# Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns

Kevin J. Brown* HyoukJoong Lee*‡ Tiark Rompf† Arvind K. Sujeeth*
Christopher De Sa* Christopher Aberger* Kunle Olukotun*

*Stanford University, USA †Purdue University, USA ‡Google, USA
{kjbrown,hyouklee,asujeeth,cdesa,caberger,kunle}@stanford.edu tiark@purdue.edu

## Abstract

High performance in modern computing platforms requires programs to be parallel, distributed, and run on heterogeneous hardware. However programming such architectures is extremely difficult due to the need to implement the application using multiple programming models and combine them together in ad-hoc ways. To optimize distributed applications both for modern hardware and for modern programmers we need a programming model that is sufficiently expressive to support a variety of parallel applications, sufficiently performant to surpass hand-optimized sequential implementations, and sufficiently portable to support a variety of heterogeneous hardware. Unfortunately existing systems tend to fall short of these requirements.

In this paper we introduce the Distributed Multiloop Language (DMLL), a new intermediate language based on common parallel patterns that captures the necessary semantic knowledge to efficiently target distributed heterogeneous architectures. We show straightforward analyses that determine what data to distribute based on its usage as well as powerful transformations of nested patterns that restructure computation to enable distribution and optimize for heterogeneous devices. We present experimental results for a range of applications spanning multiple domains and demonstrate highly efficient execution compared to manually-optimized counterparts in multiple distributed programming models.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—concurrent, distributed, and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—frameworks, patterns; D.3.4 [*Programming Languages*]: Processors—code generation, compilers, optimization

***Keywords*** Parallel patterns, distributed memory, pattern transformations



*CGO '16,* March 12-18, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2854038.2854042

## 1. Introduction

Modern hardware is trending towards increasingly parallel and heterogeneous architectures. Many machines' processors are spread across multiple sockets, where each socket can access some system memory faster than the rest, creating non-uniform memory access (NUMA). Efficiently utilizing these NUMA machines with large memory capacities is becoming increasingly important due to the prevalence of very large datasets (big data), as they enable local in-memory computation that is significantly faster than reading the data from disk or the network. In addition, many machines also contain specialized accelerators such as GPUs. Efficiently utilizing all of the hardware resources available in these machines is very difficult with the programming models in widespread use today, and fully utilizing a cluster of these machines is even more challenging. Combining together multiple classic parallel programming models such as using MPI to communicate between machines, Pthreads to parallelize across cores within a machine, and CUDA to offload work to each GPU, can produce a highly efficient implementation but is extremely low-level, tedious, and error-prone. Such programs are traditionally not only significantly harder to write but also significantly harder to maintain. What we want instead is a single system and programming model that is optimized both for modern hardware (high performance) and for modern programmers (high productivity). This system should be capable of running a variety of parallel applications, provide sequential baseline performance comparable to hand-optimized implementations, and continue to function well both as more resources are added to a single machine (scale up) and more machines are interconnected (scale out).

In recent years many new parallel and distributed systems have been introduced that provide higher-level programming models. Languages such as Scala and Haskell contain collections libraries in which certain data-parallel operations are implicitly parallelized across multiple cores within a machine [20, 29]. Other systems such as MapReduce [11], Dryad [19], and Spark [35] specifically target clusters of machines and present users with a more restrictive programming model. While many of these systems have

| System | Programming Model Features | | | | | Supported Hardware | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Rich data parallelism | Nested programming | Nested parallelism | Multiple collections | Random reads | Multi-core | NUMA | Clusters | GPUs |
| MapReduce [11] | | | | | | | | • | |
| DryadLINQ [18] | • | • | | | | | | • | |
| Thrust [15] | | | | | • | | | | • |
| Scala Collections [29] | • | • | • | • | • | • | | | |
| Delite [7] | • | • | | • | • | • | | | • |
| Spark [35] | • | | | | | • | | • | |
| Lime [2] | | • | • | • | • | • | | | • |
| PowerGraph [13] | | | | | • | • | | • | |
| Dandelion [31] | • | • | | | | • | | • | • |
| DMLL | • | • | • | • | • | • | • | • | • |

**Table 1.** Comparison of programming model features and heterogeneous hardware targets supported by various parallel frameworks in chronological order.

shown strong scalability, they also have very high overheads, sometimes requiring hundreds of processors to surpass the performance of optimized sequential C++ implementations [25]. Programming NUMA machines is possible with both types of systems, but also typically inefficient with both. Multi-core frameworks do not account for the need to distribute data across the different memory regions and therefore scale poorly, and cluster frameworks incur significant overhead by using serialization and explicit messages to communicate between regions.

We compare the programming model features and supported hardware targets of a variety of recent parallel systems in Table 1. In particular we identify several features useful for implementing real-world applications that are common in traditional programming models but various parallel systems have chosen to sacrifice. These features include *rich data parallelism*, the ability to use and compose a rich set of data-parallel operations beyond the classic `map` and `reduce`; and *nested programming*, the ability to logically nest parallel constructs. Many systems require a different (sequential) programming model within parallel computation. Even if parallel constructs can be logically nested, the system may or may not actually be capable of exploiting all levels of parallelism at runtime; we refer to this as *nested parallelism*. Systems that support *multiple collections* allow parallel computations to directly consume more than one parallel collection rather than having to first join the collections in some fashion. This is very useful in certain domains, e.g., linear algebra. Finally *random reads* allow arbitrary read access patterns of parallel collections rather than restricting reads to only the local element, which is very useful in, e.g., graph analytics. As shown in the figure, previous systems have been forced to sacrifice many of these features in exchange for expanding to more complex hardware targets. For example, Scala's parallel collection library has the complete set of features described but is restricted to only run on basic multi-core configurations, while its distributed cousin Spark sacrificed almost all of these features for automatic distribution. It is worth noting that the systems that are most feature- and hardware-complete (Delite [7], Lime [2], Dandelion [31], DMLL) are compilers rather than libraries.

In this paper our goal is to extend a data-parallel programming model to heterogeneous distributed hardware without sacrificing these programming model features, and without sacrificing single-machine performance. We introduce a novel intermediate language, the Distributed Multiloop Language (DMLL), and illustrate how applications written using data-parallel patterns in this language can be optimized to run as efficiently as hand-optimized implementations and also scale to multiple heterogeneous hardware targets.

In particular, we make the following contributions:

- We demonstrate that implicitly parallel patterns can be used to automatically distribute code and data using straightforward analyses. This is in contrast to existing systems that require explicitly distributed data structures.
- We present a novel intermediate language and a corresponding set of transformations for nested parallel patterns that restructure computation for heterogeneous devices. This enables automatically generating performance-portable code for heterogeneous hardware as opposed to directly translating the algorithms as written.
- We are the first to show a complete compiler and runtime system that can execute implicitly parallel, single-source applications across a heterogeneous cluster that includes non-uniform memory and accelerators.
- We present experimental results for multiple applications demonstrating high performance for automatically-optimized implementations compared to manually optimized implementations using existing state-of-the-art distributed programming models.

## 2. Related Work

This paper builds on a rich history of work from the functional programming, high performance computing (HPC), and database communities.

