Dandelion: a Compiler and Runtime for Heterogeneous Systems

Christopher J. Rossbach, Yuan Yu, Jon Currey, and Jean-Philippe Martin
Microsoft Research

Abstract

Computer systems increasingly rely on heterogeneity to achieve greater performance, scalability and energy efficiency. Because heterogeneous systems typically comprise multiple execution contexts with very different programming abstractions and runtimes, programming them remains extremely challenging.

Dandelion is a system designed to address this programmability challenge for data-parallel applications. Dandelion provides a unified programming model for heterogeneous systems that span a diverse array of execution contexts including CPUs, GPUs, FPGAs, and the cloud. It adopts the .NET LINQ (Language INtegrated Query) approach, integrating data-parallel operators into general purpose programming languages such as C# and F# and therefore provides an expressive data model and native language integration for user-defined functions. This enables programmers to write applications using standard high-level languages and development tools, independent of any specific execution context.

Dandelion automatically and transparently distributes the data-parallel portions of a program to the available computing resources, including compute clusters for distributed execution and the CPU and GPU cores of individual compute nodes for parallel execution. To enable the automatic execution of .NET code on GPUs, Dandelion cross-compiles .NET code to CUDA kernels and uses a GPU dataflow runtime called EDGE to manage GPU execution.

This paper describes the design and implementation of the Dandelion compiler and runtime, focusing on the distributed CPU and GPU implementation. We report on our evaluation of the system using a diverse set of workloads and execution contexts.

1 Introduction

The computing industry is experiencing a major paradigm shift to heterogeneous systems in which general-purpose processors, specialized cores such as GPUs and FPGAs, and the cloud are combined to achieve greater performance, scalability and energy efficiency. Such systems typically comprise multiple execution contexts with potentially very different programming abstractions and runtimes: this diversity gives rise to a number of programming challenges such as managing the complexities of architectural heterogeneity, resource scheduling, and fault-tolerance. The need for programming abstractions and runtime support that allow programmers to write applications in a high-level programming language portable across a wide range of execution contexts is increasingly urgent.

In Dandelion, we consider the research problem of writing data-parallel applications for heterogeneous systems. Our target systems are compute clusters whose compute nodes are equipped with multi-core CPUs and GPUs. We envision a typical Dandelion cluster to contain up to 64 moderately powerful computers, each having 32-128GB memory, 8-32 CPU cores, and 2-4 GPUs. Such a cluster can have aggregated compute resources of more than 100,000 cores and 10TB memory, and represents an attractive and affordable computing platform for many demanding applications such as large-scale machine learning and computational biology. The goal of our research is to make it simple for programmers to write high performance applications for this kind of heterogeneous system, leveraging all the resources available in the system.

Dandelion provides a "single machine" abstraction: the programmer writes sequential code in a high-level programming language such as C# or F#, and the system automatically executes it utilizing all the parallel compute resources available in the execution environment. Achieving this goal requires substantial work at many layers of the technology stack including programming languages, compilers, and distributed and parallel runtimes. Dandelion addresses two primary challenges. First, the architectural heterogeneity of the system components must be well encapsulated: the programming model must remain simple and familiar while insulating the programmer from challenges arising from the presence of diverse execution models, memory models

and ISAs. Second, the system must integrate multiple runtimes efficiently to enable high performance for the overall system.

At the programming language level, the language integration approach has enjoyed great success [82, 25, 83], and provides the most attractive high-level programming abstraction. We use LINQ [5], a general language integration framework, as the programming model. In order to run the same code in different architectural contexts, Dandelion introduces a new general-purpose cross compiler framework that enables the translation of .NET byte-code to multiple back-ends including GPU, FPGA, and vector processors. In this paper, we focus on our implementation for GPUs. The FPGA backend [1] was implemented by others and reported elsewhere.

At the runtime level, architectural heterogeneity demands that the system's multiple execution contexts seamlessly interoperate and compose. We adopt the dataflow graph as the execution model for computation. In the dataflow model, vertices of a graph represent computation and the edges represent data or control communication channels. Dandelion comprises several execution engines: a distributed cluster engine, a multi-core CPU engine, and a GPU engine. Each engine represents its computation as a dataflow graph, and the dataflow graphs of all the engines are composed by asynchronous communication channels to form the global dataflow graph for a Dandelion computation. Data transfers among the execution engines are automatically managed by Dandelion and completely transparent to the programmer. This design offers great composability and flexibility while making the parallelism of the computation explicit to the runtimes.

A large number of research efforts have goals similar to those of Dandelion: building scalable heterogeneous systems [46, 36, 80]. Most of these systems retrofit accelerators such as GPUs, FPGAs, or vector processors into existing frameworks such as MapReduce or MPI. With Dandelion, by contrast, we take on the task of building a general framework for an array of execution contexts. We believe that it is now time to take a fresh look at the entire software stack, and Dandelion represents a step in that direction. This paper makes the following contributions:

- The Dandelion prototype demonstrates the viability of using a rich object-oriented programming language as the programming abstraction for data-parallel computing on heterogeneous systems.
- We build a general-purpose compiler framework that automatically compiles a data-parallel program to run on distributed heterogeneous systems. Multiple back-ends including GPU and FPGA are supported.
- We validate our design choice of treating a heterogeneous system as the composition of a collection of dataflow engines. Dandelion composes three dataflow engines: cluster, multi-core CPU, and GPU.
- We build a general purpose GPU library for a large set of parallel operators including most of the relational operators supported by LINQ. The library is built on top of EDGE [2], a high performance dataflow engine for GPUs.

The remainder of this paper is organized as follows. Section 2 introduces the programming model for Dandelion. Section 3 and 4 describe the design and implementation of Dandelion, respectively. In Section 5, we evaluate the Dandelion system using a variety of example applications. Section 6 discusses related work and Section 7 concludes.

2 Programming Model

The goal of Dandelion is easy programming of distributed heterogeneous systems. To this end, Dandelion adopts a single high-level programming abstraction that insulates the developer from the complexity arising from the architectural and runtime differences of the execution contexts within a heterogeneous system. To support data-parallel computation, Dandelion embeds a rich set of data-parallel operators using the LINQ language integration framework. This leads to a programming model in which the developer writes programs using a single unified programming frontend of C# or F#. This section provides a high-level overview of this programming model.

2.1 LINQ

LINQ [5] is a .NET framework for language integration. It introduces a set of declarative operators to manipulate collections of .NET objects. The operators are integrated seamlessly into high level .NET programming languages, giving developers direct access to all the .NET libraries and user-defined application code. Collections manipulated by LINQ operators can contain objects of any .NET type, making it easy to compute with complex data such as vectors, matrices, and images.

