Dandelion: a Compiler and Runtime for Heterogeneous Systems

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

Abstract

Computer systems increasingly rely on heterogeneity to achieve greater performance, scalability and energy efficiency. Because heterogeneous systems typically comprise multiple execution contexts with 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 diverse 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#. It therefore provides an expressive data model and native language integration for user-defined functions, enabling programmers to write applications using standard high-level languages and development tools.

Dandelion automatically and transparently distributes data-parallel portions of a program to available computing resources, including compute clusters for distributed execution and CPU and GPU cores of individual nodes for parallel execution. To enable automatic execution of .NET code on GPUs, Dandelion cross-compiles .NET code to CUDA kernels and uses the PTask runtime [85] to manage GPU execution. This paper discusses the design and implementation of Dandelion, focusing on the distributed CPU and GPU implementation. We evaluate the system using a diverse set of workloads.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522715

1 Introduction

The computing industry is experiencing a 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 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 runtimes 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 that a typical Dandelion cluster consists of a small number of moderately powerful computers. Such a cluster can easily have aggregated compute resources of more than 100,000 cores and 10TB of memory, and represents an attractive and affordable computing platform for many demanding applications such as large-scale machine learning and computational biology. Our goal 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 insulat-

ing 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 [101, 27, 103], and provides the most attractive high-level programming abstraction. We use LINQ [3], 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 [29] 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 execution model, where 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 the same goal as Dandelion: building scalable heterogeneous systems [58, 49, 98]. 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 backends including GPU and FPGA are supported. While this framework enables the programmer to write code in a mostly familiar way, target architectures do give rise to some limitations on the programming model.
- 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 PTask [85], a high performance dataflow engine for GPUs.

The remainder of this paper is organized as follows. Section 2 introduces the programming model for Dandelion. Sections 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

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 front-end of C# or F#. This section provides a high-level overview of this programming model.

2.1 LINQ

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

```
1  IQueryable<Vector>
2  OneStep(IQueryable<Vector> vectors,
3          IQueryable<Vector> centers) {
4    return vectors
5     .GroupBy(v => NearestCenter(v, centers))
6     .Select(g => g.Aggregate((x, y) => x+y)/g.Count());
7  }
8
9  int
10 NearestCenter(Vector vector,
11               IEnumerable<Vector> centers) {
12   int minIndex = 0;
13   double minValue = Double.MaxValue;
14   int curIndex = 0;
15   foreach (Vector center in centers) {
16     double curValue = (center - vector).Norm2();
17     if (minValue > curValue) {
18       minValue = curValue;
19       minIndex = curIndex;
20     }
21     curIndex++;
22   }
23   return minIndex;
24 }
```

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

`IQueryable<T>` (a subtype of `IEnumerable<T>`) to enable deferred execution of LINQ queries by a custom execution provider. Dandelion is implemented as a new execution provider that compiles LINQ queries to run on a distributed heterogeneous system.

As an example, Figure 1 shows 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 then computes the 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 from a library of primitives that are common to the underlying relational 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. Our main challenge is therefore to preserve the LINQ/.NET programming model, rather than designing new language features. This section describes a very small set of Dandelion specific extensions that we believe are essential. They are integrated in the LINQ programming model as user-defined operators and language attributes.

Dandelion extends LINQ with three 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 k-means program in Figure 1 using Dandelion, we add a call to `AsDandelion` to the two inputs as follows:

```
vectors = vectors.AsDandelion();
centers = centers.AsDandelion();
OneStep(vectors, centers);
```

The argument `gpuType` of `AsDandelion` is optional. It 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 informs the Dandelion runtime of the record size. In k-means, the vectors are all the same size; if we know the size is, e.g. 100, we 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.DoWhile(body, cond)`, a do while loop construct for iterative computations. The arguments `body` and `cond` are both Dandelion query functions, and `DoWhile` repeatedly executes `body` until `cond` is false. For example, our iterative k-means program is expressed as follows. `DoWhile` enables Dandelion to ship the execution of the entire conditional loop to the cluster or GPU, thus avoiding unnecessary context switching.

```
vectors = vectors.AsDandelion();
centers = centers.AsDandelion();
centers.DoWhile(
  centers => OneStep(vectors, centers),
  (centers, newCenters) =>
             NotConverged(centers, newCenters)
);
```

The third operator 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 [78] and MKL [56] into Dandelion, making the primitives defined in those libraries accessible at the programming API. Unlike all the other LINQ

operators, the correct use of `Apply` may require that the input data be partitioned in a certain way. The `Apply` operator has overloads that allows the programmer to specify the data partitioning of the input. Hash and range partitioning are supported. Dandelion introduces a data partitioning operation in the execution plan if it cannot infer that the input is not already partitioned correctly.

Dandelion also introduces an `Accelerated` annotation. By default, Dandelion automatically "kernelizes" the user-defined .NET functions invoked by the LINQ operators to run on the GPUs. However, 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(dev, dll, op)` attribute to a .NET function to override the cross-compilation by Dandelion. The annotation tells the system that the current .NET function can be replaced by the function `op` in the DLL `dll` on the computing device type `dev`. 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 [101, 27] 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 it is supported, the performance can be poor. Consequently, we choose not to kernelize any .NET function that contains dynamic memory allocation, and execute such functions on CPUs. When cross-compiling .NET code, Dandelion uses static analysis to infer the size of .NET memory allocations in the code and translates the fixed-size 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 an architectural overview of the Dandelion system. 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

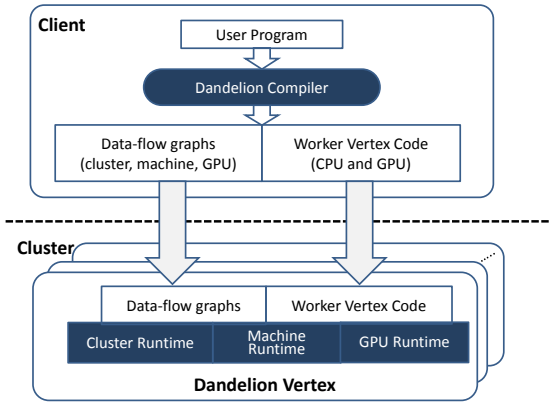

Figure 2: The Dandelion system architecture.

a computation as a dataflow graph. Each vertex in the dataflow graph represents a fragment of the computation and the edges represent communication channels. This section provides an 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 generates CUDA code, and three levels of dataflow graphs to orchestrate the execution. In this case, Dandelion generates CUDA implemenations for the `NearestCenter` function, along with the other supporting functions such as `Norm2` and other vector arithmetic operators.