***High-level Parallel Programming*** Many distributed and heterogeneous programming frameworks have previously been proposed. One of the most famous is MapReduce [11] for large clusters. More recent proposals have focused on addressing the inefficiencies of this model as in Spark [35] and/or adding more generality as in Dryad [19] and DryadLINQ [18]. Other systems such as Pregel [23] and PowerGraph [13] have extended this model particularly for graph analytics. Galois [26] performs efficient graph analytics in shared memory but does not scale out to distributed systems.

Dandelion [31] compiles LINQ programs to clusters and GPUs. Lime compiles Java programs to GPUs [12] and FPGAs [2] but provides a much more imperative programming model with only basic data-parallel operations. X10 [9] also supports only basic data parallelism over arrays but targets explicitly distributed memory. Delite [7] provides a richer set of functional operators but only targets multi-core and GPUs. NESL [4] and Data Parallel Haskell [20] support nested parallelism, but unlike DMLL they rely on flattening transformations which can reduce efficiency.

The trade-offs in programing model and hardware targets for these systems are summarized in Table 1. With DMLL we unify the most useful programming model features into a single system for data analytics and add support for one of

the most performant but previously overlooked hardware targets: big-memory NUMA machines. Supporting these features is made possible primarily through multiple compiler optimizations on parallel patterns.

***Pattern Transformations*** The nested pattern transformations in this paper leverage recent work on implementing extensible compilers for high-level programs [30], on program transformation using rewrite rules [6], and on combining optimizations in a modular fashion [10, 22, 33]. The transformations presented in this paper are designed to improve locality and are therefore similar in spirit to multiple previous transformation systems. Systems designed to optimize imperative loops using polyhedral analysis such as Pluto [5], PPCG [34], and Polly [14] can perform automatic tiling, parallelization, and distribution of nested loops to target CPUs and/or GPUs. However, while these systems work well for loops with completely affine memory accesses, they fail on commonly occurring data-dependent operations such as *filters* and *groupBys* [3]. The formalism presented in this paper exploits the higher level semantics of parallel patterns over generic *for* loops to overcome this limitation. This is particularly critical for data analytics compared to HPC as such data-dependent operations are extremely common.

Spartan [17] also performs automatic tiling and data distribution, but via parallel patterns on mult-dimensional arrays. In contrast to our work, Spartan focuses solely on the data distribution, and not how to optimize the computation at multiple levels. Systems such as FlumeJava [8] and Data Parallel Haskell [20] also use a rewrite system to optimize parallel patterns, but they only focus on pipeline fusion and flattening transformations across functional operators and do not consider optimizing nested parallelism.

## 3. Locality-Enhancing Transformations

Parallel patterns have become an increasingly popular choice for abstracting parallel programming and appear in multiple recent systems [7, 11, 15, 20, 24, 35]. Previous work has shown how a relatively small number of parallel patterns can naturally capture a wide range of operations across multiple domains [32]. The key benefit of these patterns is that they abstract over the low-level implementation differences and encode the parallelism and synchronization requirements at a sufficiently high level to be portably mapped to hardware targets as varied as large data centers [11] and an FPGA [2].

Modern distributed systems often provide a parallel pattern API but also enforce certain restrictions that require the user to reason about data movement explicitly. To better illustrate the challenges of writing applications within such restrictive distributed programming models, consider how we might parallelize and distribute the classic $k$-means application, which can be implemented very succinctly in two different ways using data-parallel patterns as shown in Figure 1. In this example we show a single iteration of the core computation loop for simplicity. The application assigns each data point to the nearest cluster, and then computes new cluster locations by averaging the data points assigned to each current cluster. With some basic pipeline fusion optimizations our initial implementation performs quite well on multi-core but suffers at larger scales due to the parallelization over the relatively small dataset `clusters`. Attempting to partition the data is also quite challenging since as written an unknown subset of `matrix` is required to compute an element of `newClusters`. Therefore this implementation cannot be directly ported to typical distributed programming models.

```
1  //matrix: large dataset being clustered
2  val matrix = Matrix.fromFile(path)(parseMatrix)
3  val clusters = Matrix.fromFunction(numClusters, numFeatures)(
4                                        (i,j) => math.random())
5
6  //shared-memory version
7  //data implicitly shuffled via indexing operation 'matrix(as)'
8  val assigned = matrix.mapRows { row =>
9    clusters.mapRows(centroid => dist(row, centroid)).minIndex
10 }
11 val newClusters = clusters.mapIndices { i =>
12   val as = assigned.filter(a => a == i)
13   matrix(as).sumRows.map(s => s / as.count)
14 }
15
16 //distributed-memory version
17 //data explicitly shuffled via 'groupBy' operation
18 val clusteredData = matrix.groupRowsBy { row =>
19   clusters.mapRows(centroid => dist(row, centroid)).minIndex
20 }
21 val newClusters = clusteredData.map(e => e.sum / e.count)
```

**Figure 1.** Two ways of writing the core computation (one iteration) of $k$-means clustering using data-parallel patterns.

We must instead figure out how to rewrite $k$-means in a more distributed-friendly style, shown in the second half of Figure 1. Now all of the outer-level parallelism is clearly over the matrix rows, which are explicitly shuffled for the following average, making the optimal data distribution strategy much more apparent. Once we have found this strategy we now need to rewrite the application to expose this information explicitly to the distributed framework. In Spark, for example, this can be achieved by lowering the matrix to an `RDD[Vector]` where each `Vector` represents a matrix row (an RDD is a linear distributed collection). Unfortunately, performing this lowering makes targeting GPUs very difficult due to the inefficiencies of reducing vector types. For GPUs we instead need a different lowering transformation that transposes the operation and reduces scalars instead of vectors. We present solutions to these issues in the remainder of this section.

### 3.1 The Distributed Multiloop Language

Generating efficient implementations of applications written like the above example requires multiple high-level optimizations. Making parallel patterns compose efficiently is often the single most important optimization required. Previous work has considered optimizing across flat pipelines [8, 20], but does not consider nested parallelism.

To define these transformations we present the Distributed Multiloop Language (DMLL) as an intermediate language for modeling a wide range of parallel patterns. In this language we represent high-level data-parallel pat-

terns as *multiloops*, a highly flexible loop abstraction that can contain multiple fused loop-based patterns. A multiloop is a single-dimensional traversal of a fixed-size integer range that may produce zero or more values at each iteration. Each multiloop contains a set of *generators*, which capture the high-level structure of the loop body (the parallel pattern) and accumulate the loop outputs. After the loop terminates, each generator returns a result to the surrounding program. Previous work has shown how multiloops lend themselves well to advanced loop transformations, including pipeline fusion and horizontal fusion (returning multiple disjoint outputs from a single traversal) [30]. Each multiloop is typically constructed with a single generator for its body (it returns a single output), but after horizontal fusion may contain multiple generators. Here we extend and modify the previous language to capture nested pattern transformations and make it portable to heterogeneous accelerators. The current set of DMLL generators is defined in Figure 2(a). A *collect* generator accumulates all generated values and returns a collection as the result of the loop. This is general enough to implement the classic operations of `map`, `zipWith`, `filter`, and `flatMap`. A *reduce* generator performs on-the-fly reduction with a given associative operation. A *bucket-collect* or *bucket-reduce* generator collects or reduces values in buckets indexed by keys. A *bucket-collect* with only the key function defined is often called `groupBy`.

