University of Washington
Department of Computer Science and Engineering
Undergraduate Senior Thesis

The Elan Programming Language for Field-Programmable Gate Arrays

By Elliott Brossard

Advised by Professor Carl Ebeling

June 4, 2012

Presentation of work given on June 4, 2012

Thesis and presentation approved by: _____

Date: _____

Contents

1 Introduction

Field Programmable Gate Arrays (FPGAs) give programmers the ability to write highly optimized parallel and pipelined applications that take full advantage of the hardware and memory on which they run, yet with this freedom generally comes the burden of using a low-level language such as VHDL or Verilog to program them. High-level synthesis tools such as AutoESL[1], Impulse C[2] and Catapult C[3] help to abstract the burden of using such languages through intelligent front-ends and code generation, but they do not reach the level of providing a high level language and compiler capable of targeting multiple platforms while still capitalizing on the characteristics of each. Chapel, which is a language being developed by Cray, Inc. for large-scale parallel computers[4], espouses a programming model that allows for the clean separation of algorithm and hardware characteristics while still providing the ability to configure the application to leverage the full capabilities of the target platform through caching and locality. While Chapel has many of the characteristics that would be desirable to FPGA application developers, FPGAs are different enough from Chapel's supported systems that an attempt at directly porting the language is unrealistic. In this paper, we formalize and present the Elan language, which is similar in paradigm to Chapel but is targeted to FPGAs, and we identify and discuss the tradeoffs between the two languages.

2 Goals

The goals of the Elan programming language are relatively simple, and are quite feasible as we shall show over the course of this paper. In the Elan language, we first seek to enable developers to write sequential object-oriented code in the sense of objects that store state and provide functions that other objects can use to invoke actions. This will make the prototyping of applications much faster than in widely used hardware description languages like VHDL and Verilog, and will make it easier to write correct, testable code as with high-level software languages. As an added benefit, it will be possible to emulate applications written in the Elan language in software, since the semantics of the language are expressible in languages such as C++ and Java that have compilers (and JVM support, in the case of Java) for most architectures.

We also seek to be able to describe parallel algorithms without tying them to a specific platform, which is extremely useful to developers in the setting of distributed applications. By separating algorithm specifics from hardware specifics, we will make it much easier to port applications between platforms and to test and debug applications across configurations. This is one of the main draws of Chapel, which uses a variety of constructs such as distributions and locale-aware structures to simplify the descriptions of algorithms while preserving their parallelism.

In the Elan language, we will provide a distributed memory system that is platform agnostic and provides useful features such as read/write ordering, atomic read/modify/write, and barriers while making use of the parallelism of the hardware. The performance of distributed applications on FPGAs relies heavily on the efficient use of memory, so in implementing such a system, we will expose a clean yet flexible enough memory interface such that it can easily be incorporated into sequential, object-oriented code and still allow for full maximization of memory bandwidth across available ports.

Some of the specific applications that we take as use-cases for Elan are sparse-matrix multiplication, breadth-first search, and short read alignment. These applications all have high memory throughput requirements and require different levels and types of parallelism to be efficient.

[1] http://www.xilinx.com/products/design-tools/autoesl/index.htm
[2] http://www.impulseaccelerated.com/products_universal.htm
[3] http://www.mentor.com/esl/catapult/overview
[4] http //chapel.cray.com/

3 Compilation and execution of Elan code

Having written an application in the Elan language, a developer can compile the source code into Verilog, which can be synthesized onto the target FPGA(s). The developer can then start the application, which begins with an initialization phase and proceeds to a running phase. In this paper, we distinguish between *compile-time*, which is the phase during which Elan code is being compiled into Verilog, from *initialization*, which is the phase during which data structures are being configured with runtime parameters, from *runtime*, which is the phase that begins with the main method of the application being invoked and ends with a shutdown signal.

The global `init` function is invoked by the Elan framework during the initialization phase. `config` variables (see Section 4.1) are assigned values based on command-line parameters immediately before this function is executed, after which the `init` function may perform any other initialization needed. The `init` function may also invoke an external program with references to its arrays (see Section 6) in order for the arrays to be populated with the data set prior to execution. The `main` function is invoked at runtime, and commences execution of the application.

4 Elan language constructs and syntax

The Elan language provides constructs that will be familiar to C++ and Java programmers, albeit with some slight differences. Elan is statically-typed, like Java, and does not provide for the use of pointers to memory. Elan programs interface with memory comprising local registers, block RAM, and DRAM through the use of primitive types, modules, and arrays, the latter of which is the only means through which an FPGA's DRAM can be accessed. Control flow in Elan is derived from the common mechanisms of C, C++, and Java: `for`, `while`, `do ... while`, `if`, `else if`, `else`, `switch`, `case`, `default`, `break`, and `continue` all have the same meaning and syntax as in those languages. See Appendix C for a full list of Elan's reserved words.