LINQ operators perform transformations on .NET data collections, and LINQ queries are computations formed by composing these operators. Most

```
1  IQueryable<Vector> OneStep(IQueryable<Vector> vectors,
2                             IQueryable<Vector> centers) {
3    return vectors
4      .GroupBy(v => NearestCenter(v, centers))
5      .Select(g => g.Aggregate((x, y) => x + y)/g.Count());
6  }
7
8  int NearestCenter(Vector vector,
9                    IEnumerable<Vector> centers) {
10   int minIndex = 0;
11   double minValue = Double.MaxValue;
12   int curIndex = 0;
13   foreach (Vector center in centers) {
14     double curValue = (center - vector).Norm2();
15     if (minValue > curValue) {
16       minValue = curValue;
17       minIndex = curIndex;
18     }
19     curIndex++;
20   }
21   return minIndex;
22 }
```

Figure 1: A simplified k-means implementation in LINQ.

LINQ operators are familiar relational operators including projection (`Select`), filters (`Where`), grouping (`GroupBy`), aggregation (`Aggregate`), and joins (`Join`). LINQ also supports set operations such as `Union` and `Intersect`.

The base type for a LINQ collection is `IEnumerable<T>`, representing a sequence of .NET objects of type `T`. LINQ also exposes a query interface `IQueryable<T>` (a subtype of `IEnumerable<T>`) to enable deferred execution of LINQ queries with a custom execution provider. Dandelion is implemented as a new execution provider that compiles LINQ queries to run on a distributed heterogeneous system.

Figure 1 shows as an example a simplified version of k-means written in C#/LINQ. The k-means algorithm is a classical clustering algorithm for dividing a collection of vectors into k clusters. It is a simple, iterative computation that repeatedly performs `OneStep` until some convergence criterion is reached.

At each iteration, `OneStep` first groups the input vectors `vectors` by their nearest center, and computes the new center for each new group by computing the average of the vectors in the group. `OneStep` invokes the user-defined function `NearestCenter` to compute the nearest center of a vector. To run k-means on a GPU, Dandelion cross-compiles the user-defined functions to CUDA and uses the resulting GPU binaries to instantiate primitives that are common to the underlying relation algebra (these primitives are treated in detail in Section 4.2).

2.2 Dandelion Extension

The overriding goal of Dandelion is to enable the automatic execution of programs such as the one shown in Figure 1 on distributed heterogeneous systems *without any modification*. The main challenge is to preserve the LINQ programming model, rather than designing new language features. Our approach is to extend LINQ with a very small number of new features that we believe are essential.

Dandelion extends LINQ with two new operators. The first operator is `source.AsDandelion(gpuType)`. It turns the LINQ collection `source` into a Dandelion collection, enabling any LINQ query using it as input to be executed by Dandelion. For example, to run the above k-means program using Dandelion, we add a call to `AsDandelion` to the two inputs as follows:

```
vectors = vectors.AsDandelion();
centers = centers.AsDandelion();
while (!Converged(centers, oldCenters)) {
    oldCenters = centers;
    centers = OneStep(vectors, centers);
}
```

The `AsDandelion` function can also take an optional argument `gpuType` that specifies the GPU type of the objects in the input collection. Dandelion can improve the performance of computations on the GPU when it knows ahead of time that it is operating on a sequence of fixed-length records. This argument inform the Dandelion runtime of the record size. In k-means, the vectors are all the same size; if we know the size is, for example, 100, we would write `AsDandelion(GPU.Array(100))`. Section 4 explains how this information is used to generate a more efficient execution plan for the GPU.

The second operator added in Dandelion is `source.Apply(f)`, which is semantically equivalent to `f(source)` but its execution is deferred, along with the other LINQ operators. At the cluster level, the input data is partitioned across the cluster machines, and the function `f` is applied to each of the partitions independently in parallel. At the machine level, the function `f` runs on either CPU or GPU, depending on its implementation. The primary use of `Apply` is to integrate existing CPU and GPU libraries such as CUBLAS [63] and MKL [43] into Dandelion, bringing the primitives defined in those libraries to the same level of abstraction. Unlike all the other LINQ operators, for some choices of `f` the answer may depend on how the data is partitioned, so the programmer should be mindful.

Dandelion automatically "kernelizes" the user-defined .NET functions invoked by the LINQ opera-

tors to arrive at GPU implementations. Some functions may already have an existing high-performance GPU implementation, which the developer would definitely like to use. In Dandelion, the developer can add an `Accelerated(deviceTarget, dllName, opName)` annotation to a .NET function to override its cross-compilation by Dandelion. The annotation tells the system that the current .NET function can be replaced by the function `opName` in the DLL `dllName` on the computing device `deviceTarget`. This is similar to the .NET PInvoke and Java JNI mechanisms.

2.3 Limitations

Dandelion imposes some restrictions on programs. First, all the user-defined functions invoked by the LINQ operators must be side-effect free, and Dandelion makes this assumption about user programs without either static or runtime checking. Similar systems we are aware of make the same assumption.

Second, Dandelion depends on low-level GPU runtimes such as CUDA. These runtimes have very limited support for device-side dynamic memory allocation, and even when supported, the performance can be unpredictable and often very poor. Consequently, we choose to not kernelize any .NET function that contains dynamic memory allocation, meaning that such functions will be executed on CPUs. When cross-compiling .NET code, Dandelion uses static analysis to infer the size of .NET memory allocations in the code and turns fixed sized allocations to stack allocations on GPUs. In our k-means example above, Dandelion is able to infer the sizes for all the memory allocations in the code, given the size of the input vectors.

3 System Architecture

Figure 2 shows a high-level architectural view of the Dandelion system. Dandelion compiles LINQ programs to run on on a heterogeneous compute cluster comprising CPUs and GPUs. There are two main components: the Dandelion compiler generates the execution plans and the worker code to be run on the CPUs and GPUs of cluster machines, and the Dandelion runtime uses the execution plans to manage the computation on the cluster, taking care of issues such as scheduling and distribution at multiple levels of the cluster. An execution plan describes a computation as a dataflow graph. Each vertex in the dataflow graph represents a fragment of the computation and the edges represent communication chan-

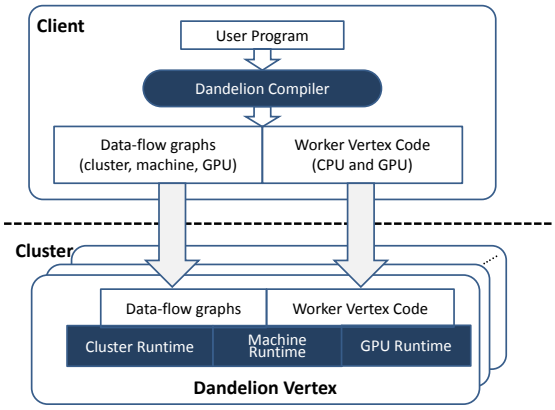

Figure 2: The Dandelion system architecture.

nels. In this section we provide a high-level overview of the system, highlighting its key features.

3.1 Dandelion Compiler

Consider the k-means example shown in Figure 1. To run it on a compute cluster comprising CPUs and GPUs, the Dandelion compiler performs the following transformations on the LINQ program.

The compiler first applies query rewrite rules to arrive at a version of the query that is optimized for parallel execution on the distributed compute substrate. It then generates a cluster-level execution plan that is used to distribute parts of the computation to the cluster machines. It also assigns application code fragments to the vertices of the execution plan. Because this aspect of the system is similar to existing systems [82, 83, 39] we do not elaborate on it further.