Each generator contains multiple functions that capture the key user-defined components of the computation. Keeping the components separated rather than composed into a single block in the IR makes it possible to later compose the functions in different ways to generate efficient code for multiple hardware targets. Figure 2(b) illustrates DMLL semantics by providing one possible sequential implementation. For example, to perform a *collect* operation on the CPU we can emit code that uses the *condition* block to guard appending the result of the *value function* block to an output buffer. There are many more possible implementations however. For the GPU we can instead evaluate the *condition* block for all indices up front, allocate an output buffer of the correct size, and then perform a second loop to write the result of the *value function* directly to the correct output location. The *bucket* generators are more complicated to implement but still sufficiently abstract. In particular each implementation can transparently decide to maintain the buckets by hashing (CPU) or sorting (GPU).

In this section we will write examples in pseudocode using standard data-parallel operators available in DMLL. A simple map-reduce example looks like:

```
x.map(e => math.exp(e)).reduce((a,b) => a + b)
```

We translate this language to our DMLL formalism by implementing `map` using a `Collect` generator with an always true condition and similarly `reduce` using a `Reduce` generator:

```
C = Collect_x(_)(i => math.exp(x(i)))
Reduce_C(_)(i => C(i))((a,b) => a + b)
```

We denote the always true condition as `_`. $\texttt{Collect}_x$ is shorthand for a `Collect` over the size of `x` (and similarly for $\texttt{Reduce}_C$). We can then fuse these operations with a simple rewrite rule:

```
C = Collect_s(c1)(f1)          →  Reduce_s(c1&c2)(f2(f1))(r)
Reduce_C(c2)(i => f2(C(i)))(r)
```

Note that `Collect` is more general than `Map` and therefore the rule applies to many more cases than the simple example. In fact we can further generalize this rule to apply to any generator `G` that consumes a `Collect`.

```
C = Collect_s(c1)(f1)          →  G_s(c1&c2)(k(f1))(f2(f1))(r)
G_C(c2)(i => k(C(i)))(i => f2(C(i)))(r)
```

This rule alone captures all traditional pipeline fusion optimizations in DMLL (e.g., `map-reduce`, `filter-groupBy`, etc.).

### 3.2 Nested Pattern Transformations

We use the DMLL formalism to express a set of useful transformations on nested patterns, motivated with simple code examples drawn from classic applications in the domains of machine learning and data querying. First consider the following simple aggregation query common in SQL-like languages:

```
lineItems.groupBy(item => item.status).map(group =>
  group.map(item => item.quantity).reduce((a,b) => a + b))
```

As written, it creates a bucket for each `lineItem` status. Each bucket is then summed to return an array of scalar values. However, it is not necessary to construct the buckets first and then reduce them. We can instead transform this computation into a single multiloop with a `BucketReduce` generator:

```
BucketReduce(lineItems.size)(_)(i => lineItems(i).status)
  (i => lineItems(i).quantity)((a,b) => a + b))
```

This version, while less easy to read, performs only a single traversal and reduces the quantities as they are assigned to buckets. We therefore generalize this transformation to the *GroupBy-Reduce* rule in Figure 3. This rule matches a `BucketCollect` that is consumed by a `Collect` which contains a nested `Reduce` of each bucket. The expanded lambda expression $\texttt{i => Reduce}_s\texttt{(c)(f)(r)}$ denotes a pattern match on the `Collect` function, where an enclosing context is implicitly assumed (i.e., the function may contain statements besides the `Reduce`). We can then combine the functionality contained within the `BucketCollect` and the `Reduce` into a single `BucketReduce`. The `Collect` that consumes the transformed `BucketReduce` simply expresses an identity loop over $H$, but also implicitly contains the remaining (untransformed) enclosing context of the pre-transformed `Collect`. For example, if the application instead averages each group, the division after the sum will remain in the untransformed `Collect` context. In the simple case where the context is empty, this extra identity loop is simply optimized away.

We see this pattern appear not only in many data-querying applications but in machine learning applications as well. Consider the $k$-means clustering algorithm in the second half of Figure 1. While the computation is much more complex than the first example, operates on matrices rather than flat arrays, and performs summations over vector types rather

```
G ::= Collect_s(c)(f)              : Coll[V]
    | Reduce_s(c)(f)(r)            : V
    | BucketCollect_s(c)(k)(f)     : Coll[Coll[V]]
    | BucketReduce_s(c)(k)(f)(r)   : Coll[V]

c:  Index => Boolean     condition
k:  Index => K           key function
f:  Index => V           value function
r:  (V,V) => V           reduction
s:  Integer              loop size
```

(a) DMLL Syntax

```
[[ Collect_s(c)(f): Coll[V] ]] =
  val out = new Coll[V]
  for (i <- 0 until s) {
    if (c(i)) out += f(i)
  }

[[ Reduce_s(c)(f)(r): V ]] =
  var out = identity[V]
  for (i <- 0 until s) {
    if (c(i)) out = r(out, f(i))
  }

[[ BucketCollect_s(c)(k)(f): Coll[Coll[V]] ]] =
  val out = new Coll[Coll[V]]
  val map = new Map[K,Index]
  for (i <- 0 until s) {
    if (c(i)) out(map(k(i))) += f(i)
  }

[[ BucketReduce_s(c)(k)(f)(r): Coll[V] ]] =
  val out = new Coll[V]
  val map = new Map[K,Index]
  for (i <- 0 until s) {
    if (c(i)) out(map(k(i))) = r(out(map(k(i))), f(i))
  }
```

(b) DMLL Semantics (Sequential Implementations)

**Figure 2.** DMLL Syntax & Semantics

| | | | |
|---|---|---|---|
| (GROUPBY-REDUCE) | $A$ = BucketCollect$_s$(c)(k)($f_1$)<br>Collect$_A$(_)(i => Reduce$_{A(i)}$(_)($f_2$)(r)) | $\rightarrow$ | $H$ = BucketReduce$_s$(c)(k)($f_2$($f_1$))(r)<br>Collect$_H$(_)(i => H(i)) |
| (CONDITIONAL REDUCE) | Collect$_{s_1}$(_)(i =><br>Reduce$_{s_2}$(j => g(j) == h(i))(f)(r)) | $\rightarrow$ | $H$ = BucketReduce$_{s_2}$(_)(g)(f)(r)<br>Collect$_H$(_)(i => H(h(i))) |
| (COLUMN-TO-ROW REDUCE) | Collect$_{s_1}$(_)(i => Reduce$_{s_2}$(c)(f)(r)) | $\rightarrow$ | $R$ = Reduce$_{s_2}$(c)(fv)(rv)<br>Collect$_{s_1}$(_)(i => R(i)) |
| (ROW-TO-COLUMN REDUCE) | Reduce$_{s_1}$(c)(fv)(rv:($a_1$,$b_1$) =><br>Collect$_{s_2}$(_)(i => r($a_1$(i),$b_1$(i))))<br>*iff* size($a_1$) == size($b_1$) == $s_2$ | $\rightarrow$ | Collect$_{s_2}$(_)(i => Reduce$_{s_1}$(c)(f)(r)) |

**Figure 3.** DMLL: Nested Parallel Pattern Transformations.

than scalar types, the overall outer structure of the computation in terms of parallel patterns is the same and the same rewrite rule applies. The same pattern also appears in the $k$-nearest neighbors application, which uses grouping to count the fraction of $k$ data samples per data label and select the label with the largest count. In fact the recent widespread success of the MapReduce programming model is a testament to just how often this pattern occurs in real applications.