The first dataflow graph is at the cluster level. The compiler applies query rewrite rules to the LINQ program to transform it into a dataflow graph that is optimized for parallel execution on the distributed compute substrate. At runtime, the graph is then annotated, deciding on which machine each vertex should execute, leveraging information gleaned at runtime. Because this aspect of the system is similar to existing systems [101, 103, 52] we do not elaborate on it further.

At the next level, as vertices from the cluster-level dataflow graph are assigned to machines, they are expanded into a machine-level dataflow graph describing the execution on that machine. The system decides which vertices are executed on GPUs. In our k-means example, the entire computation can be offloaded to GPU(s).

Finally, for vertices that are assigned to the GPU, a GPU-level dataflow graph is generated to describe the computation. It combines the CUDA-code generated earlier with GPU code for the LINQ operators that use

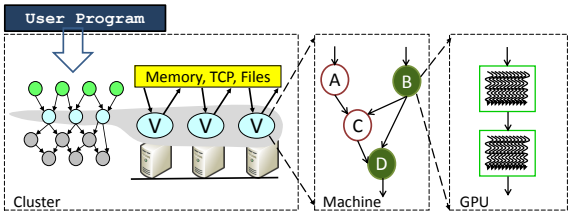

Figure 3: A global view of a Dandelion dataflow graph.

them. Dandelion implements LINQ operators via a GPU library consisting of a large collection of modular primitives. Most 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 for composability: new composite primitives can be formed from more basic ones. 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.

3.2 Dandelion Runtime

Figure 3 shows the three dataflow graphs, each corresponding to an execution layer in the system. These three layers together form the Dandelion runtime, and the composition of those graphs forms the global dataflow graph for the entire computation.

The cluster execution engine assigns vertices to available machines and distributes code and graphs, orchestrating the computation. Each machine executes its own dataflow graph, managing input/output and execution threads. Vertices in the machine dataflow graph run on either CPUs or GPUs. The execution of the GPU vertices is delegated to the GPU dataflow engine; we use PTask [85] as our GPU dataflow 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 Dandelion system, including various layers of the compiler and runtime. Dandelion is a 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 goal of Dandelion is to enable automatic execution on GPUs for programs written in C# or F#. 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 code. This compilation step takes .NET bytecode as input and produces CUDA source code as output. Working at the bytecode level allows us to handle binary-only application code.

We build our cross-compiler using the Common Compiler Infrastructure (CCI) [2]. CCI provides the basic mechanism of reading the bytecode from a .NET assembly into an abstract representation (AST) that is amenable to analysis. To perform the cross-compilation, Dandelion maps all referenced .NET object types to CUDA struct types, translating all the reachable .NET methods into GPU kernel functions. For generic methods, each reachable instance is translated into a separate CUDA function. 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 the 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. `KernelVector` is the translated CUDA type for the .NET type `Vector`.

While Dandelion's cross-compiler is quite general, there are limitations on its ability to find a usable mapping between .NET code and GPU code. The primary constraint, discussed in Section 2.3, derives from the presence of dynamic memory allocation. Dandelion 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 applies 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 propagation through the code using the summary. The final pass emits the CUDA code.

In languages such as C# and Java, array size is 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 major issue with 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 value. Such data are logically decoupled into separate channels: records in the data channel are laid out in a format optimal for GPU processing. To avoid destroying that data layout, metadata are kept in a separate channel that is associated with the data channel to transfer the metadata. Dandelion performs this transformation for all the inputs to GPU when it generates the serialization code. This enables us to automate the handling of variable-length input records. 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. Dandelion resorts to this general mapping only when it fails to statically infer the size of the record type.

Another important problem that Dandelion must address is the GPU execution context for a cross-compiled kernel function. This involves capturing all the global variables referenced by the function and transferring their values to GPU. When the compiler comes across a global variable, if its value is small, the compiler just inlines it in the generated code. If a value is large, the compiler adds a binding in the context and uses it to reference the value in the generated code. The context is treated as a GPU value and transferred to GPU using a dedicated vertex in the PTask computation graph.

4.2 GPU Primitive Library

The GPU primitive library exposes a set of primitives that are used by the Dandelion compiler to construct the GPU dataflow graph. It also provides an API that allows programmers to form new primitives by composing existing ones. The primitives exposed include low-level building blocks (e.g., parallel scan and hash tables) and high-level primitives for relational operators (e.g., GroupBy and Join).