Cluster-level vertices are processes running on cluster machines, whose execution is controlled by the machine execution plan. Vertices of this machine-level execution plan can run either on the CPUs or the GPUs of the machine. The compiler must next decide which vertices are to run on CPUs and which on GPUs. For vertices that can be run on GPUs, the compiler generates a GPU execution plan for each of them, and cross-compiles all the involved .NET functions and types to CUDA. In our k-means example, the entire computation can be offloaded to GPU(s). Dandelion generates CUDA implemenations for the `NearestCenter` function, along with the other supporting functions such as `Norm2` and other vector arithmetic operators.

Dandelion implements a GPU library consisting of a large collection of primitives which are the building blocks to construct the GPU execution plan. Some

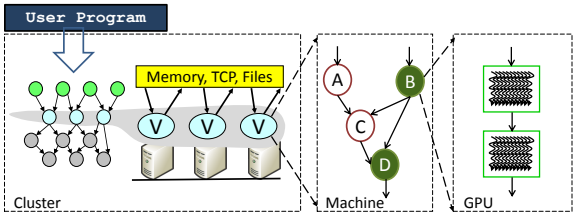

Figure 3: A global view of a Dandelion dataflow graph.

of the primitives are implemented as generic templates. For example, the primitive for `Select` is parameterized by both the data type and the processing function, which Dandelion instantiates using the cross-compiled CUDA code. The primitives are designed to be composable in the sense that new composite primitives can be formed by composing more basic primitives. The library supports relational operators and a number of well-known basic parallel operators and data structures such as parallel scan (inclusive and exclusive prefix sum), hash-table and sorting. The GPU execution plan is simply a composition of these primitives with proper generic template instantiations. A complete list of primitives from Dandelion is shown in Table 1.

3.2 Dandelion Runtime

The execution plans and worker code generated by the Dandelion compiler contain all the information necessary to run the computation on the cluster. After they are generated, Dandelion automatically distributes them to the machines in the cluster. The Dandelion runtime is composed of three cooperating execution engines, each managing one layer of the execution structure.

Figure 3 shows the three execution layers in the system. At runtime, the computation at each layer is represented as a dataflow graph, and the composition of those dataflow graphs forms the global dataflow graph for the entire computation. The cluster execution engine manages the distributed aspect of the execution and is responsible for cluster level scheduling and distribution. After it assigns a vertex to a cluster machine, the machine execution engine takes control. Each vertex of its data-flow graph is run either on CPU or GPU. This machine execution engine manages the execution of CPU vertices and delegates the execution of the GPU vertices to the GPU execution engine.

Dandelion manages machine-machine and CPU-GPU data communication automatically. The compiler generates efficient serialization code for all the data types involved in both cases. All data communication is implemented using asynchronous channels.

4 Implementation

This section provides implementation details of the components of the system, including various layers of the compiler and runtime. Dandelion is a large and complex system and it leverages a number of existing technologies that have been published elsewhere. We therefore focus our attention mainly on the novel aspects of the system. We continue to use the k-means example in Figure 1 as a running example.

4.1 GPU Compiler and Code Generation

A key feature of Dandelion is to enable automatic execution on GPUs for programs written in C#. As explained in Section 3, the Dandelion compiler relies on a library of generic primitives to construct the execution plans and a cross compiler to translate user-defined types and lambda functions from .NET to GPU. This compilation step takes .NET bytecode as input and outputs CUDA source code. Working at the bytecode level allows us to handle binary-only application code.

We build our cross-compiler using the Common Compiler Infrastructure (CCI) [3]. CCI provides the basic mechanism of reading the bytecode from a .NET assembly into an abstract representation (AST) that is amenable for analysis tools. To perform the cross-compilation, Dandelion maps all referenced C# object types to CUDA struct types, translating methods on those objects into GPU kernel functions. The compiler must also generate serialization and deserialization code so that objects in managed space can be translated back and forth between C# and GPU-compatible representations as Dandelion performs data transfers between CPU and GPU memory. Figure 4 shows CUDA code generated by Dandelion for the user-defined function `NearestCenter` in the k-means example. Functions called by `NearestCenter` such as `Norm2` are also translated by recursively walking through the call graph.

While Dandelion's cross compiler is quite general, there are limitations on its ability to find a usable mapping between C# code and GPU code. The primary constraint derives from the presence of dynamic memory allocation, which typically has very poor performance on GPUs. Dandelion therefore

converts dynamic allocation to stack allocation in cases where the object size can be unambiguously inferred, and falls back to executing on the CPU when it cannot infer the size.

Converting from dynamic (heap) allocation to stack allocation requires Dandelion to know the allocation size statically. Dandelion employs standard static analysis techniques to infer the allocation size at each allocation site. It makes three passes of the AST. The first pass collects summary information for each basic block and each assignment statement. The second pass performs type inference and constant propogation through the code using the summary. The final pass emits the CUDA code.

In languages such as C# and Java, array size is inherently dynamic, so Dandelion will fail to convert any functions involving array allocations. Dandelion provides an annotation that allows the programmer to specify the size when the array actually has a known fixed size. For example, the vector size in k-means is known and fixed: annotation on the inputs of k-means enables Dandelion to convert the entire k-means computation to run on GPU.

A related limitation of GPU computing is the difficulty of handling variable-length records. To address this problem, Dandelion treats a variable-length record as an opaque byte array, coupled with metadata recording the total size of the record and the offset of each field. In the generated CUDA code, field accesses are converted into kernel functions that take the byte array and metadata as arguments and return a properly typed CUDA record. This is a very general scheme and works for nested data types, but the overhead of the metadata and additional wrappers functions can cause performance problems. So Dandelion resorts to this general mapping only when it fails to statically infer the size of the record type.

4.2 GPU Primitive Library

To construct local dataflow graphs for each node in the computation, Dandelion takes advantage of a per-operator canonical form taken on by dataflow graphs for relational operators: these canonical forms are largely independent of the underlying types and user-defined functions. A select operation, for example, can always be expressed as a dataflow graph with primary input and output channels, and optionally, some number of additional input channels for auxilliary data sources referenced by the user-defined selector function. Because this graph structure is independent of the select function and data types Dandelion can decouple construction of local dataflow graphs from the cross compilation.

```
1  __device__ __host__ int
2  NearestCenter(KernelVector point,
3                KernelVector *centers,
4                int centers_n) {
5      KernelVector local_6;
6      int local_0 = 0;
7      double local_1 = 1.79769313486232E+308;
8      int local_2 = 0;
9      int centers_n_idx = -1;
10     goto IL_0041;
11     {
12         IL_0018:
13         KernelVector local_3 = centers[centers_n_idx];
14         local_6 = op_Subtraction_Kernel(local_3, point);
15         double local_4 = ((double)(Norm2_Kernel(local_6)));
16         if (((local_1) > (local_4))) {
17             local_1 = local_4;
18             local_0 = local_2;
19         }
20         local_2 = ((local_2) + (1));
21         IL_0041:
22         if (((++centers_n_idx) < centers_n)) {
23             goto IL_0018;
24         }
25         goto IL_0058;
26     }
27     IL_0058:
28     return local_0;
29 }
```

Figure 4: CUDA code generated by the Dandelion compiler for the C# code in the k-means workload.