But what if $k$-means is written as in the first half of Figure 1 rather than the second? The simple strategy of just parallelizing the outer loop can actually perform quite well in a multi-core environment, but for distributed memory it will require the entirety of `matrix` to be broadcast to every machine computing the updated cluster locations, defeating the purpose of distributing the data across multiple machines. The required transformation is not immediately obvious, and requires introducing intermediate data structures. The pattern to notice is that the application is conditionally reducing values (e.g., reducing some subset of a dataset) within an outer loop where the reduction predicate is a function of the outer loop index. This leads us to the *Conditional Reduce* transformation, which when applied yields the transformed code for $k$-means shown in Figure 5. The transformation breaks the dependency on the outer loop index and lifts the inner reduction out of the loop by pre-computing all of the partial reductions with a single pass over the dataset. Note that we are using the condition function of the original `Reduce` as the key function of the `BucketReduce` so each partial reduction in the original code is accumulated into a separate bucket. Now we traverse the large dataset `matrix` only once to precompute sums for each cluster in parallel and store them indexed by cluster. Then a second loop performs just index lookups. It is important to note that even though `ss` and `cs` are expressed as separate traversals loop fusion will merge these two `BucketReduce` loops, along with the loop that computes `assigned`, into a single traversal. In fact, after transformation and fusion take place we end up with the *exact same* optimized code as the result of applying the *GroupBy-Reduce* rule to the `groupBy` formulation of $k$-means. Furthermore we can now see a straightforward partitioning and parallelization strategy of the large dataset across the cluster.

***Optimizing for Heterogeneous Architectures*** Rather than flattening nested patterns it is often necessary to instead interchange their order. Consider the example of logistic regression. The textbook description translates into parallel patterns in a straightforward way:

```
1  val newTheta = Range(0, x.numCols) map { j =>
2    val gradient = Range(0, x.numRows) map { i =>
3      x(i,j)*(y(i) - hyp(oldTheta, x(i)))
4    }.sum
5    oldTheta(j) + alpha * gradient
6  }
```

For each feature (column) `j`, the algorithm computes a gradient of the sample dataset in a nested summation loop. Unfortunately, this implementation can be rather inefficient. In practice, the number of samples (rows) in the dataset is orders of magnitude larger than the number of features, and as such it is the samples that should be distributed. As written however every sample needs to be broadcast to all of the processors computing the summations. Therefore it is much more efficient to parallelize over the samples, traversing the big data set only once and accumulating results for each feature in parallel. The implementation requires several individual transformations: fissioning the imperfectly nested loop, interchanging the nested loops, and vectorizing the reduction. The final result is shown below:

```
1  val gradientVec = Range(0, x.numRows) map { i =>
2    Range(0, x.numCols) map { j =>
3      x(i,j)*(y(i) - hyp(oldTheta, x(i)))
4    }
5  }.sum
6  val newTheta = Range(0, x.numCols) map { j =>
7    val gradient = gradientVec(j)
8    oldTheta(j) + alpha * gradient
9  }
```

This version makes crucial use of the fact that our multiloop reduction facility is not limited to scalar values but is also able to reduce collections, using `Collect` (i.e., `zipWith`) to implement vectorized addition. Instead of constructing a vector of sums, we are computing a sum of vectors.

Despite the seemingly complex changes to the application code, we can generalize this variety of loop interchange with the simple *Column-to-Row Reduce* rule in Figure 3. In this rule `fv` and `rv` denote vectorized versions of `f` and `r` which are implemented by wrapping each scalar function with a `Collect`. Once again the transformed `Collect` contains the remaining untransformed enclosing context of the original `Collect`. Applying this rule to the logistic regression example provides a better traversal pattern when targeting multiple CPUs as well as a better way of partitioning the dataset `x`. However when considering GPU execution the original traversal order shown was actually superior since it reduces scalars rather than vectors and reducing non-scalar types on a GPU is typically very inefficient due to limited shared memory capacity. Fortunately we can also express the inverse transformation to a *Column-to-Row Reduce*, the *Row-to-Column Reduce* in Figure 3, to invert the loops to create multiple scalar reduces.

To generate code for clusters of GPUs for this example it is therefore necessary to perform a *Column-to-Row Reduce* transform for parallelization across the cluster and then apply the *Row-to-Column Reduce* when generating the GPU kernel implementation, effectively distributing over samples (rows) and then summing over features (columns) within each node. This is always possible as the two transformation rules are completely reversible, which is straightforward to show by simply substituting one into the other and reducing identities. The *Row-to-Column Reduce* allows us to generate far more efficient GPU code for the $k$-means example as well, which also reduces vectors as written. We have observed this rule to be useful in many applications in which the user wishes to somehow reduce the columns of a matrix. Examples in machine learning include ridge regression and Naïve Bayes. We refer the reader to previous work [21] for details on how to generate GPU kernels from nested patterns after these transformations have been applied.

***Discussion*** In many situations there is no single "correct" way to structure the application efficiently for heterogeneous devices, however the high-level semantics of DMLL still provide sufficient information to automatically transform to an optimized representation per device. The rules in Figure 3 represent highly recurring patterns in the applications we have studied within the domains of machine learning and data querying, and using these few generator patterns, it is easy to add new rules for other powerful optimizations. Ultimately what these transformation rules enable is a more relaxed end-user programming model. Users can write applications in a straightforward way and still obtain portable performance to multiple architectures. While in general the system can always fall back to warnings or errors that require the user to manually restructure the application, we believe that it is important that this transformation facility be extensible by DSL authors, power users, etc. to help ensure that such errors occur as infrequently as possible.

Since these transformations are meant to improve locality, the only missing piece for deciding when to apply them is calculating the original and resulting memory access patterns and knowing which is a better access pattern for the target hardware. We perform this step for distributed memory in the following section. For the GPU we always perform a Row-to-Column Reduce when possible since it enables utilizing shared memory.

## 4. Distribution Analysis and Optimization

The biggest challenge in terms of making data-parallel applications portable to heterogeneous hardware is that parallel patterns primarily abstract over the computation, but not the data structures. Many applications have both "large" collections that should be distributed and "small" collections which should only be computed by a single machine or be broadcast to every machine. It is for this reason that existing systems typically require the user to manually decide how data is distributed. Returning to our $k$-means example in Figure 1 this amounts to deciding that `matrix` should be distributed and `clusters` broadcast and then use different types and operations for each data structure even though they are both logically matrices. In this section we show how we can use straightforward analyses and the DMLL rewrite rules to automatically transform this example to the explicitly fused and distributed implementation that a programmer would normally write manually.