4.1 Variables

In the Elan language, the programmer may store primitive types, module references, array references, and distribution references in variables, all of which have a phase type associated with them. They may be declared as `compile`, in which case they must be assigned a value at compile-time, or `config`, in which case they must be assigned a value during application initialization. Variables not declared with either of these qualifiers will take on the earliest possible phase type, or `runtime` if they can be written during execution of the program, where the `runtime` phase type designates that their values may change over the course of execution. This gives us the ordering `compile` ⇒ `config` ⇒ `runtime` in direct correspondence with *compile-time* ⇒ *initialization* ⇒ *runtime*, where variables may only be assigned values in phases up to the one corresponding to the variable's phase type. Integer, boolean, and fixed point floating point constants (e.g. -5.78) are considered to have a `compile` phase type.

4.2 Primitive types

Primitive types include integers, booleans, fixed point floating point numbers, structs, and local arrays. Integers may be of any width greater than or equal to a single bit, and can be either signed or unsigned. The default value of integers, if none is assigned, is 0. Booleans are single-bit unsigned integers that take a value of `false` or `true`, which are represented in memory as 0 and 1, respectively, and have a default value

of `false`. Floating point numbers include single precision (float) and double precision (double) values, and have a default value of +0. Structs may contain any positive number of primitive types up to any level of nesting and may define simple accessors and mutators. Local arrays can store any primitive type and may be of any dimension so long as they fit in local registers or block RAM. Local arrays must always be declared with a statically-determined size, and can be thought of as structs storing some fixed number of the same primitive type accessible by index.

4.2.1 Integer variables

To declare an integer type variable, one specifies the phase type, signedness, width, and name of the variable and initial value of the same or earlier phase type, where the phase type, signedness, and initial value are optional:

```
[compile|config] [unsigned] int:width identifier [= value];
```

The width of the integer must be a compile-time constant, which can be either an identifier or a numerical value. If the integer is signed, its highest bit is used as a sign bit, and the other $width - 1$ bits are used for the integer's value, as stored in two's complement form. If a signed integer variable is assigned an unsigned value, only $width - 1$ bits of the unsigned value will be used in the assignment. Similarly, if a signed integer variable is assigned a signed value of greater width, the sign bit will be preserved but only $width - 1$ bits of the value will be used in the assignment. If a signed value is assigned to an unsigned integer variable, the sign bit is dropped and at most $width$ bits from the value are used in the assignment. If a floating point value is assigned to an integer variable, its value is first truncated to be a signed integer (treating -0 as integer value 0) before performing the actual assignment following the above rules.

4.2.2 Boolean variables

Boolean variables have a somewhat simpler form than their integer counterparts:

```
[compile|config] boolean identifier [= value];
```

Here `value` must be either `false` or `true` if specified. Assignment of boolean types to integer variables and vice versa is not permitted; instead, use logical statements such as `booleanValue = intValue == 1` and `intValue = booleanValue ? 1 : 0` to convert between them.

4.2.3 Floating point variables

To declare a floating point variable, one specifies the phase type, floating point type, name, and initial value of the same or earlier phase type, where the phase type and initial value are optional:

```
[compile|config] float identifier [= value];
[compile|config] double identifier [= value];
```

4.2.4 Struct definition and declaration

To use a struct, one must first define it. The definition of a struct can reside either in the outermost scope of a source file or inside the scope of a module, in which case it will only be possible to refer to the struct in the scope of the module itself. Struct definitions are of the following form:

```
struct identifier {
  variable-list
  function-list
}
```

In the variable list, variables may not be declared with a phase type, as the phase type of all variables inside the struct is determined by the phase type with which the struct is declared. The function list may include only accessors or mutators; in the sense of synchronous and asynchronous functions that will be discussed later, struct functions are all synchronous. Having defined a struct, one can declare a struct as follows:

```
[compile|config] struct identifier;
```

As an example, one could declare two structs as follows:

```
struct Inner {
  unsigned int:32 countA;
  unsigned int:32 countB;

  unsigned int:32 getCountA() {
    return countA;
  }

  unsigned int:32 getCountB() {
    return countB;
  }
}
struct Outer {
  int:8 index;
  Inner inner;

  int:8 getIndex() {
    return index;
  }

  void setIndex(int:8 index) {
    this.index = index;
  }

  Inner getInner() {
    return inner;
  }
}
```

To declare and assign values to the fields of an instance of the `Outer` struct, one could write:

```
Outer outer;
outer.setIndex(0);
outer.inner.countA = 3;
outer.inner.countB = 2;
```

It is important to note that `outer.getInner()` returns `inner` by *value*, not by reference. That is, the expression `outer.getInner()` returns a copy of `inner`, whereas `outer.inner` references `outer`'s actual `inner` struct variable.