Almost all the primitives are generic: the input/output datatypes and user-defined functions are template parameters of the primitives. When constructing GPU dataflow graphs, the Dandelion compiler instantiates the primitives using generated GPU datatypes and code. This design leads to a clean separation of graph construction and cross compilation. The parallelization strategy for each operator is primarily the concern of the primitive implementation rather than of compiler, yielding a general approach to parallelization.

```
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.

We again use the k-means example to illustrate the primitives and their compositions to form new primitives. Figure 5 shows the GPU dataflow graph for k-means. Recall that the k-means LINQ query is a `GroupBy` followed by an aggregation expressed as a `Select` operation. A naïve implementation of this query would connect a subgraph that implements the `GroupBy` with a subgraph that implements the `Select`. However, as described in [100], a better approach is to fuse the aggregation with the grouping on each input datablock. The k-means graph uses this optimization.

The graph starts with a stateless incremental grouping primitive (`grouper`). Mapping the grouping operation to the parallel architecture leverages its fundamental similarity to shuffling: shuffle a block of records into contiguous regions such that the records with the same keys end up in the same region. To perform this operation in parallel, each hardware thread reads a single input record and writes it to the output at some offset in

its key region. [1] Finding the output offset for each record requires us to know the start index of each group in the output, which in turn requires us to know the number of groups and the number of elements in each group. The `grouper` primitive computes these numbers in several stages.

In the first stage, the `buildHT` primitive uses a lock-free hash table to maintain the set of unique group keys: each hash entry is a record containing a key and an integer-valued group identifier, which is assigned upon successful insertion. Each thread computes the key for a record and attempts to insert it into the hash table. The hash-table uses a CAS operation on hash table record pointers to ensure that only the first attempting to insert a particular key value will succeed; upon successful insertion, we rely on atomic increment support on the GPU to maintain the number of unique groups. The result of this increment becomes the group identifier for a new group. The `buildHT` primitive produces two outputs: the hash table (`hashtbl`) and the unique key count (`unq_cnt`). The `keymap` primitive uses the hash table and the key counts to build a map from each record in the current input datablock to its group identifier. The hash table and key count are then returned on back edges of the dataflow graph to be used in subsequent invocations of `buildHT`.

The `groupsizes` primitive then uses the output of `keymap` to compute the number of elements present in each group. It allocates an array of counters for the keys, and each thread increments the count in the array for an input key using the atomic increment operation. The output of `groupsizes` is the array of counters, which is used by `prefixsum` to compute the start offsets for each group. The `shuffle` primitive then uses the offsets to shuffle the original input datablock. The output of the `shuffle` primitive is a triple of arrays: the group keys, the group sizes, and the records shuffled into contiguous groups. When the grouper is not followed by an aggregation, the three arrays are synthesizeed into an object of type `IEnumerable<IGrouping<K,T>>`, the return type of the LINQ `GroupBy` API.

When, as is the case with k-means, the grouper is followed by an aggregator, the graph is extended with a subgraph consisting of `aggregate`, `accumulate`, and `reduce` primitives. The `aggregate` primitive computes a segmented prefix scan (implemented using thrust [77]) over each shuffled datablock, producing a partial aggregated value for each group. The `accumulate` primitive then accumulates those partial aggregated values over all input datablocks. The `reduce` primitive produces a final aggregation value for each group, which in k-means gives us the set of new centers.

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

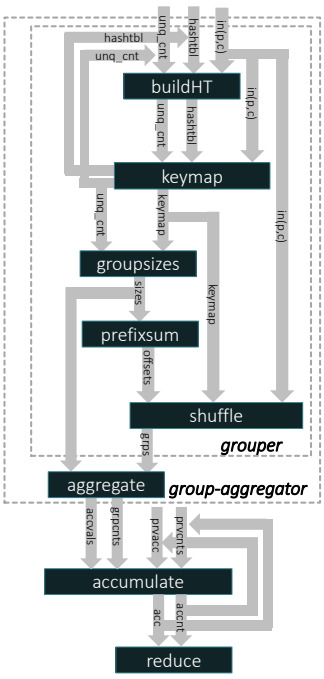

Figure 5: The GPU dataflow graph for the k-means workload.

4.3 GPU Dataflow Engine

The primitives that compose the computation are linked together in a dataflow graph driven by PTask, Dandelion's GPU dataflow engine. PTask extends the DAG-based dataflow execution engine described in [85] with constructs that can be composed to express iterative structures and data-dependent control flow; these constructs are required to handle cyclic dataflow and streaming for Dandelion. PTask is a *token model* [35] 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 task is ready for execution when all of its input ports have available datablocks and all of its output ports have capacity. PTask multiplexes a number of threads over the tasks in a graph, ideally mapping a unique thread to each task, and degrading to a pool-based assignment when the number of nodes in the graph is large. This approach allows PTask to overlap data movement and computation

for many concurrent tasks. We extended PTask with the following abstractions to expose control flow constructs needed by Dandelion:

ControlSignals. PTask graphs carry control signals by annotating each datablock with a stack of control codes. Operations that examine control codes always examine the top of the stack. The programmer defines arbitrary flow paths for these signals using an API to define *scope entries* and *scope exits*, and *control propagation pairs*. Scope entries and exits can be either ports or channels and represent locations in the graph where a subgraph that requires its own control signals can push and pop control signals as datablocks enter and leave the subgraph. PTask additionally supports an API for defining actions that occur in response to control signals observed at scope entries and exits. Control propagation pairs connect ports: the control code stack observed on a datablock received at the first port, will be propagated in its entirety to the datablock at the second port (or to the next one that is bound to the second port). Examples of control signals include `BEGIN/END-STREAM` for streaming and `BEGIN/END-ITERATION` for iteration. In the k-means example described above, `BEGIN/END-STREAM` control signals are used extensively to enable stateful primitives to emit output only when all the input datablocks have been processed, and scope entries and exits are used to enable composition of stateful primitives.

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. Dandelion relies heavily on MultiPorts to support cyclic constructs and to allow primitives to unambiguously choose amongst input datablocks when they may be available on both forward and back channels bound to a port.

PredicatedChannel. PredicatedChannelsare used to define conditional routing for datablocks. A PredicatedChannel allows a datablock to pass through it only if the predicate holds for the datablock. In general, the predicate function is a programmer-supplied callback, but we provide common predicates for predefined control signals such as `BEGIN/END-STREAM` and `BEGIN/END-ITERATION`. For example, in the k-means graph from Figure 5, the forward and back channels labelled `acc` and `accnt` are predicated such that the forward channels block any datablocks that do not have an `END-STREAM` control signal, while the back channels do the converse: as a result, the `accumulate` primitive produces an output datablock only when all input datablocks have been processed upstream.