### 4.1 Partitioning Analysis

To decide which data structures to partition in a program we first need to know whether each input dataset should be partitioned. We obtain this information by having the user annotate each data source (e.g., the file reader operations) with this information. Alternative approaches include using JIT compilation when the inputs are available or exploiting domain knowledge. How exactly this information is obtained does not affect the rest of the discussion. Once we have this information we use a forward dataflow analysis (Algorithm 1) that uses the principle of "move the computation to the data" to decide whether or not every other data structure should be partitioned based on where and how they are produced. `Local` means the collection should be allocated entirely in one memory region and `Partitioned` means the collection should be spread across multiple memory regions.

If a parallel operation consumes only `Local` data, then its output is also `Local`. If instead the pattern consumes one or more `Partitioned` collections, the pattern itself (the code) and

**Algorithm 1** Pseudocode for Partitioning Analysis.

```
1:  DataLayout ::= Local | Partitioned
2:  Stencil ::= Interval | Const | All | Unknown
3:  transforms: List[RewriteRule]
4:  Input: layouts: Map[Sym,DataLayout]
5:  //all layouts initialized to Local unless specified otherwise
6:  procedure TRAVERSE(op)
7:      if layouts(inputs(op)) contains Partitioned then
8:          if isParallel(op) then
9:              CheckInputStencil(op)
10:             if outputIsPartitionable(op) then
11:                 layouts(output(op)) := Partitioned
12:         else
13:             if not isWhitelisted(op) then
14:                 warn()
15: procedure CHECKINPUTSTENCIL(op)
16:     stencils := ComputeInputStencils(op)
17:     if stencils contains Unknown then
18:         for transform in transforms do
19:             xformed := transform(op)
20:             newStencils := ComputeInputStencils(xformed)
21:             if not (newStencils contains Unknown) then
22:                 op := xformed return
23:         warn()
24: procedure COMPUTEINPUTSTENCILS(op)
25:     //for each array read operation within op,
26:     //use standard affine analysis to classify as one of Stencil patterns
```

```
1   val matrix = PartitionedArray.fromFile(parseMatrix)
2   val clusters = LocalArray.fromFunction(...)
3   val assigned: PartitionedArray = matrix.mapRows { row =>
4     clusters.mapRows(c => dist(row, c)).minIndex //fused
5   }
6   val newClusters: LocalArray = clusters.mapIndices { i =>
7     //note: PartitionedArray matrix accessed at dynamic indices
8     val as = assigned.filter(a => a == i)
9     val sum: LocalArray = matrix(as).sumRows //fused with line 8
10    sum.map(s => s / as.count)
11  }
```

**Figure 4.** $k$-means after the data partitioning analysis

any other local inputs will be broadcast to the partitioned data. Whether or not the output data is created as `Partitioned` is determined by the type of pattern (i.e., a `map` generates a `Partitioned` output but a `reduce` generates a `Local` output). Note that we have said nothing about the data access patterns yet. Even though the parallel pattern is partitionable, it may still require a significant amount of communication between partitions to execute. Sometimes the communication is fundamental (e.g., graph applications), but other times it can be avoided by restructuring the computation.

Applying this algorithm to $k$-means we end up with the transformed code in Figure 4, where data structures have been explicitly lowered to `PartitionedArray` or `LocalArray`. The two inputs `matrix` and `clusters` are given as `Partitioned` and `Local`, respectively. Pipeline fusion eliminates multiple intermediate data structures including the result of `mapRows` on line 4, as well as `as` and `matrix(as)` on lines 8-9. `sum` is determined to be `Local` since it is the output of a `reduce` and each `map` on lines 3, 6, and 10 inherits the same type as its input. The only problem with this lowering is the random access of partitioned `matrix` within a local loop. We can detect such problematic cases automatically using a standard read stencil analysis.

```
1   val matrix = PartitionedArray.fromFile(parseMatrix)
2   val clusters = LocalArray.fromFunction(...)
3   def assigned = i => clusters.mapRows(centroid =>
4     dist(matrix(i), centroid)).minIndex //fused
5   )
6   //lines 7-8 horizontally fused
7   val ss = bucketReduce(true, assigned, i => matrix(i), _ + _)
8   val cs = bucketReduce(true, assigned, i => 1,         _ + _)
9   val newClusters: LocalArray = clusters.mapIndices { i =>
10    val (sum: LocalArray, count: Int) = (ss(i), cs(i))
11    sum.map(s => s / count)
12  }
```

**Figure 5.** $k$-means after nested pattern transformations. This is how the program is traditionally manually written for high performance in distributed systems.

### 4.2 Read Stencil Analysis and Transformations

For every multiloop we perform a read stencil analysis for each input collection to statically detect the range of the collection the multiloop may access. We consider a simple set of access patterns that are straightforward to detect using standard affine analysis.

- `Interval` denotes that loop index `i` accesses the `ith` element of one dimension of a collection (e.g., the `ith` row of a matrix), or equivalently a contiguous interval of the flattened representation. The runtime should choose a partition that only splits the collection on the interval boundaries so that all accesses are local.
- `Const` denotes an access at a constant index. The runtime should broadcast the element to every partition.
- `All` denotes that the entire collection is consumed at each loop index. The runtime should broadcast the collection.
- `Unknown` denotes an access at some unknown element `x`. The runtime can choose any partition, but must either fully replicate the collection or must detect non-local accesses and move data between partitions dynamically.

In many cases more than one parallel operation will consume the same data structure. Therefore after computing the local stencils, we then compute a global stencil for each collection by conservatively joining its local stencils. Furthermore if two partitioned collections are consumed by an operation we mark them to be co-partitioned at runtime.

Using the stencil analysis we can statically predict if partitioning may require non-local accesses. If any stencil is `Unknown` we attempt to apply a set of rewrite rules to improve the access patterns. If any of the rewrites succeed in replacing the `Unknown` stencil with `Interval` we replace the pattern with the transformed version. The set of rewrites we currently consider are those presented in Figure 3. These rules do not overlap and we only try to apply a single rule at a time rather than an exponential combination of them, which keeps the search space linear and order-independent. If all available transformations fail, we fall back to transferring data at runtime. We mark this last case with a warning to the user since the communication may be expensive.

In the case of $k$-means, line 9 creates an `Unknown` stencil for `matrix`, which triggers the `Conditional Reduce` rule, resulting in the code shown in Figure 5. The transformation allows `assigned` to be pipeline-fused into the `bucketReduce` and both `bucketReduce`s are horizontally fused into a single traversal over the partitioned `matrix`. Furthermore the data structures read inside the loop on line 10 now have simple access stencils of `Interval` rather than `Unknown`. As discussed in Section 3.2, this implementation is an equivalent but more optimized version of the original distribution-friendly $k$-means snippet shown in Figure 1.