Consequently, the parallelization strategy for each operator is primarily the concern of the primitive implementation rather than of compiler, yielding a successful general approach to parallelization. The primitives in Dandelion are chosen such that relational operators may be composed from a small set of primitives. Examples include parallel scan (prefix sum), hash tables, and sorts. Table 1 shows all the primitives implemented by Dandelion, indicating their relationship to relational operators in LINQ.

Figure 5 shows the local dataflow graph for k-means. The graph includes a subgraph that implements incremental grouping (`buildHT`, `keymap`, `groupsizes`, `prefixsum`, and `shuffle` primitives), and a subgraph that implements the aggregation (`aggregate`, `accumulate`, and `reduce`).

Mapping the groupby operation to the parallel architecture leverages its fundamental similarity to shuffling: given an input stream of records (potentially broken into multiple discrete chunks), the records for which the key extractor function returns the same value must be shuffled into contiguous regions in the final output. To perform this operation in parallel, we map each hardware thread to a single input record. [1] To find the output offset of each

[1] In practice, we can take advantage of memory coalescing support on the GPU by mapping each thread to multiple input records at some fixed stride apart, but to understand the function of the primitive, it suffices to think of all input records being processed in parallel, each by a unique hardware

primitive	GroupBy	Aggregate	Select	Join	Where	OrderBy	SelectMany	Distinct	Zip	Description
groupsizes	X			X	X		X	X		computes downstream collection size/offsets
bnlj				X						block-nested loop join
compact					X			X		compacts a non-unique list into a unique one, given a stencil
predicate					X			X		compute a stencil of predicate values
apply			X		X			X		apply a function to each input member
buildHT	X			X				X		build a hash table given a key extractor function
destroyHT	X			X				X		destroy hash table
distinctSort								X		sort-based distinct primitive
keymap	X			X						map from input to key identifier
reduce	X									final reduction in aggregated group by primitive
distinctHT								X		hash-table based distinct primitive
hashjoin				X						hash join
shuffle	X			X			X			output scatter/gather for group, join
accumulate	X									recursive accumulator for aggregation
aggregate	X	X								accumulator for aggregation
grouper	X									simple groupby primitive
kvsort	X					X		X		key value sort (order by)
prefixsum	X			X	X		X	X		prefix sum
sort	X					X		X		sort primitive
stencil								X		compute a stencil of "first" (unique) elements
zip									X	zip primitive

Table 1: The primitive building blocks implemented by the Dandelion primitive library and the relational operators each one is used to support.

record in the input, we need to know the start index of each group in the output sequence, which in turn requires knowledge of the number of groups, as well as the number of elements in each group. Acquisition of these data requires that the grouper process the input in stages. In the first stage the `buildHT` primitive uses a lock-free hash table to count the number of unique key extractor return values (and consequently the number of groups), and we map each unique key to a synthetic integer-valued group identifier. Each thread runs the key extractor function on its input record and attempts to insert that return value as well as the key itself into a hash table. The hash-table implementation uses a CAS operation to ensure that only the first thread attempting to insert a particular value will succeed; upon successful insertion, we rely on atomic increment support on the GPU to increment the number of known unique groups; the result of this increment becomes the group identifier for the new group. The hash-table output of the `buildHT` primtive is used by the `keymap` primitive to build a map from each record in the current input chunk to its group identifier. The hash table is then returned on a back edge of the dataflow graph to be used by subsequent invocations of `buildHT` to the next chunk of input. The `keymap` output map from input to group identifier `keymap` is then used to compute the number of elements in each group in the `groupsizes` primitive, which uses the "nkeys" output from `keymap` to allocate an array of counters equal to the number of unique keys. Each thread in the `groupsizes` primitive uses an atomic increment at the array index corresponding to the group identifier for its input element. The output of `groupsizes` is an array of group sizes for the current input chunk. The `prefixsum` primitive accepts this array as input, and computes an exclusive prefix sum over the group sizes (using the thrust library from NVIDIA [62]), yielding the start offsets in the grouping's final output buffer for each group. The `shuffle` primitive accepts this buffer as well as the original input and the key map produced by the `keymap` primitive, which enables it to process each input in parallel, writing each element of the input to its shuffled location in the output. (Atomic counters are used per group to manage the current output offset of each group). Conseqeuntly, the output of the `shuffle` is an array of group sizes and an array key extractor return values arranged such that all elements of each group are contiguous.

When the groupby primitive is not followed by an aggregation, the output of the `shuffle` can be deserialized directly into the IGrouping data structure required by the LINQ GroupBy API. When, as is the

thread.

case with k-means, the grouper is followed by an aggregation operation, the graph is extended with an aggregation subgraph consisting of the `aggregate`, `accumulate`, and `reduce` primitives shown in Figure 5. The `aggregate` primitive works by computing a segmented scan over the shuffled output produced by the upstream `shuffle` primitive, which produces an incremental aggregated value per group (for k-means, the sum of all the vectors in each group). In the general case, the aggregator produces an array of pairs where the key is the per-group incremental aggregation value, and the value is the count of each group–in many cases like k-means where the aggregation function explicitly uses the group count (see Figure 1), it is unnecessary to recompute the group sizes as part of the aggregation; it is already known due to upstream computations in the graph. The aggregator primitive uses the segmented scan APIs in the CUDA thrust library [62] to perform this step. The `accumulate` primitive takes care of accumulating each input chunk's share of the aggregation over multiple input chunks, and the `reduce` primitive produces a final value over all input chunks once all the input has been seen.

Note that the local dataflow graphs used by Dandelion are not acyclic. Significant extensions to the baseline GPU dataflow engine (PTask) are required to support cyclic dataflow and streaming in general; we describe these extensions in Section 4.3. Space constraints and readability concerns prevent us from describing the implementation of each relational operator at this level of detail. In general, the primitive library satisfies the need for parallel scan and sort operations by generating code that invokes thrust [62] APIs from nodes in the graph that execute on the host. Other primitives work by generating type- and lambda-specific instantiations of methods hand-coded to support each primitive such as hash table APIs and so on. Including additional code for managing mappings between CPU and GPU types, and instantiating dataflow graphs from descriptor files, the primitive library comprises 18,000 lines of C# code, and 3,500 lines of CUDA code.