InitializerChannel. An InitializerChannel provides a 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. InitializerChannels are convenient for providing seed values for stateful primitives and for avoiding the proliferation of channels exposed to the runtime: their primary use is to provide initial values in response to control signals. For example, in the k-means graph, the `unq_cnt` and `hashtbl` inputs are InitializerChannelswith predicates that allow them to produce a new initial value only when a `BEGIN-STREAM` control signal is observed on a datablock received at the `in(p,c)` input to `buildHT`. This enables the `grouper` primitive to create new hash table and unique key count blocks for each distinct stream of datablocks without requiring synchronous intervention from the runtime to reset the primitives to an initial state.

IteratorPort. An IteratorPort is a port responsible for maintaining iteration state and propagating control signals when iterations begin and 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 back to the top for 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 in PTask. The routing decisions for datablocks are by construction always made locally, so scheduling and resource-management for tasks remain conceptually simple.

Our goal with Dandelion is to bring GPUs to general-purpose programs, which in turn implies some kind of OS multiplexing of the machine between multiple applications. PTask, as originally envisioned, proposes OS-level abstractions for building and managing dataflow graphs that perform heterogeneous computations: in return for expressing computations as a graph, the user gets an environment where the OS can make (potentially best-effort) guarantees about fairness and isolation in the presence of multiple computations, as well as minimal (or zero-) copy data movement as data are shared dynamically across devices that may have disjoint or incoherent memory spaces. Due to lack of direct PTask support in Windows, we rely on a user-mode implementation of PTask. However, we hold that a production implementation of Dandelion would rely on OS-level sup-

port from a kernel-mode PTask implementation to ensure that local machine resources are well-multiplexed across concurrent computations that share the cluster.

4.4 Cluster Dataflow Engine

Dandelion can run in two modes on a compute cluster. When it runs on a large cluster where fine grained fault tolerance matters, it uses the Dryad dataflow execution engine. In this design, the output of a vertex is written to disk files so transient failures can be recovered by vertex re-execution using its disk inputs. However, for our main target platform of relatively small clusters of powerful machines with GPUs, we favor a different design that attempts to maximize performance. We therefore built a new distributed 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). Similarly to Spark's RDD [103], Moxie holds intermediate data in memory and can checkpoint them to disk. The high-level architecture of Moxie is similar to Dryad and YARN [1]. A Moxie job consists of an application master and a collection of containers, all running on the compute nodes of a cluster. The application master is responsible for assigning tasks to the containers which perform the actual computations. Below we highlight some of the important features of Moxie.

First, Moxie aggressively caches in memory datasets (including intermediate data) that will be used multiple times in a single job. This is especially important for applications involving iterative computations such as k-means or PageRank that would otherwise reload the same large input for each iteration. Each container maintains a cache. A cached value is exposed as a .NET collection, but it could be stored either in the CPU memory or the GPU memory. A cache entry is reference-counted and is removed when its reference count reaches 0. When there is memory pressure, Moxie automatically detects it and spills some of the cached datasets to disk. To maximize the benefits of caching, the Moxie application master tries to assign a vertex to the container where its input was generated and possibly cached in memory. This allows us to reuse the cached objects by pointer passing. In the event that the input of a vertex is in a remote cache, Moxie uses TCP to transfer the data from the remote memory to the current container.

Second, Moxie uses asynchronous checkpoints to support coarse-grained fault tolerance. Moxie selectively checkpoints the intermediate results that it considers to be important to protect. For example, Moxie may choose to protect the output of an expensive task. In the current implementation, Moxie creates a checkpoint for the outputs of every iteration of the `DoWhile` operator. Any failure would trigger the re-execution of all the unprotected upstream tasks. The checkpointing is asynchronous so it does not introduce unnecessary synchronization barriers in the execution. Checkpoints are stored in a distributed fault tolerant file system.

4.5 Machine Dataflow Engine

The machine dataflow engine manages the computations on a compute node. 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. Dandelion contains a new multi-core implementation of LINQ operators that substantially outperforms PLINQ [4] for the kind of data intensive workloads we are interested in. To run a vertex on GPU, it dispatches 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 transfer 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 primitive library assumes that the outer relation is not streamed while the inner relation may be. This requires that primitives be able to interact dynamically with readers and writers 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 in both single-machine and distributed cluster environments. We compare the performance of the sequential LINQ implementation shipped in .NET against Dandelion using only multiple CPU cores and Dandelion using GPUs to offload parallel work. Additionally, to provide perspective on the quality of Dandelion's generated code and query plans, we compare the performance of Dandelion's k-means on a single machine against a number of hand

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-dim points $\rightarrow$ k clusters
pagerank	P:100k L:20	P:500k L:20	P:1m 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 [102]: O objects, N neighbors
black-scholes	P:10^5	P:10^6	P:10^7	option pricing: P prices
terasort	R:10^6	R:10^7	R:10^8	sort R 100-byte records [5]
decision tree	R:10^5, A:100	R:10^6, A:100	R:10^5, A:1000	ID3 decision trees R records, A attributes
bm25f	D:2^{18}	D:2^{19}	D:2^{20}	search engine ranking [104], D documents

Table 1: Benchmarks used to evaluate Dandelion. For skyserver, and pagerank, inputs for the single-machine experiments are synthesized to match the profile of the cluster scale inputs. For bm25f, inputs are drawn from [19].

	Configuration
Processor	2×Intel Xeon E5-2630 2.30GHz
CPU Cores	12 (2 threads per core)
L1	32 KB i + 32 KB d per core
L2	unified 256 KB per core
L3	15MB
Memory	256 GB
GPU	1 × NVIDIA Tesla K20M
GPU cores	13 SMs → 2496 cores per GPU
GPU Memory	5 GB GDDR5
GPU Engine	user-mode PTask, CUDA 5.0
OS	Windows Server 2008 R2 64-bit
Network	Mellanox ConnectX-3 10 Gigabit Ethernet

Table 2: Machine and platform parameters for all experiments.

tuned C#, C++, and CUDA implementations. Table 1 provides a summary of benchmarks and inputs used in the evaluation, while machine and platform parameters are detailed in Table 2.

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 speedups over sequential LINQ (labelled LINQ), Dandelion using only CPU parallelism (labelled Dandelion-M), and two versions of Dandelion using only GPU parallelism. The first (labelled Dandelion) represents the Dandelion default configuration. The second (labelled Dandelion-H) reflects the performance of Dandelion with additional hints, provided with annotations, about expected dataset sizes. In all cases, we measure wall-clock time, which for the GPU case, includes serialization and de-serialization of C# objects as well as PCI transfer latencies incurred moving data to and from the GPU memory.