### 4.3 Sequential Operations

Finally we consider the case where a partitioned collection is consumed by a sequential operation. We make the conservative assumption that in addition to the well-structured parallel patterns the programming model also allows arbitrary sequential code with arbitrary effects. Therefore only the parallel patterns can be distributed and the sequential code must run in order at a single location. We believe the best practical solution is to simply handle these operations differently based on the target. If compiling for multi-core allow the operation, and if compiling for clusters disallow it. In the algorithm we mark this case with an abstract `warn()` to the user. We also relax the restriction slightly by allowing whitelisting of operations that the compiler developer knows to be safe. For example, determining the size of a collection often doesn't require dereferencing the collection data but is instead stored as an additional field. Therefore reading that field is always allowed.

## 5. Data Structure Optimization and Runtime

In this section we discuss our low-level data structure implementations and distributed runtime. We implemented DMLL on top of the Delite DSL framework in order to re-use the existing heterogeneous code generators (Scala, C++, CUDA) and compiler optimizations (e.g., code motion, common subexpression elimination (CSE)).

***Distributed Data Structures*** Using ideas from previous work [30], we implemented multiple data structure optimizations including struct unwrapping, dead field elimination (DFE), and array-of-struct to struct-of-array (AoS to SoA) transformations, which work together to reduce complex data structures to simple arrays of primitives. In addition to yielding more efficient generated code by removing indirections and enabling vectorization, these optimizations greatly simplify the stencil analysis.

Delite provides a shared memory execution model. While writes within parallel patterns are typically required to be disjoint, reads can safely be random. DMLL also supports this through `Unknown` read stencils. We extended Delite's array abstraction by adding new runtime types for distributed arrays that contain not only the local chunk of array data but additional metadata for accessing the remainder of the logical array as well. Reads at indices that are not physically present are trapped and transparently fetched from the appropriate remote location. To determine the location of the data we build a directory of index ranges to locations when the array is first instantiated and broadcast the directory to every physical instance of the logical array. When the partitioning analysis decides an array should be partitioned we lower to this new back-end implementation. The remainder we lower to the original shared-memory implementation. This ensures that we only pay the runtime cost of tracking partitioned arrays when necessary.

Many distributed systems forbid remote reads altogether for better performance guarantees, but we rely on the structured nature of parallel patterns, transformation rules, and good DSL design to limit remote reads. There are a number of important applications in which the primary distributed dataset cannot be perfectly partitioned (e.g., graph problems), thereby requiring either replication of the dataset or reading portions of the dataset from remote memories.

***Hierarchical Heterogeneous Execution*** The key insight to adding NUMA and cluster support on top of the multi-core and GPU runtime in an incremental manner is that a multiloop is agnostic to whether it runs over the entire loop bounds or a subset of the loop bounds. Therefore the cluster master can partition a given multiloop into chunks and distribute those chunks across machines. The range of each machine's chunk is chosen by combining the input data's access stencil with the input's directory to obtain the range of indices that will result in local reads (we move the computation to the data). Then each machine can further partition its chunk of work across sockets, cores, and/or GPUs using similar logic. The multi-core partitioner also provides dynamic load balancing within each machine, which provides much better scaling for irregular applications.

## 6. Experimental Results

In this section we present performance results for DMLL for multiple applications across multiple hardware configurations. We chose benchmarks from the domains of machine learning, data querying, and graph analytics that are representative of the important field of big data analytics.

The first sets of experiments were all performed on a single NUMA machine with 4 sockets, 12 Xeon E5-4657L cores per socket, and 256GB of RAM per socket. DMLL generated C++ code which was then compiled with gcc 4.8, optimization level 3. Each experiment was performed five times and we report the average of all runs.

***Baseline performance*** First we compare DMLL's sequential performance to hand-optimized C++ implementations in Table 2. For the iterative algorithms we present execution time per iteration and total execution time for the others. We observe that DMLL is within 25% of hand-optimized for every application. The optimizations required to achieve this performance are summarized in Table 2. DMLL Query 1 is even faster than the C++ version due to using a more efficient HashMap than is in the C++11 standard library. The remaining performance gaps are mostly due to the fact that like many functional languages, DMLL applications typically allocate more memory than strictly necessary, whereas

| Benchmark | Optimizations | Data Set | DMLL | C++ | Δ |
|---|---|---|---|---|---|
| TPC-H Query 1 | GroupBy-Reduce, pipeline fusion, AoS to SoA, CSE, DFE | TPC-H SF5 (5.3GB) | 1.07s | 1.82s | -41% |
| Gene Barcoding | pipeline fusion, DFE | 3.5M genes (689MB) | 0.341s | 0.311s | 9.6% |
| GDA | pipeline fusion, horizontal fusion, CSE | 500k x 100 matrix (835MB) | 8.50s | 6.92s | 23% |
| $k$-means | Conditional Reduce, Row-to-Column Reduce, pipeline fusion | 500k x 100 matrix (835MB) | 0.885s per iter | 0.843s per iter | 5.0% |
| Logistic Regression | Column-to-Row Reduce, Row-to-Column Reduce | 500k x 100 matrix (835MB) | 0.082s per iter | 0.075s per iter | 9.3% |
| PageRank | domain-specific push-pull transformation, pipeline fusion | LiveJournal [1] (1.1GB) | 0.646s per iter | 0.518s per iter | 25% |
| Triangle Counting | domain-specific push-pull transformation, pipeline fusion | LiveJournal (1.1GB) | 12.3s | 12.4s | -0.8% |

**Table 2.** Benchmarks: DMLL optimizations applied and sequential performance comparison to hand-optimized.

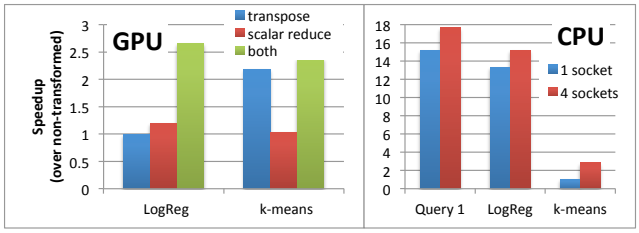


**Figure 6.** Speedups obtained by applying the nested pattern transformations for GPUs (left) and CPUs (right).

the hand-optimized applications aggressively reuse allocated memory. Pipeline fusion greatly reduces this issue but is not sufficient to remove it entirely.

***Impact of Nested Pattern Transformations*** Next we study the performance and scalability impact of the nested pattern transformations in Section 3.2 in Figure 6. DMLL often requires these transformations to properly partition data across a cluster (they are not simply performance optimizations), so we only study the performance impact in shared memory. For $k$-means the impact is very small (around 3%) when parallelizing across cores within a single socket, as both versions effectively traverse the large dataset once (albeit in different orders) and consume consecutive chunks of the dataset at a time. When parallelizing across multiple sockets however, the original shared-memory implementation stops scaling due to the more limited exposed parallelism and constrained memory bandwidth due to shuffling data across sockets, leading to nearly 3x speedup on 4 sockets for the transformed version. Logistic regression and Query 1 both run faster even in one socket as the applied transformations greatly improve cache locality and memory access patterns and are therefore always beneficial for CPUs whether the architecture is single-threaded or massively distributed.