4.3 GPU Dataflow Engine

The primitives that compose the computation are linked together in a dataflow driven by Dandelion's GPU dataflow engine, called EDGE. EDGE extends a basic DAG-based dataflow execution engine with constructs that can be composed to express iterative structures and data-dependent control flow; these constructs are required to handle cyclic dataflow

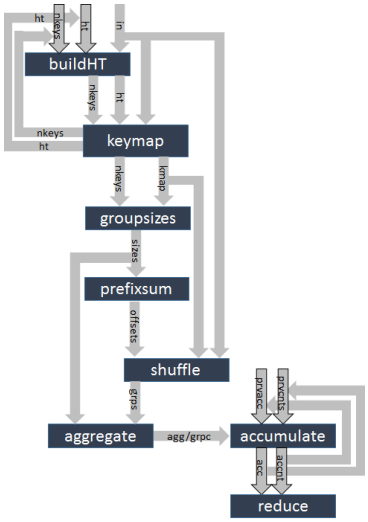

Figure 5: A local dataflow graph for the k-means workload.

and streaming for Dandelion. EDGE is similar to PTask [70], with the caveat that EDGE is entirely in user-mode. We adopt their nomenclature and *token model* [29] dataflow system: computations, or nodes in the graph are *tasks*, whose inputs and outputs manifest as *ports*. Ports are connected by *channels*, and data moves through channels discretized into chunks called *datablocks*. The programmer codes to an API to construct graphs from these objects, and drives the computation by pushing and pulling datablocks to and from channels. A tasks executes when a datablock is available at all of its input ports. Following the example of PTask, a unique thread manages each task in the graph, allowing EDGE to overlap data movement and computation for different tasks. EDGE's control flow constructs extend the functionality of ports and channels. EDGE uses the following abstractions:

ControlSignals. EDGE graphs carry control signals by annotating datablocks with a control code. The programmer defines arbitrary flow paths for these signals using an API to define a *c*ontrol propagation pair, which connects port pairs. Any control code received on a datablock received at the first port, will be propagated to the datablock on the second port. Examples of control signals include `BEGIN/END-STREAM` and `BEGIN/END-ITERATION`.

MultiPort. A MultiPort is a specialized input port that can be connected to multiple (prioritized) input channels. If a datablock is available on any of the input channels, the MultiPort will dequeue it, preferring the highest priority channel if many are ready.

PredicatedChannel. A PredicatedChannel allows a datablock to pass through it if the predicate

holds for the datablock. The API for configuring predicates allows the programmer to define whether the datablock is dropped from the channel or queued for later re-evaluation when the predicate fails. In general, the predicate function is a programmer-supplied callback, but we provide special support common predicates such as open/close on control signals such as `BEGIN/END-ITERATION`.

InitializerChannel. An InitializerChannel provides an pre-defined initial value datablock. InitializerChannels can be predicated similarly to PredicatedChannels: they are always ready, except when the predicate fails. InitializerChannels simplify construction of sub-graphs where the initial iteration of a loop requires an initial value that is difficult to supply through an externally exposed channel.

IteratorPort. An IteratorPort is a port responsible for maintaining iteration state and propagating control signals when iterations begin an end. An IteratorPort maintains a list of ports within its *scope*, which are signaled when iteration state changes. An IteratorPort also propagates `BEGIN/END-ITERATION` control signals along programmer-defined control propagation paths, which in combination with backward/forward PredicatedChannels can conditionally route data either to the top of another iteration, or forward in the graph when a loop completes. IteratorPorts can use callbacks to implement arbitrary iterators, or select from a handful of pre-defined functions, such as integer-valued loop induction variables.

Collectively, these constructs allow us to implement rich control flow constructs and iteration without additional specialized task nodes. Scheduling and resource-management for tasks maintains conceptual simplicity, and routing decisions are always computed locally by construction.

4.4 Cluster Dataflow Engine

Dandelion can run in two modes on a compute cluster. When it runs on a large cluster where fault tolerance is important, it uses a dataflow engine similar to Dryad and MapReduce. However, our main target platform is relatively small clusters of powerful machines with GPUs. In this case, fault tolerance is arguably not important. So we choose to maximize performance while sacrifcing fault tolerance. We build a new dataflow engine called Moxie that allows the entire computation to stay in memory when the aggregate cluster memory is sufficient. Disk I/O is only needed at the very beginning (to load the input) and at the very end (to write the result). The following are the important features of Moxie.

First, Moxie uses TCP to stream in-memory data between machines. It establishes a TCP channel between every pair of machines that are linked in the dataflow graph, and uses those links for data transfer. Using direct links avoids disk IO and increases the overall system performance, but loses the fault tolerance supported by systems like MapReduce and Dryad.

Second, Moxie tries to cache datasets in memory. This is especially important for iterative workloads like k-means or PageRank that would otherwise reload the same large input multiple times at each iteration. Caching at the application level allows Dandelion to store deserialized objects, saving further time. To maximize the benefits of caching, Moxie tries to assign a vertex to the same machine where its input is generated and possibly cached, and run the vertex in the same process with the cache. This allows us to reuse the cached objects by pointer passing. Here we again trade fault tolerance for performance. When there is memory pressure, Moxie automatically detects it and writes some of the cached datasets to disk.

4.5 Machine Dataflow Engine

The machine dataflow engine manages the computations on a machine. Each vertex of its dataflow graph represents a unit of computation that can always be executed on CPU and possibly on GPU. For vertices running on CPU, the dataflow engine schedules and parallelizes them to execute on the multiple CPU cores. It contains a new multi-core implementation of LINQ operators that has substantially better performance than PLINQ [] for the kind of workloads we are interested in. To run a vertex on GPU, we dispatch the computation to the GPU dataflow engine described in Section 4.3.

Asynchronous channels are created to transfer data between the CPU and GPU memory spaces. In general Dandelion tries to discover a well-balanced chunk size for input and output streams, and will dynamically adjust the chunk size to attempt to overlap stream I/O with compute. The presence of a GPU substrate can complicate this effort for a number of reasons. First since PCI express transfer latency is non-linear in the tranfer size, latencies for larger transfers are more easily amortized. Second, some primitives may be unable to handle discretized views of input streams for some inputs. For example the hash join implementation from our primtive library assumes that the outer relation is not-streamed while the inner relation may be. This requires that primtives be able to interact dynami-

	Configuration
Processor	Intel Xeon E5505 2.00GHz: 4 cores
L1	32 KB i + 32 KB d per core
L2	unified 4 × 256 KB per core
L3	4MB
Memory	12 GB
GPU	1 × NVIDIA Tesla M2075
GPU Memory	6 GB GDDR4
GPU Framework	PTask, CUDA 5.0
OS	Windows Server 2008 R2 64-bit
network	Intel 82575 Gigabit Dual Port

Table 3: Machine and platform parameters for all experiments.

cally with readers and writer on channels connecting local dataflow graphs to the global graph, and in some cases all the input for a particular channel must be accumulated before a transfer can be initiated between GPU and CPU memory spaces and GPU side computation can begin.

5 Evaluation

In this section, we evaluate Dandelion. Table 2 shows benchmarks used in the evaluation along with input details, while machine and platform parameters are detailed in Table 3. We evaluate Dandelion in both single-machine and distributed cluster environments, the two primary use scenarios of Dandelion. We compare the performance of a sequential LINQ implementation against Dandelion using only multiple CPU cores and Dandelion leveraging GPUs to offload parallel work.