In its default configuration, Dandelion is generally able to outperform parallel CPU dandelion: the geometric mean across all benchmarks for Dandelion-M is $3.1\times$, while the GPU dandelion sees $6.1\times$ and $6.4\times$ for the default and hint-based variants respectively. Because serialization and PCI transfer costs are fundamental in any GPU offload system we do not report device-only execution times; however, it is worth noting that if we neglect such costs, the geometric mean speedup of Dandelion over all benchmarks is over $30\times$.

The Dandelion-H data demonstrate that memory management and transfer overheads are a first order concern: additional speedup is available if the runtime has some hints to help it avoid memory allocation and PCI transfers. Device-side memory allocations force the GPU driver to synchronize with the host, eliminating the runtime's ability to hide data transfer latencies with GPU execution. However dynamic memory allocation is a requirement in many of our benchmarks. For example, a group-by operation must compute the number of groups before allocating downstream buffers for those groups. Pooling of datablocks, the PTask encapsulation of GPU buffers, can mostly eliminate device memory allocation on the critical path: PTask adopts a strategy similar to the Linux slab allocator to provision in advance for such dynamic allocations. However, because PTask's pools are sized heuristically, the runtime may still need to perform memory allocation at GPU dispatch time if pools are poorly sized for the workload. The Dandelion-H variant shown in Figure 6 shows the performance achievable when block pools are sized to avoid critical-path allocations. The improvements enabled by this optimization range from under 1% to 20% over Dandelion in its default configuration, with a geometric mean across all benchmarks of 8.9%. We do not consider the need for such hints fundamental to Dandelion: this need could be eliminated by fully virtualizing all of GPU memory within PTask. We leave this virtualization effort as future work; in the current prototype, we use annotation to communicate such hints to the runtime.

While Dandelion is performance profitable over sequential and multi-core CPUs, the extent of that profitability depends heavily on the ratio of arithmetic computation to memory access, as illustrated by the performance of PageRank. In LINQ, PageRank is expressed

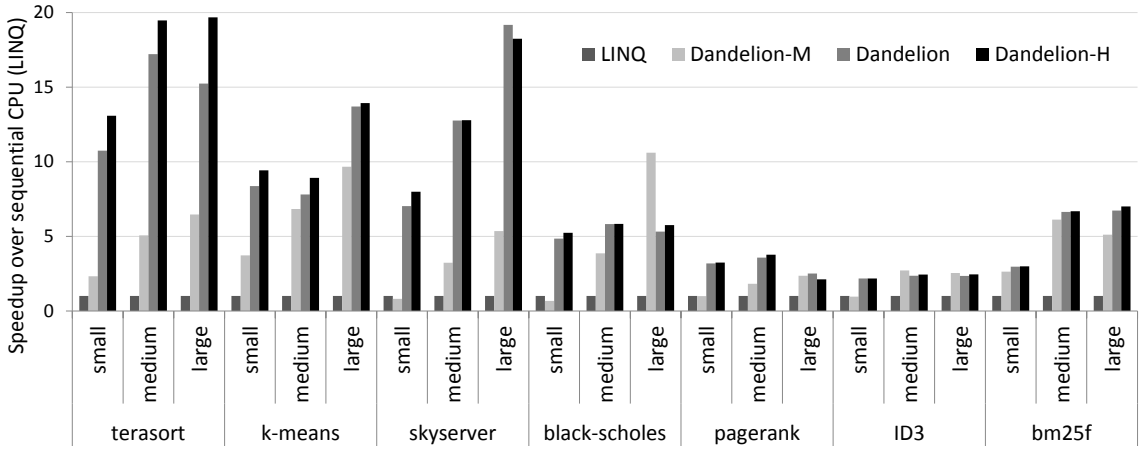

Figure 6: The speedup over sequential CPU (LINQ) of parallel CPU (Dandelion-M), GPU enabled (Dandelion) and GPU enabled with memory allocation hints (Dandelion-H) versions of Dandelion, for different workloads and input data sizes.

using join, groupby and select, where the key extractor and join predicates are very simple computations, making it a mostly memory-bound workload on the GPU. Nonetheless, while Dandelion PageRank performance on the GPU is modest compared to the other benchmarks, it still improves over sequential implementations by a geometric mean of $2.9\times$ over all the input sizes. Because Dandelion's join implementation is a hash join, the performance of the GPU version has some sensitivity to the size of the outer relation. Our lock-free GPU hashtable manages its own memory allocation for hash table entries to avoid relying on the performance-sapping CUDA `malloc` implementation; however, we do fall back to CUDA `malloc` under heavy memory pressure, which occurs with the large input, explaining a drop off from $3.2\times$ to $2.44\times$ moving from the medium to large input. We believe this performance loss can be ameliorated with better heuristics for provisioning our hashtable sub-allocator.

5.2 K-means in depth

In this section we compare the performance of 15 implementations of k-means across 3 different input sizes to provide a detailed characterization the trade-offs in performance and developer effort for Dandelion relative to more familiar tool-chains such as C++, C#, as well as a GPU-specific tool-chain: CUDA. Table 3 describes the implementations, characterizes them in terms of implementation difficulty (a subjective estimate based on the level of architectural expertise required of the programmer for each implementation), the number of lines of code, as well as the speedup of each over the sequential C++ implementation. Figure 7 plots the speedups over C++ for the medium size input only; other input sizes show the same trend.

The number of CUDA implementations illustrates the range of optimization strategies that any performance-hungry CUDA programmer must explore, and provides a backdrop against which to consider the performance of the GPU code generated by Dandelion. A typical CUDA optimization effort must consider multiple specialized low-latency memories, such as constant and shared, and hard-to-manage optimizations like memory coalescing [79]. To ensure we characterize this space with reasonable fidelity, we implement CUDA versions of k-means that represent a cross-product of these strategies: in Table 3, only those that illustrate an important performance delta are represented. Similar to Section 5.1, we evaluate Dandelion in both default and hint-driven configurations.

The data illustrate that Dandelion provides a compelling fraction of the available performance for minimal programmer effort, out-performing sequential native managed code by $45\times$, $56\times$, and beating multi-threaded C# (using 24 cores) by $4\times$, simultaneously reducing the number of lines of code required to express the workload by $10\times$, $4.9\times$, and $12.4\times$ respectively. Performance of the CUDA implementations varies dramatically across input sizes: an optimization strategy effective for one input may be ineffective for another (e.g. the constant memory optimization does little to improve the **cuda-ro** variant for the small input but provides a dramatic benefit for the medium input because the L1 GPU cache is able to provide the same reduction in effective memory latency). In general, the most effective optimization is the arrangement of input data to promote memory coalescing (which for k-means can be rea-

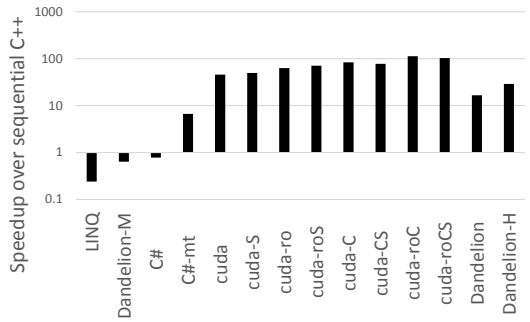

Figure 7: The speedup over sequential C++ for various hand-optimized implementations and the Dandelion implementation of k-means, shown only for the "medium" input, as the trend across other input sizes is similar. Speedups are shown in logarithmic scale - higher is better.

sonably understood as arranging input vectors column-major instead row-major), followed closely by the ability to leverage the GPU shared memory.

All these optimizations require programmer expertise, and all of the hand-optimized CUDA versions require $20\times$ more lines of code to express than Dandelion. However, Dandelion's generated implementation provides performance that we consider to be a reasonable fraction of that achieved by the hand-optimized implementations, ranging from $2\times$ to $6.7\times$, $1.6\times$ to $2.8\times$, and $2.8\times$ to $7\times$ slower for the small, medium, and large inputs respectively. With the benefit of memory allocation hints, Dandelion is able to make up much of the remaining performance gaps, coming within a factor of 1.9 of the best-case hand-optimized CUDA for the medium input. Finally, we believe that many of these optimizations can be automated in the future in Dandelion (see the **Automatable** column in Table 3). For example, LINQ collections are immutable, giving the Dandelion compiler the flexibility to place them in GPU constant memory when a collection is small enough.

5.3 Distributed Performance