For the GPU, both $k$-means and logistic regression perform summations over vectors as written. However DMLL's CUDA code generator can only use local shared memory for reduction temporaries when they have a fixed size (scalar types), which leads to poor performance when attempting to reduce vector types. In addition the input matrix must be transposed to enable multiple threads' memory requests to be coalesced by the GPU's memory controller. DMLL uses the *Row-to-Column Reduce* rule to transform vector reductions into scalar reductions and transposes the input matrix when transferring it to the GPU to optimize kernel access patterns. The read stencil information used to distribute the matrix also tells DMLL how to transpose it. For logistic regression both the *Row-to-Column* and transpose transformations must be combined for maximum performance, but for $k$-means transposing provides most of the performance improvement as it speeds up the entire operation, not just the reduction.

### 6.1 NUMA Scalability

In Figure 7 we study NUMA performance compared to popular alternative frameworks: Spark [35] for the machine learning and data querying apps and PowerGraph [13] for the graph analytics. We also compare to Delite [7] without DMLL improvements. For each application DMLL applies all of the compiler optimizations in Table 2 automatically. For the other systems we ported each application directly to their programming model and performed all possible optimizations manually. However not all of DMLL's optimizations were possible to express. In particular, DMLL transparently performs AoS to SoA transformations, which enables storing the table in Query 1 as multiple arrays of primitives rather than a single array of objects. Such a transformation is not possible in Spark because each field of the output record is produced from multiple fields of the input record, and therefore the input collection cannot simply be split into an RDD per field. Also note that performing NUMA-aware memory allocations is not currently possible within the JVM, which severely limits the scalability of JVM systems on this machine.

With DMLL we first augmented the Delite runtime to pin threads to physical cores in a locality-aware manner and to create thread-local heaps. These two items are sufficient to ensure that thread-local data remains local (DMLL Pin-only). We then added the NUMA-aware features which in addition to pinning use the analyses in Section 4 to partition large arrays across multiple memory regions (DMLL). In general reading data from all memory banks simultaneously is necessary to maximize memory bandwidth. The generated application code is essentially identical except for how each array is physically allocated. For the pin-only version each array is allocated entirely within a single socket's memory, with the socket determined by which thread calls `malloc` for that array. For the NUMA-aware version, partitioned arrays are instead partially allocated across every socket's memory. As shown in Figure 7 the majority of benchmarks scale reasonably well up to two sockets but then stop scaling for Delite, while the DMLL versions continue to scale. For the first two apps the NUMA-aware version scales best as most

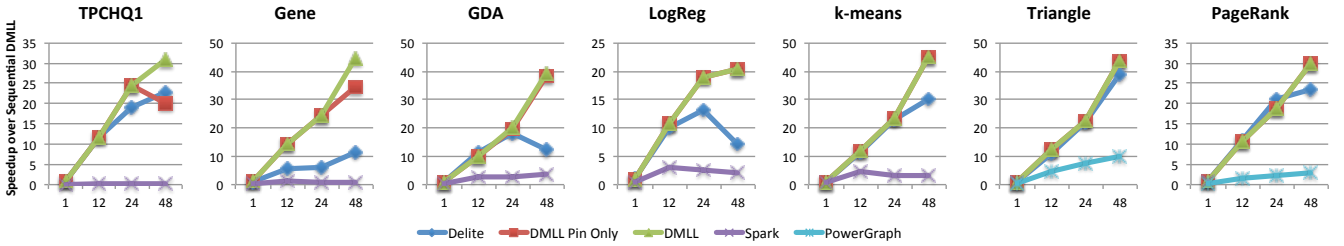


**Figure 7.** Performance and scalability comparison of DMLL, Delite, Spark, and PowerGraph on a 4 socket machine.

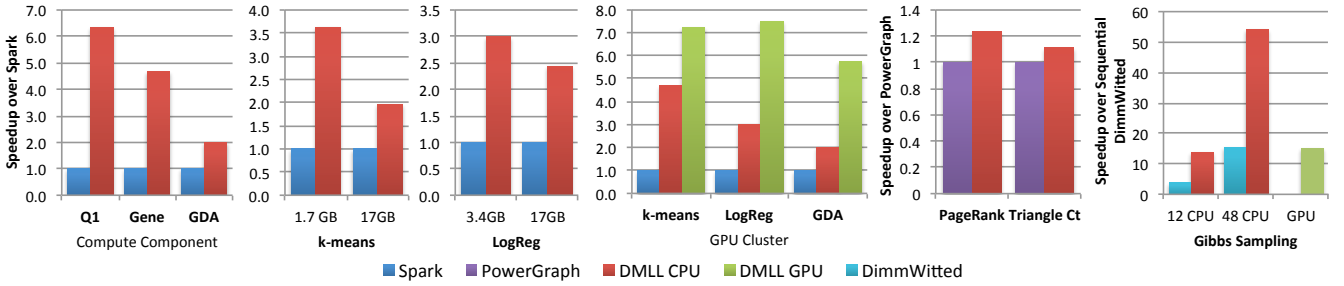


**Figure 8.** Performance of DMLL compared to manually optimized implementations in alternative systems. The left three graphs were run on the 20 node Amazon cluster, and the next two on the 4 node GPU cluster connected within a single rack.

of the memory accesses are to the partitioned data structures, therefore partitioning increases the available bandwidth. For the next three apps most of the computation is within nested loops over thread-local data structures, so pinning is sufficient to maximize locality and memory bandwidth. For Triangle Counting even the baseline multi-core version scales well as the working sets tend to fit in cache, thereby hiding NUMA issues. PageRank benefits from pinning, but requires significant inter-socket communication, which limits the overall scalability and bandwidth benefit of partitioning.

We also show that all DMLL versions are significantly faster than the Spark and PowerGraph implementations. The single-threaded performance improvements are largely due to the compiler optimizations discussed and highly efficient code generation of Delite compared to library implementations, while the improved scalability is due to NUMA-aware optimizations and a low overhead runtime.

## 6.2 Heterogeneous Clusters

The next set of experiments were performed on Amazon EC2 using 20 m1.xlarge instances. Each machine contained 4 virtual cores, 15GB of memory, and 1Gb Ethernet connections. DMLL generated Scala code and ran entirely in the JVM to provide the most fair comparison with Spark. We used Java 1.6.0b24 with default options. The results are shown in Figure 8. We do not show Delite performance here as it does not scale to multiple machines. For each application DMLL uses the same parallelization and distribution strategy as Spark, however in the Spark version this is enforced manually and in DMLL it is performed automatically by the partitioning analysis. Spark allows users to control whether collections are written to disk or kept in memory. In these experiments we kept all data entirely in memory, sacrificing fault tolerance for maximum performance.

The first two benchmarks (Query1 and Gene) only iterate over the primary dataset once, so in isolation they are fundamentally I/O bound, and both systems take roughly the same amount of time to read data from disk. However, we separate the input loading time from the computation time to demonstrate the potential speedup if the dataset being queried already resides in memory. GDA is similar but iterates over its dataset twice. $k$-means and logistic regression however both iterate over the primary dataset many times, and therefore the initial I/O cost is amortized over a large number of iterations. Overall the performance difference between DMLL and Spark is much smaller on this configuration, comparable to the single-threaded performance difference between the two systems, as each machine has very few resources and both systems distribute the data and work across machines identically. Switching to a cluster of higher-end machines with more CPUs increases the gap, as seen in the results on the GPU cluster.