5.1 Single Machine Performance

In this section we consider the performance of Dandelion running on a single host with a multi-core CPU and one GPU. Figure 6 shows the speedup over sequential performance for Dandelion using CPU parallelism and GPU parallelism. Table 4 provides details about dataflow graph structure and per-task latencies. Speedups range from 0.8x to 6.9x and 1.2x to 21.9x for the multi-core and GPU configurations respectively. The geometric mean speedup across all benchmarks is 2.2x for the multi-core configuration and 5.6x for the GPU-based configuration.

The data show that while Dandelion is able to improve performance over sequential in almost all cases with both multi-core parallelism and GPU parallelism, the degree to which the GPU-based implementation is profitable depends heavily on the ratio of arithmetic computation to memory access.

bnc	**tsk**	**dc**	**dt**	**ac**	**H-D**	**D-H**
k-means	11	777	2.5	0.5	0.6	0.1
page rank	21	34	1.6	0.5	18.0	0.1
skyserver	34	56	0.3	1.5	3.3	0.1
blk-scholes	1	16	0.1	0.4	4.2	0.1
sort	1	1	10.2s	0.1	386	0.1
dec. tree	31	1250	0.48	0.47	0.34	0.5

Table 4: Benchmark statistics for single-machine executions of Dandelion benchmarks, collected for the large variants only, using the 1 GPU configuration. The **tsk** column is the number of vertices in the dataflow graph, while **dc** is the dispatch count, or the total number of vertex executions required to complete the benchmark. The **ac**, **H-D**, and **D-H** columns represent the average latency in milliseconds required per vertex execution for device-side memory allocation, host-to-device memory transfer, and device-to-host memory transfer respectively.

Sky Server Q18. For the Sky Server Q18 running in the single-machine configuration, data are synthesized according to parameters that determine the number of unique objects in the outer relation, the number of unique objects in the inner relation, and the percentage of objects that will actually match on the join attribute. Relation sizes run from 10000 records to 50,000,000 records. We omit data that correspond to a range sweep on the percentage of matches on the join attribute because we found that the performance was comparatively insensitive to it.

Because Dandelion's join implementation is a hash join, the performance of the GPU version is almost entirely determined by the size of the outer relation: in general, when Dandelion can build a reasonably small hash table on the GPU, performance is very good, especially as the size of the outer relation increases. For example, when with just 10,000 or 100,000 unique keys in the outer relation, we get 20X speedup over sequential once we hit 50,000,000 records in the inner relation. At 1,000,000 records outer relation, Dandelion hits a performance cliff; and is always slower than sequential and parallel CPU in these cases. Moreover, the end-to-end runtime seems to be largely insensitive to the actual size of the inner relation in these cases. We experimented with expanding the number of hash-table buckets and the GPU device heap size to ensure the hash table is balanced for these large outer relations, but performance remained unaffected.

Page Rank. Page rank, expressed in LINQ, is a join followed by a group-by and select: very simple computations are used for the key extractor functions and testing of the join predicate, and so on.

benchmark	small	medium	large	description
k-means	k:10, N:20, M:10^6	k:10, N:20, M:10^6	k:80, N:40, M:10^6	k-means: M N-di pts $\rightarrow$ k clusters
page rank	P:50k L:20	P:50k L:20	P:50k L:20	page rank: P pages, L links per page
skyserver	O:10^5 N:10^6	O:10^5 N:10^7	O:10^5 N:5x10^7	SkyServer Q18: O objects, N neighbors
black-scholes	P:10^5	P:10^6	P:10^7	option pricing: P prices
sort	R:10^6	R:10^7	R:10^8	sort R 32-byte records
decision tree	R:10^5, A:100	R:10^6, A:100	R:10^6, A:1000	ID3 decision trees R records, A attributes

Table 2: Benchmarks used to evaluate Dandelion. For skyserver, and page rank, inputs for the single-machine experiments are synthesized to match the profile of the cluster scale inputs.

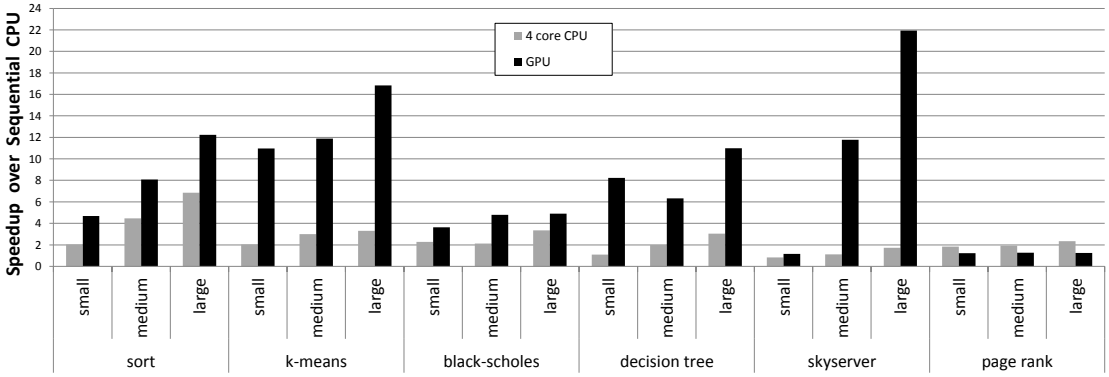

Figure 6: The speedup over Sequential CPU of Parallel CPU and GPU Enabled LINQ Provider, for different workloads and input data sizes

Consequently, on the GPU it is a mostly memory-bound workload: the ratio of machine cycles devoted to arithmetic operations to the cycles devoted to loads and stores is quite small, making this workload a near worst-case for Dandelion, and we expect the performance benefit from offloading to the GPU will be limited. Dandelion performance on the GPU improves over sequential performance only in cases where the input shard of the web graph is sized to yield a hash-table that does not push the device-side memory allocator into a non-performant range, and where that shard of the graph has high connectivity. Building the hash-table on the GPU is the most significant overhead for this workload, so when the graph is highly connected, Dandelion can amortize that cost more effectively. For small inputs, the cost of migrating data to and from the GPU compounds with the latency of building the hash table to yield unattractive performance. As the inputs sizes increase device-side memory allocation for hash-table nodes becomes a bottleneck. In the remaing envelope, the GPU is able to yield speedups over sequential in proportion to the connectivity of the graph. To examine this effect in more detail we ran the medium version of the workload with different levels of connectivity. With an average of only 3 outlinks per page, Dandelion on the GPU is 20% slower than a single core; in contrast, with averages of 20 and 40 outlinks per page, the GPU implementation provides 17% and 102% speedups respetively, over sequential.

5.2 Distributed Performance

In this section we consider the performance of Dandelion running on a small GPU cluster of 10 machines. Our evaluation is rather preliminary, and we only report detailed evaluations on k-means, our running benchmark.

Running on a cluster of computers allows Dandelion to scale to larger problems. In designing Dandelion, we believe we can obtain superior overall system performance by a) keeping the computation in memory and b) offloading computations to GPUs. We will focus on the performance impacts of these two aspects of Dandelion. We measure their contributions separately and their combined contribution against the base system.