In this section we consider the performance of Dandelion running on a small GPU cluster of 10 machines. The primary objective is to evaluate the effectiveness of GPU offloading in a distributed setting. Similar to our single machine evaluation, we first take an in-depth look at the performance of a single benchmark, k-means, and then report on the overall system performance of Dandelion by comparing Dandelion running with the GPU against Dandelion running single-threaded and multi-threaded without the GPU.

Table 4 details the benchmarks. There are four

experiment	data
k-means	1B 40-dimensional points, 120 centers. 152.7 GB.
PageRank	CatB ClueWeb dataset, 14.5 GB.
SkyServer	SkyServer datasets. 53.6 GB .
Terasort	500M 10-byte keys, 100-byte records. 50 GB.

Table 4: Input sizes for distributed experiments.

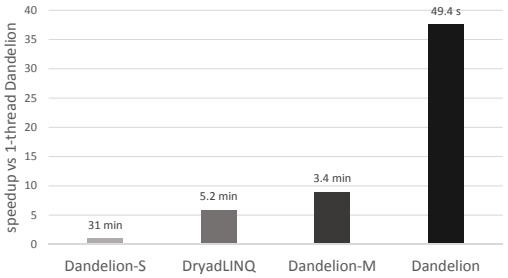

Figure 8: k-means performance comparison against DryadLINQ and Dandelion-M.

benchmarks: k-means, PageRank, SkyServer and Sort. For PageRank, the input dataset is the "Category B" dataset of the widely used ClueWeb09 datasets. SkyServer is the most time consuming query (Q18) from the Sloan Digital Sky Survey database [44]. Terasort is a general-purpose sort running on 50GB of Terasort records, evenly partitioned over the 10 machines. The benchmarks were originally written for CPU; running them on Dandelion required no modification beyond the `AsDandelion()` extension discussed in Section 2.2.

K-means. Figure 8 compares the k-means performance of Dandelion against DryadLINQ and two variants of Dandelion. Dandelion-S and Dandelion-M both use the Dandelion distributed execution engine but run the vertex code using single threaded and multi-threaded CPU LINQ implementations respectively. In this experiment, we run one iteration of k-means. The evaluation shows that Dandelion with GPU has the best performance, approximately $5\times$ and $3.9\times$ faster than the base system DryadLINQ and our CPU variant Dandelion-M respectively. Since DryadLINQ and Dandelion-M use a very similar CPU execution engines, we attribute the speedup gained by Dandelion-M over DryadLINQ to two factors: a) keeping the data in memory and using TCP for data transfer as opposed to disk file based communication and b) a bigger constant job startup/exit costs of DryadLINQ. While the performance gain from using GPU is quite good, it is significantly less than the single machine case. We attribute it to the overhead associated to distributed execution. For example, in this experiment, the input data need to be read from disk files, deserialized to CPU memory, and then transferred to GPU memory.

Name	Description	Difficulty	Automatable	LOC	Speedup over sequential C++		
					Small	Medium	Large
C++	sequential C++, CPU	easy		491	1.0	1.0	1.0
LINQ	sequential LINQ, CPU	easy		38	0.2	0.26	0.24
Dandelion-M	parallel LINQ, CPU	easy		38	0.5	0.56	0.64
C#	sequential C#, CPU	easy		186	0.7	0.8	0.78
C#-mt	parallel C#, 24 cores	moderate		473	3.64	9.88	6.66
cuda	basic CUDA	moderate		651	15.8	71.4	45.6
cuda-S	CUDA + shared memory	hard	yes	781	25.8	81.4	49.6
cuda-ro	CUDA + constant memory	moderate	yes	668	17.1	114.6	62.7
cuda-roS	CUDA + const + shared mem	hard	yes	815	30.7	125.3	70.8
cuda-C	CUDA coalesced mem access	moderate	maybe	731	38.9	104.1	83.6
cuda-CS	CUDA + coalesced + shared	hard	no	874	41.6	81.22	77.5
cuda-roC	CUDA + coalesced + const	moderate	no	757	46.0	70.6	112.7
cuda-roCS	CUDA + coalesced + const + shared	hard	no	909	50.8	60.2	102.5
Dandelion	dandelion default configuration	easy		38	7.7	45.2	16.6
Dandelion-H	dandelion, memalloc hints	moderate	no	42	14.4	66.2	28.8

Table 3: Comparison of different implementations of k-means, in terms of difficulty of implementation, lines of code and speedup over C++ CPU variant. The Difficulty and Automatable columns represent our subjective appraisal of the level of programming challenge and whether or not a the set of GPU optimizations could be applied automatically by the Dandelion compiler.

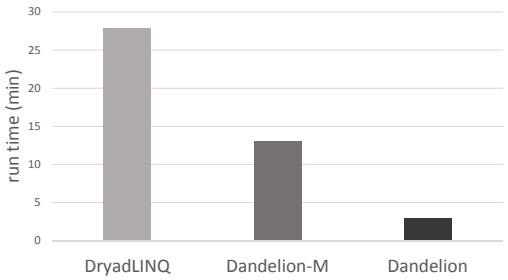

Figure 9: The performance of 5 steps k-means.

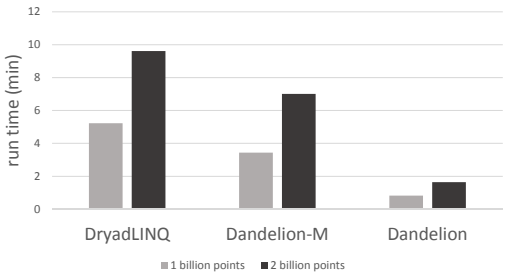

Figure 10: k-means data scaling

We next evaluate the k-means performance of Dandelion running five iterations (Figure 9), which make the benefits of caching datasets in memory apparent: Dandelion-M outperforms DryadLINQ by more than $2\times$. The speedup is primarily due to the effective use of caching to avoid reading the points dataset from disk at every iteration as DryadLINQ does. Dandelion with the GPU gives a further factor of $2\times$. The GPU speedup is less than in the single iteration evaluation, largely because garbage collection of PTask datablocks and graph structures between iterations introduces some synchronization.

We use the k-means benchmark to measure Dandelion's scalability by varying the size of the inputs. Figure 10 shows the performance of one step k-means on two datasets. We observe that the total running time is roughly proportional to the size of the input.

Overall system performance. Figure 11 shows the overall system performance of Dandelion on the four benchmarks, comparing runtime on the 10-machine cluster using a single thread per machine (Dandelion-S), all 24 cores (Dandelion-M), and using GPUs (Dandelion). For k-means and PageRank, the computations is for 5 iterations. The data show that while Dandelion is able to improve performance using parallel hardware in all cases, the end-to-end speedup can erode significantly due to I/O. Moreover, the degree to which GPU parallelism is profitable relative to CPU parallelism varies significantly as a function of Dandelion's ability to hide data marshalling, transfer, and memory management latencies with device-side execution. For example, Dandelion is able to achieve $65.6\times$ and $4.7\times$ speedups over Dandelion-S and Dandelion-M for k-means not just

name	Dandelion-S	Dandelion-M	Dandelion
kmeans-5x	10033s	723s	153s
PageRank-5x	272s	257s	190s
SkyServer	421s	126s	229s
Terasort	243s	101s	79s

Figure 11: Overall system performance

because the computation is overwhelmingly compute-bound, but because of its modest state requirements: only the centers list must remain in GPU memory (230K for the problem size reported here), while the points list can be streamed through GPU memory freely. This property combines with abundant GPU compute, to enable Dandelion to effectively hide almost all serialization costs and PCI transfer costs.

In contrast, SkyServer is in many respects a worst-case workload for Dandelion due to unmaskable marshalling costs and memory managment overheads. SkyServer features two joins, implemented in Dandelion as GPU hash joins which require one relation to be fully present in memory while the other relation may be streamed. This places the serialization from C# objects to GPU memory for one relation on the critical path. Intermediate state required for the computation is large relative to GPU memory, forcing PTask to perform frequent garbage collection to avoid exhaustion of GPU memory. In the SkyServer executions reported in Table 11, GPU memory pressure resulted in over 100 GC sweeps. The result is that Dandelion-M, which enjoys no serialization costs and abundant CPU memory, is $1.8\times$ faster for SkyServer. We argue that while some marshalling overheads are imposed by our design, their performance impact can be dramatically reduced through engineering effort. By contrast, memory management overheads are imposed by the hardware. While our cluster comprises latest-generation GPU and CPU hardware, nodes in our cluster have over $50\times$ more CPU memory than GPU memory. Our experience with Dandelion suggests that for GPUs to be more generally applicable to more "big-data" workloads, a better balance of GPU to CPU memory is necessary, regardless of whether such systems rely on front-end programming and runtime systems like Dandelion.

6 Related work

