessly integrate them with our algorithm. Additionally, we will employ verification techniques, such as SMT solvers [8, 9], to ensure the correctness of these functions. Third, we intend to analyze user-written programs to automatically infer their inverse functions, drawing on techniques from program analysis [28], symbolic execution [3, 21], and SMT solvers.

The second challenge is that the selection of inverse functions varies based on the nature of the data-parallel computations. For example, it is feasible to manually prove the correctness of our proposed inverse functions by induction for the `SOFTMAX` test illustrated in Figure 2 [7, 26]. However, this process becomes more complex if computations on lines 4 or 5 are modified. In such scenarios, new inverse functions for `SUM`, `MAX`, and `MIN` would be required. To address this challenge, we aim to leverage formal methods, including program analysis and symbolic execution, to infer appropriate inverse functions automatically. These methods will help ensure the correctness of the computation across varying configurations.

To make our algorithm more practical and impactful, we intend to integrate it into the query optimizers of widely used distributed computing systems, such as SCOPE [5] and Apache Spark [39]. We also plan to explore applications of our technique in the emerging fields of deep learning and large language models [7].

## 6 Conclusion

This paper presents a reduction fusion algorithm that optimizes distributed data-parallel computations through the fusion of dependent reducers and mappers. The proposed approach can significantly reduce various overheads by leveraging inverse recomputation and preserving partial aggregation. Preliminary results validate the effectiveness of this approach. We believe that this work contributes valuable insights for the future development of distributed data-parallel programs.

## References


[1] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 136–149. doi:10.1145/3514221.3517889

[2] Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2020. Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs. *ACM Trans. Archit. Code Optim.* 17, 4, Article 26 (sep 2020), 26 pages. doi:10.1145/3416510

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[4] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. *CoRR* abs/1903.10970 (2019). arXiv:1903.10970

[5] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1265–1276. doi:10.14778/1454159.1454166

[6] Wei Chen and Jan Tijmen Udding. 1990. Program inversion: More than fun! *Science of Computer Programming* 15, 1 (1990), 1–13. doi:10.1016/0167-6423(90)90042-C

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS '08/ETAPS '08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[9] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. 2007. A Tutorial on Satisfiability modulo Theories. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV '07)*. Springer-Verlag, Berlin, Heidelberg, 20–36.

[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[11] Reinhard Diestel. 2017. *Graph Theory* (5th ed.). Springer Publishing Company, Incorporated.

[12] Facundo Domínguez and Alberto Pardo. 2011. Exploiting algebra/coalgebra duality for program fusion extensions. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications* (Saarbrucken, Germany) *(LDTA '11)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. doi:10.1145/1988783.1988789

[13] Herbert B. Enderton. 1977. *Elements of Set Theory*. Academic Press, San Diego.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[15] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 121–133. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/guo

[16] Qinheping Hu and Loris D'Antoni. 2017. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 376–389. doi:10.1145/3062341.3062345

[17] Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.* 9, 4 (jul 1999), 355–372. doi:10.1017/S0956796899003500

[18] Myung-Hwan Jang, Yunyong Ko, Hyuck-Moo Gwon, Ikhyeon Jo, Yongjun Park, and Sang-Wook Kim. 2023. SAGE: A Storage-Based Approach for Scalable and Efficient Sparse Generalized Matrix-Matrix Multiplication. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (Birmingham, United Kingdom) *(CIKM '23)*. Association for Computing Machinery, New York, NY, USA, 923–933. doi:10.1145/3583780.3615044

[19] Patricia Johann. 2000. Testing and enhancing a prototype program fusion engine. *SIGSOFT Softw. Eng. Notes* 25, 1 (jan 2000), 60–61. doi:10.1145/340855.340964

[20] Ken Kennedy and John R. Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. doi:10.1145/360248.360252

[22] Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. 2014. Automating Distributed Partial Aggregation. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2670979.2670980

[23] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the Interoperability between Deep Learning Frameworks by Model Conversion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1320–1330. doi:10.1145/3368089.3417051

[24] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. 2014. Revisiting loop fusion in the polyhedral framework. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 233–246. doi:10.1145/2555243.2555250

[25] Microsoft. 2023. Language Integrated Query (LINQ). https://learn.microsoft.com/en-us/dotnet/csharp/linq/.

[26] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *CoRR* abs/1805.02867 (2018). arXiv:1805.02867

[27] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New query optimization techniques in the Spark engine of Azure synapse. *Proc. VLDB Endow.* 15, 4 (dec 2021), 936–948. doi:10.14778/3503585.3503601

[28] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[29] NVIDIA. 2024. Hopper Tuning Guide. https://docs.nvidia.com/cuda/pdf/Hopper_Tuning_Guide.pdf.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1099–1110. doi:10.1145/1376616.1376726

[31] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[32] Simon Robillard. 2014. Catamorphism Generation and Fusion Using Coq. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 180–185. doi:10.1109/SYNASC.2014.32

[33] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 492–503. doi:10.1145/1993498.1993557

[34] Finn Teegen, Kai-Oliver Prott, and Niels Bunkenburg. 2021. Haskell$^{-1}$: automatic function inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 41–55. doi:10.1145/3471874.3472982

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[36] Tian Xiao, Zhenyu Guo, Hucheng Zhou, Jiaxing Zhang, Xu Zhao, Chencheng Ye, Xi Wang, Wei Lin, Wenguang Chen, and Lidong Zhou. 2014. Cybertron: pushing the limit on I/O reduction in data-parallel programs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 895–908. doi:10.1145/2660193.2660204

[37] Yuan Yleyu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. USENIX Association, San Diego, CA.

[38] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 247–260. doi:10.1145/1629575.1629600

[39] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. doi:10.1145/2934664