We evaluate k-means using two datasets containing 100 millions points, with the dimensionality of 20 and 40 respectively. The datasets are divided into 10 partitions with one partition on each machine. We always use 80 centers for the experiments.

We evaluate the overall system performance of Dandelion, including Moxie and GPU acceleration. Figure 7 shows the performance of one iteration of k-means on various configurations. The GPU column adds only GPU acceleration to the Base sys-

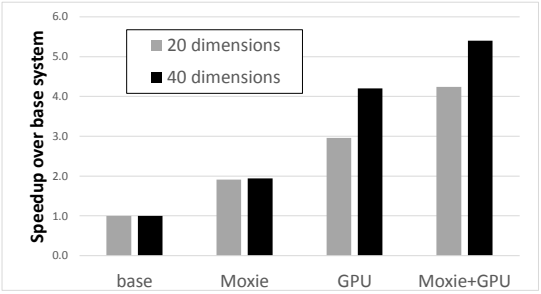

Figure 7: *k*means speedup for 100M points and 80 centers, running in various configurations on 10 machines

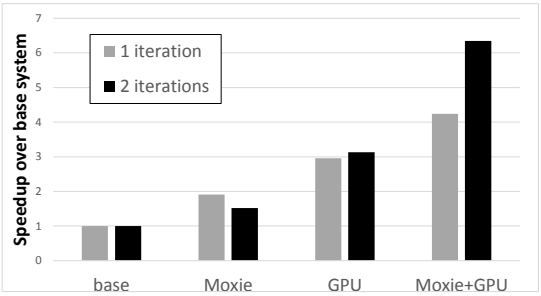

Figure 8: KMeans improvements

tem; Moxie+GPU is the complete system. As shown in the figure, GPU acceleration alone yields a 2.96x speedup. For the overall system, Dandelion achieved a performance improvement of 4.24x. Figure 8 shows the performance of *k*-means with two iterations. We observe a similar speedup to that observed with one iteration.

6 Related work

General-purpose GPU computing. The research community has focused considerable effort on the problem of providing a general-purpose programming interface to the specialized hardware supported by GPUs (GPGPU). GPGPU computing frameworks such as CUDA [64], OpenCL [51], and others [73, 16, 60, 19, 35, 77] provide increasingly expressive and full-featured front-end programming models that are specific to, and expose the underlying parallel GPU architecture. Dandelion, in contrast, provides a managed, sequential front-end programming model that exposes the programmer to comparatively little architectural detail, and which generalizes well across diverse target execution environments including not just GPUs but CPUs and FPGAs. A large body of research has been dedicated to offloading techniques such as channel processors [7], smart disks [50, 69, 26], and TCP offload [28].

Dataflow and streaming. Hardware systems such as Intel IXP [44] Imagine [8], and SCORE [22] rely on graph-based expression of parallel work where vertices are computations and edges represent dependences or data movement. Monsoon and Id [65] support dataflow execution directly in hardware. Click [54], CODE2 [61], and P-RIO [58] provides a graph-based programming models but do not address architectural diversity. Self/star [31] and Dryad [45] are graph-based programming models for distributed parallel execution in data center, the latter of which is extended by DryadLINQ [82] to support a sequential LINQ-based front-end programming model. Dandelion provides the same programming abstraction over a cluster of heterogeneous compute nodes, while Dryad and DryandLINQ can leverage CPUs at each compute node. Unlike Dryad and DryadLINQ, Dandelion does not yet enjoy support for fault-tolerance. Dandelion's support for caching of intermediate data in RAM between vertex executions is similar to the RDDs used in Spark [83].

GPUs and Dataflow. There is increasing recognition of dataflow as an effective programming model for managing and exploiting architectural diversity. StreamIt [76] and DirectShow [57] support graph-based parallelism; in the former, computations are connected by communicating channels, and in the latter, filter graphs are connected via "pins". OmpSs [20], Hydra [78], and PTask [70] all provide a graph-based dataflow programming models for offloading tasks across heterogeneous devices. EDGE, the GPU dataflow engine used by Dandelion adopts the PTask model directly, and extends it to support cyclic and iterative graph structures required to efficiently support relational operators on GPUs. Liquid Metal [42] and Lime [9] are programming platforms for heterogeneous targets such as systems comprising CPUs and FGPAs. Lime's filters, and I/O containers allow a computation to be expressed (by the compiler, in intermediate form) as a pipeline, and Lime's buffer objects encapsulate disjoint memory spaces in a way that is similar to the memory-space transparency provided by Dandelion's GPU dataflow engine. Flextream [40] is compilation framework for synchronous dataflow models that dynamically adapts applications to FPGA, GPU, or CPU target architectures, and Flextream

applications are represented as a graph. All of these works to are essentially complementary to Dandelion, which could use any of these systems as the local GPU dataflow engine; unlike these systems Dandelion does not directly expose the graph-based execution model.

Sponge [41] is an SDF language compiler framework that addresses portability challenges for GPU-side code across different generations of GPUs and CPUs, as well abstracting hardware details such as memory hierarchy and threading models. Like SDF languages in general [55, 34, 18], scheduling for Sponge is static, while Dandelion cross-compiles for a GPU target on demand and schedules work across available resources dynamically. The Dandelion compiler does not yet focus on optimization of GPU code; the runtime could benefit from implementation of the optimizations from Sponge.

Front-end programming models Many systems simplify GPU programming by providing GPU support in a high-level language: C++ [32], Java [81, 4, 67, 21], Matlab [6, 66], Python [23, 53]. Some of these systems go beyond simple GPU framework API bindings to provide support for compiling the high-level language to GPU code, none leverage multiple computers as Dandelion does, and unlike Dandelion all expose the underlying device abstraction to varying degrees. JaMP [52] provides Java bindings for OpenMP, allowing the use of multiple computers in a shared memory abstraction (without GPU acceleration). Unlike Dandelion, all of these systems leave the burden of scheduling squarely on the programmer's shoulders: the programmer must explicitly express which computations run on which devices and when. While Merge [56] provides a high-level language based on map-reduce that will transparently choose among multiple implementations given the input and machine configuration, it is targeted at a single machine with multiple CPUs and GPUs. Many of the workloads we evaluate for Dandelion have enjoyed research attention in the context of single-machine GPU implementations [30, 48] or cluster-scale GPU implementations [71]. These efforts have typically used GPU programming frameworks directly while Dandelion represents a new level of programmability for such platforms and workloads.

Scheduling and Execution engines for heterogeneous processors. Scheduling for heterogeneous systems is an active research area: PTask [70] and TimeGraph [49] focus on eliminating destructive performance interference in the presence of GPU sharing. Maestro [74] also shares GPUs but focuses on task decomposition, automatic data transfer, and autotuning of dynamic execution parameters in some cases. Sun et al. [75] share GPUs using task queuing. Others focus on making sure that both the CPU and GPU can be shared [47], on sharing CPUs from multiple (heterogeneous) computers [14, 15], or on scheduling on multiple (heterogeneous) CPU cores [17, 13]. Several systems [59, 33] automatically choose whether to send jobs to the CPU or GPU [11, 12, 10, 27, 72], others focus on support for scheduling in the presence of heterogeneity in a cluster [20]. Several systems consider providing a map-reduce primitive to GPUS, taking care of scheduling the various tasks and moving data in and out of memory [36, 24, 79]. The same abstraction can be extended to a cluster of machines with CPUs and GPUS [46]. This work was later improved with better scheduling [68].