General-purpose GPU computing. The research community has focused considerable effort on the problem of general-purpose programming interfaces for GPUs and other specialized hardware [6, 62, 84, 28, 34]. GPU frameworks such as CUDA [79], OpenCL [63], and others [88, 17, 71, 22, 48, 94] provide rich front-end programming models, specific to the underlying parallel GPU architecture. Dandelion, provides a high-level managed programming model that generalizes well across diverse execution environments and exposes the programmer to comparatively little architectural detail.

OS-support for GPUs and peripherals The Dandelion prototype relies on user-mode PTask, making OS-level support a largely orthogonal concern. A production implementation of Dandelion could benefit significantly from kernel- or hypervisor-level support for GPUs to enable fairness and/or isolation guarantees [85, 47] in the presence of cluster sharing, or to optimize data movement with demand-driven data movement to the GPU [89] or with zero-copy I/O [36].

Dataflow and streaming. execution directly in hardware. Click [65], CODE2 [76], and P-RIO [68] provide graph-based programming models but do not generalize across diverse hardware. Self/star [42] and Dryad [57] are graph-based programming models for distributed execution, the latter of which is extended by DryadLINQ [101] to support LINQ. Dandelion provides the same programming abstraction over a cluster of heterogeneous compute nodes: DryandLINQ can use only CPUs. Dandelion's support for caching of intermediate data in RAM is similar to the RDDs used in Spark [103].

FFPF [20] provides support for compiling code written in different languages such as FPL [32, 31] and Ruler [54] to heterogeneous targets, composing the result in a graph-like structure. Dandelion shares many basic objectives and similarly relies on dataflow to address heterogeneity; however, Dandelion targets GPU-based clusters and general-purpose computations, making the application and execution domains quite different. FFPF later evolved into Streamline [37], which details I/O optimizations and front-end programming model techniques that could benefit Dandelion's GPU engine, PTask, significantly. PTask requires a programmer (or the Dandelion compiler) to construct dataflow graphs explicitly through API calls, while Streamline enables a much more compact and easily encapsulated graph construction front end that manipulates graph objects through the file system interface. PTask implements many of the copy-minimization techniques described by FFPF and Streamline, however the presence of disjoint, non-coherent memory spaces required to target GPUs requires a coherence protocol not required by these systems.

GPUs and Dataflow. StreamIt [93] and DirectShow [67] support graph-based parallelism. OmpSs [23], Hydra [96], and PTask [85] all provide a graph-based dataflow programming models for offloading tasks across heterogeneous devices. IDEA [33] extends PTask to support cyclic and iterative graph structures required to efficiently support relational operators on GPUs. Liquid Metal [55] 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 as a pipeline. Flextream [53] 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. Unlike these systems,

Dandelion does not directly expose the graph-based execution model. Extending dataflow systems to support iteration [21, 40, 70, 75] or incremental iterative computation [72, 41, 73, 74] is an active research area. Dandelion provides front support for expressing iteration that it maps to the control flow constructs in PTask.

Front-end programming models Many systems provide GPU support in a high-level language: C++ [45], Java [99, 8, 81, 24], Matlab [7, 80], Python [25, 64]. While some go beyond simple GPU API bindings, and provide support for compiling the high-level language to GPU code, none have Dandelion's cluster-scale support; unlike Dandelion, all expose the underlying device abstraction. Merge [66] provides a high-level language based on MapReduce that transparently chooses amongst implementations dynamically, it is targeted at a single machine unlike Dandelion. Many workloads we evaluate have enjoyed research attention in a single-machine GPU context [39, 60] or cluster-scale GPU implementations [86], using GPU programming frameworks directly; Dandelion represents a new level of programmability for such platforms and workloads.

Dandelion explores an automatic parallelization problem that has challenged the research community for decades, and in its current form, relies entirely on language-level support in .NET for relational algebra via LINQ to identify code regions that may be safely parallelized. Liszt's [38] static meshes, Halide's [82] coordinate spaces, and Legion's [16] logical regions provide runtimes and compilers with similar leverage for identifying automatically parallelizable code: all enable high-level programs to be parallelized and compiled to different target architectures. All define new front-end programming models and language, while Dandelion relies on model and language support that have been present in production tool-chains. Moreover, Liszt, Halide, and Legion all rely on static schedules: Dandelion supports dynamic scheduling and dynamic re-targeting of computations based on the available resources at each node in a heterogeneous cluster.

Scheduling and Execution engines for heterogeneous processors. Scheduling for heterogeneous systems is an active research area: systems such as PTask [85], TimeGraph [61] and others [95] focus on eliminating destructive performance interference in the presence of GPU sharing. Maestro [90] also shares GPUs but focuses on task decomposition, automatic data transfer, and auto-tuning of dynamic execution parameters in some cases. Sun et al. [91] share GPUs using task queuing. Others focus on making sure that both the CPU and GPU can be shared [59], on sharing CPUs from multiple (heterogeneous) computers [14, 15], or on scheduling on multiple (heterogeneous) CPU cores [18, 13]. Several systems [69, 46] automatically choose whether to send jobs to the CPU or GPU [11, 12, 10, 30, 87], others focus on support for scheduling in the presence of heterogeneity in a cluster [23]. Several systems consider support a MapReduce primitive on GPUs, taking care of scheduling the various tasks and moving data in and out of memory [49, 26, 97]. The same abstraction can be extended to a cluster of machines with CPUs and GPUS [58]. This work was later improved with better scheduling [83]. Teodoro et al. describe a runtime that accelerates the analysis of microscopy image data sets on GPU-CPU clusters [92]. Like Dandelion, the resulting system relies on task-level dataflow to map the application to a heterogeneous platform, and requires support for cyclic structures in the di-graphs that express the image-processing pipeline; unlike Dandelion the system is absent the front-end language support enabling a programmer to express the application in a high-level, declarative language.

Relational Algebra on GPUs The Thrust [77] library, included with CUDA, offers some higher level operations like reduces, sorts, or prefix sums. Dandelion uses the sort and scan primitives from Thrust library in several higher-level relational primitives. Others have explored primitives for supporting GPU-based relational algebra [51, 50] and database operations [43], with emphasis on optimization and performance for operations over a very limited set of data types (primitive types).

7 Conclusion

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

Dandelion is a research prototype under active development. The design goal of Dandelion is to build a complete, high performance system for a small to medium 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.

8 Acknowledgements