For the GPU and graph experiments we switched to a smaller high-end cluster of 4 nodes, each with 12 Intel Xeon X5680 cores, 48GB of RAM, an NVIDIA Tesla C2050 GPU, and 1Gb Ethernet within a single rack. We implemented the graph benchmarks in OptiGraph, a graph analytics DSL built on top of DMLL that uses the same domain-specific transformations that have been shown previously [16] to transform applications between a pull model of computation (common in shared memory) to a push model of computation (common in distributed systems) based on the hardware target. For these experiments we compare against PowerGraph as it has been shown to be substantially faster than Spark for graph analytics applications [13]. Both systems implement the same high-level model of computation, namely pushing the required data to local nodes and then performing the computation locally. As such both systems transfer the data across the network in roughly the

same time. The computation portion runs faster in DMLL due to the low-level nature of DMLL's generated code compared to the PowerGraph library implementation, however this is largely overshadowed by the communication, leading to comparable overall performance. Comparing the previous NUMA results, we see the usefulness of large memory NUMA machines for graph analytics. For the cluster implementation of both systems, most of the execution time is spent transferring the graph over the network and building a local cache of the received remote data, which is slower than just using a single machine. In a NUMA machine however, accessing remote portions of the graph is still relatively fast, and therefore the efficiency of the generated code has a large impact on the overall performance and scalability.

Finally we show DMLL's GPU performance. GDA is well-suited to GPU execution and runs over 5x faster than Spark without any additional optimizations beyond what DMLL already performed for the CPU. $k$-means has been previously shown to perform well on GPUs when manually optimized, but, as discussed in Section 6, generating efficient code automatically requires multiple transformations. Without these transformations the GPU code performs worse than DMLL's CPU code, but applying them provides 7.2x speedup over Spark. The same transformations are required for logistic regression. Logistic regression has much lower arithmetic intensity than $k$-means however, and so the improved execution time comes largely from the GPU's higher memory bandwidth rather than compute power. The other benchmarks we studied generally do not perform very well on GPUs due to the fact that the cost of moving the data to the GPU is often more expensive than just computing the result on the CPU. For iterative algorithms however this is not the case. Just as the cost of reading the data from disk is amortized over many iterations, so is the initial cost of moving the data to the GPU.

### 6.3 Application Case Study: Gibbs Sampling

In this section we apply DMLL to a real-world application that is actively being researched and used in commercial data analytics engines today. Gibbs sampling on factor graphs is a popular method to solve Bayesian statistical problems including price modeling and information extraction that is used in several commercial data analytics engines. It is one of the core components of DeepDive [28], which uses the DimmWitted [36] implementation.

The application presents a unique challenge for parallel frameworks as the optimal parallelization strategy is different for cores within a socket compared to across sockets. The algorithm exploits the fact that threads within a socket can communicate very inexpensively (through last-level cache) by performing Hogwild! [27] updates, in which threads read and write the shared output model asynchronously. This execution model quickly stops scaling beyond a single socket however due to the much higher costs of cache coherency. Therefore the implementation creates a distinct model for each socket, samples each model independently, and then averages the samples to produce the final output. Expressing this algorithm using data-parallel constructs fundamentally requires the system to be able to exploit nested parallelism, which is lacking in many systems. DMLL distributes the outer parallelism over models across threads on different sockets, and then for the inner parallelism within a model it uses multiple threads within a socket.

As shown in Figure 8, using this strategy both DMLL and DimmWitted scale nearly linearly across sockets on the 4 socket machine. However the DMLL version is over 2x faster sequentially and 3x faster with multi-core due to the efficiency of our generated code that uses unwrapped arrays of primitives, while the hand-written version contained more pointer indirections in the factor graph implementation for the sake of user-friendly abstractions. DMLL's GPU implementation is limited by the random memory accesses into the factor graph, which greatly reduces the achievable bandwidth.

## 7. Conclusion

In this paper we introduced the Distributed Multiloop Language (DMLL), an intermediate language based on common parallel patterns that enables targeting heterogeneous distributed hardware from a rich implicitly parallel programming model. We showed straightforward analyses for partitioning data structures and powerful transformations for restructuring applications to target heterogeneous devices. We presented experimental results across multiple application domains and hardware platforms demonstrating that this model is sufficient to generate optimized target-specific implementations that are within 25% of hand-optimized C++ and greatly exceed the performance of explicitly distributed versions in other high-level frameworks. We demonstrated speedups of up to 11x over PowerGraph and 40x over Spark.

## Acknowledgments

We are grateful to the anonymous reviewers for their comments and suggestions. This work is supported by DARPA Contract-Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grant, BIG-DATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF Grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Dept. of Energy- Pacific Northwest National Lab (PNNL)- Integrated Compiler and Runtime Autotuning Infrastructure for Power, Energy and Resilience-Subcontract 108845; NSF Grant- EAGER- XPS:DSD:Synthesizing Domain Specific Systems-CCF-1337375; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Huawei, Intel, NVIDIA, SAP Labs. Authors also acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

## References


[1] Livejournal social network. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[2] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. OOPSLA. ACM, 2010.

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. Springer Verlag, 2010.

[4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 1996. doi: http://doi.acm.org/10.1145/227234.227246.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. 2008.

[6] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69:123–178, July 2005. ISSN 0169-2968.

[7] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. PLDI. ACM, 2010. ISBN 978-1-4503-0019-3.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 2005.

[10] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995. ISSN 0164-0925.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, OSDI, pages 137–150, 2004.

[12] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI '12, 2012.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[14] T. Grosser, A. Groesslinger, and C. Lengauer. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[15] J. Hoberock and N. Bell. Thrust: C++ template library for CUDA, 2009.

[16] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. CGO, 2014.

[17] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao. Spartan: A distributed array framework with smart tiling. USENIX Association, 2015.

[18] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. SIGMOD. ACM, 2009.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys. ACM, 2007.

[20] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[21] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on gpus. IEEE Micro, 2014.

[22] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002. ISSN 0362-1340.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD '10. ACM, 2010.

[24] B. L. Massingill, T. G. Mattson, and B. A. Sanders. A pattern language for parallel application programs. In *Euro-Par 2000 Parallel Processing*, pages 678–681. Springer, 2000.

[25] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost?

[26] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. SOSP '13, 2013.

[27] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24:693–701, 2011.

[28] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28, 2012.

[29] A. Prokopec, P. Bagwell, and T. R. abd Martin Odersky. A generic parallel collection framework. Euro-Par, 2010.

[30] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[31] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. ACM, 2013.

[32] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. ECOOP, 2013.

[33] T. L. Veldhuizen and J. G. Siek. Combining optimizations, combining theories. Technical report, Indiana University, 2008.

[34] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI, 2011.

[36] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 2014.