Relational Algebra on GPUs The Thrust library, included with CUDA, offers some higher level operations like reduces, sorts, or prefix sums. He et al. also implemented a join primitive [38, 37]. Dandelion uses the sort and prefix sum primitives from Thrust library in several higher-level relational primitives.

7 Conclusion

Heterogeneous systems have entered the mainstream of computing. The adoption of these systems by the general developer community hinges on their programmability. In Dandelion, we takes on the ambitious goal to address this challenge for writing data-parallel applications on heterogeneous clusters.

Dandelion is still a research prototype under active development. The design goal of Dandelion is to build a complete, high performance system for a small sized cluster of powerful machines with GPUs. We believe that there are a lot of potential applications of such a system, in particular in the areas of machine learning and computational biology. We plan to continue to investigate the applicability of Dandelion across a broader range of workloads in the future.

References

[1] Anonymized for blind review.

[2] Anonymized for blind review.

[3] The CCI project. `http://cciast.codeplex.com/`.

[4] JCuda: Java bindings for CUDA.

[5] The LINQ project.

[6] Matlab plug-in for CUDA.

[7] *IBM 709 electronic data-processing system: advance description*. I.B.M., White Plains, NY, 1957.

[8] *The Imagine Stream Processor*, 2002.

[9] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*. ACM, 2010.

[10] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*, Shangai, Chine, Dec. 2010.

[11] C. Augonnet and R. Namyst. StarPU: A Unified Runtime System for Heterogeneous Multicore Architectures.

[12] C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE Architecture with the StarPU Unified Runtime System. In *SAMOS '09*, pages 329–339, 2009.

[13] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] R. M. Badia, J. Labarta, R. Sirvent, J. M. Prez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1:2003, 2003.

[15] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. 2004.

[16] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, and G. Refai-Ahmed. Scientific and Engineering Computing Using ATI Stream Technology. *Computing in Science and Engineering*, 11(6):92–97, 2009.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. In *SC 2006*.

[18] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, November 1992.

[19] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM TRANSACTIONS ON GRAPHICS*, 2004.

[20] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] P. Calvert. Part II dissertation, computer science tripos, university of cambridge, June 2010.

[22] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). FPL '00, 2000.

[23] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[24] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[25] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI'10*.

[26] S. C. Chiu, W.-k. Liao, A. N. Choudhary, and M. T. Kandemir. Processor-embedded distributed smart disks for I/O-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput.*, 65(4):532–551, 2005.

[27] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF 2008*, 2008.

[28] A. Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[29] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, 1982.

[30] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen. Parallel pagerank computation using gpus. In *Proceedings of the Third Symposium on Information and Communication Technology*, SoICT '12, pages 223–230, New York, NY, USA, 2012. ACM.

[31] C. Fetzer and K. Hgstedt. Self/star: A dataflow oriented component framework for pervasive dependability. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*, pages 66–73. IEEE Computer Society, 2003.

[32] K. Gregory and A. Miller. *C++ Amp: Accelerated Massive Parallelism With Microsoft Visual C++*. Microsoft Press Series. Microsoft GmbH, 2012.

[33] D. Grewe and M. OBoyle. A static task partitioning approach for heterogeneous systems using opencl. *Compiler Construction*, 6601:286–305, 2011.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[35] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU 2009*.

[36] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[37] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[38] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD '08, 2008.

[39] The HIVE project. `http://hadoop.apache.org/hive/`.

[40] A. Hormati, Y. Choi, M. Kudlur, R. M. Rabbah, T. Mudge, and S. A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, pages 214–223. IEEE Computer Society, 2009.

[41] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[42] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/-software boundary. In *ECOOP*, pages 76–103, 2008.

[43] Intel. Math kernel library. `http://developer.intel.com/software/products/mkl/`.

[44] Intel Corporation. *Intel IXP 2855 Network Processor*.

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

[46] W. Jiang and G. Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. *Parallel and Distributed Processing Symposium, International*, 0:644–655, 2012.

[47] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC 2009*.

[48] P. K., V. K. K., A. S. H. B., S. Balasubramanian, and P. Baruah. Cost efficient pagerank computation using gpu. 2011.

[49] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, Berkeley, CA, USA, 2011. USENIX Association.

[50] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, 1998.

[51] Khronos Group. *The OpenCL Specification, Version 1.2*, 2012.

[52] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. Jamp: an implementation of openmp for a java dsm. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007.

[53] A. Kloeckner. pycuda, 2012.

[54] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18, August 2000.

[55] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987.

[56] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, Mar. 2008.

[57] M. Linetsky. *Programming Microsoft Directshow*. Wordware Publishing Inc., Plano, TX, USA, 2001.

[58] O. Loques, J. Leite, and E. V. Carrera E. P-rio: A modular parallel-programming environment. *IEEE Concurrency*, 6:47–57, January 1998.

[59] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[60] M. D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM.

[61] P. Newton and J. C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 167–177, New York, NY, USA, 1992. ACM.

[62] NVIDIA. The thrust library. `https://developer.nvidia.com/thrust/`.

[63] NVIDIA. *CUDA Toolkit 4.0 CUBLAS Library*, 2011.

[64] NVIDIA. *NVIDIA CUDA 5.0 Programming Guide*, 2013.

[65] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th annual international symposium on Computer Architecture (ISCA)*, 1990.

[66] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[67] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using gpus from java. In *HPCC-ICESS*, pages 375–380. IEEE Computer Society, 2012.

[68] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 140–147, Washington, DC, USA, 2012. IEEE Computer Society.

[69] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[70] C. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *SOSP*, 2011.

[71] A. Rungsawang and B. Manaskasemsak. Fast pagerank computation on a gpu cluster. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '12, pages 450–456, Washington, DC, USA, 2012. IEEE Computer Society.

[72] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP 2008*.

[73] M. Segal and K. Akeley. The opengl graphics system: A specification version 4.3. Technical report, OpenGL.org, 2012.

[74] K. Spafford, J. S. Meredith, and J. S. Vetter. Maestro: Data orchestration and tuning for opencl devices. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2010.

[75] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, New York, NY, USA, 2012. ACM.

[76] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC 2002*.

[77] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC 2008*.

[78] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *ASPLOS 2008*.

[79] P. Wittek and S. DaráNyi. Accelerating text mining workloads in a mapreduce-based distributed gpu environment. *J. Parallel Distrib. Comput.*, 73(2):198–206, Feb. 2013.

[80] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 12, 2012.

[81] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In *Euro-Par*, pages 887–899, 2009.

[82] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2008.

[83] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.