We thank Igor Ostrovsky for sharing his GPU Kernelizer work, which served as a starting point for our cross-compiler. We thank Herman Venter and Mike Barnett for answering CCI related questions. Thanks also to the SOSP review committee and our shepherd Herbert Bos for valuable feedback.

References

[1] Apache YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

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

[3] The LINQ project. http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx.

[4] The PLINQ project. http://msdn.microsoft.com/en-us/library/dd460688.aspx.

[5] Sort benchmark home page. http://sortbenchmark.org/.

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

[7] Matlab plug-in for CUDA. https://developer.nvidia.com/matlab-cuda, 2007.

[8] JCuda: Java bindings for CUDA. http://www.jcuda.org/jcuda/JCuda.html, 2012.

[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*, 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] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[17] 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.

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

[19] B. Billerbeck, N. Craswell, D. Fetterly, and M. Najork. Microsoft Research at TREC 2011 Web Track. In *Proc. of the 20th Text Retrieval Conference*, 2011.

[20] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *Proceedings of OSDI'04*, 2004.

[21] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[22] 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.

[23] 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.

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

[25] 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, 2011.

[26] 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.

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

[28] 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.

[29] E. Chung, J. Davis, and J. Lee. Linqits: Big data on little clients. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013.

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

[31] M.-L. Cristea, W. de Bruijn, and H. Bos. Fpl-3: towards language support for distributed packet processing. In *Proceedings of IFIP Networking 2005*, 2005.

[32] M. L. Cristea, C. Zissulescu, E. Deprettere, and H. Bos. Fpl-3e: towards language support for reconfigurable packet processing. In *Proceedings of the 5th international conference on Embedded Computer Systems: architectures, Modeling, and Simulation*, SAMOS'05, pages 82–92, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] J. Currey, S. Baker, and C. J. Rossbach. Supporting iteration in a heterogeneous dataflow engine. In *SFMA*, 2013.

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

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

[36] W. de Bruijn and H. Bos. Pipesfs: Fast linux i/o in the unix tradition. *ACM SigOps Operating Systems Review*, 42(5), July 2008. Special Issue on R&D in the Linux Kernel.

[37] W. de Bruijn, H. Bos, and H. Bal. Application-tailored i/o with streamline. *ACM Trans. Comput. Syst.*, 29:6:1–6:33, May 2011.

[38] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[39] 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, 2012.

[40] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*. ACM, 2010.

[41] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *VLDB*, 2012.

[42] 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.

[43] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, 2005.

[44] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. Vandenberg. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, pages 189–210, Paris, France, March 2002. Carleton Scientific. also as MSR-TR-2002-01.

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

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

[47] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

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

[49] 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, 2008.

[50] 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.

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

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

[53] 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, 2009.

[54] T. Hruby, K. van Reeuwijk, and H. Bos. Ruler: high-speed packet matching and rewriting on npus. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 1–10, New York, NY, USA, 2007. ACM.

[55] 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.

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

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

[58] 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.

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

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

[61] 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*, 2011.

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

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

[64] A. Kloeckner. pycuda. `https://pypi.python.org/pypi/pycuda`, 2012.

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

[66] 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.

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

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

[69] 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, 2009.

[70] 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. In *SIGMOD*. ACM, 2010.

[71] 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, 2006.

[72] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[73] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: recursive, delta-based data-centric computation. *Proc. VLDB Endow.*, 5(11):1280–1291, July 2012.

[74] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. *SOSP*, 2013.

[75] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed dataflow computing. In *NSDI*, 2011.

[76] 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, 1992.

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

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

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

[80] 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, 2011.

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

[82] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[83] 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, 2012.

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

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

[86] 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, 2012.

[87] 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*.

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

[89] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating file systems with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[90] 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.

[91] 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, 2012.

[92] G. Teodoro, T. Pan, T. Kurc, J. Kong, L. Cooper, N. Podhorszki, S. Klasky, and J. Saltz. High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms. 2013.

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

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

[95] U. Verner, A. Schuster, and M. Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 120–129, New York, NY, USA, 2011. ACM.

[96] 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*.

[97] 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.

[98] 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.

[99] 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.

[100] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

[101] 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.

[102] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan. Some sample programs written in DryadLINQ. Technical Report MSR-TR-2008-74, Microsoft Research, May 2008.

[103] 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.

[104] H. Zaragoza, N. Craswell, M. Taylor, S. Saria, and S. Robertson. Microsoft Cambridge at TREC-13: Web and HARD tracks. In *Proc. of the 13th Text Retrieval Conference*, 2004.