Structs can be compared with the comparison operators `<`, `<=`, `==`, `>=`, and `>` if they define a `compareTo` function that takes another struct of the same type as an argument and returns a signed integer value:

```
struct s {
  int:32 compareTo(s other) {
    // Return:
    // < 0 if this < other,
```

```
    // 0 if this == other, and
    // > 0 if this > other.
  }
}
```

The contract of `compareTo` is that if `a.compareTo(b) < 0`, then `b.compareTo(a) > 0`, if `a.compareTo(b) == 0`, then `b.compareTo(a) < 0`, and that these relations are transitive.

4.2.5 Local array variables

Declarations of local arrays have the following form:

```
[compile|config] type[len];
```

`len` must be either a positive integer constant or a `compile` or `config` phase type unsigned integer. If the local array is of `compile` phase type, it may only be assigned values in the outermost scope of the source code. If it is of `config` phase type, it may be assigned values either in the outermost scope of the source code or in the initialization block of the application. Arrays of `runtime` phase type may be assigned values during any phase of compilation or execution. Each local array has a `length` associated with it, which is accessed as a field of the array, such as with `arr.length`.

To declare and assign values to an array, one could write:

```
compile unsigned int:32 arr[20];
for (compile int:32 i = 0; i < arr.length; ++i) {
  arr[i] = i;
}
```

4.3 Objects

There are two types of objects in Elan: modules and arrays. Instantiations of either must be static, and modules and arrays must be initialized through their constructors exactly once either globally, inside of the application's entry point (main method), or inside the constructor of a module. For example, a module may be instantiated in the global address space of the application and then initialized with parameters determined at runtime, such as the size of the application's data set.

4.3.1 Module declarations

Modules are declared with an identifier and optionally a distribution map, which specifies how to parallelize the module (see Section 7). Module declarations are of the following form:

```
module identifier [maps distribution-identifier] {
  struct-definition-list
  variable-list
  constructor-list
  function-list
}
```

4.3.2 Module state

Modules may store local state in the form of primitive types, module references, and array references, though they may not store actual modules or arrays themselves. Module state may be accessed and modified only by the module itself, such as through its constructor and functions, and not by external modules, arrays, or functions.

4.3.3 Module constructors

Modules may have zero or more constructors, which may be called either globally or inside of the application's entry point, and hence are evaluated at compile-time. Constructors may assign values to the module's local state, but may not call internal or external functions. Any number of primitive types, module references, and array references may be accepted by a module constructor as arguments. Any local state not assigned a value in a constructor is assigned its default value, or null for module and array references. If a module does not define a constructor, then it implicitly defines a constructor with no arguments and no body. For example, one could define a module like this:

```
module Example {
  struct SampleStruct {
    unsigned int:16 index;
    int:32 value;
  }

  boolean toggle;
  SampleStruct sample;

  Example(boolean toggle, unsigned int:16 index, int:32 value) {
    this.toggle = toggle;
    sample.index = index;
    sample.value = value;
  }

  Example(boolean toggle) {
    this.toggle = toggle;
    // sample's fields will be assigned their default values.
  }
}
```

Module declaration and construction occur separately: declaring a module allocates space for it, while construction performs the initialization of its fields. The special `construct` keyword is used to invoke a module's constructor:

```
identifier.construct(argument_1, argument_2, ..., argument_n);
```

To use the `Example` module defined above, one could declare and construct an instance of it in the global scope as follows:

```
Example example;
example.construct(false, 10, -47);
```

4.3.4 Module templating

To ease the writing of modules to support multiple data types, developers may parametrize modules with any number of template parameters, which may be primitives types or reference types such as other modules or arrays. Module templates take the following form:

```
module identifier<template-type1, ..., template-type2> {
  ...
}
```

Here "template" takes the meaning of C++ templates; it is valid to write a templated module that performs array indexing on its templated type and to declare an instance of the module with a local array type of a fixed size as a template parameter, but it is invalid to declare an instance of such a module with a boolean as a template parameter, since array indexing is undefined for boolean types. As an example, one could define a templated module and use it as follows:

```
module Queue<type> {
  unsigned int:32 currentSize;
  type queue[];
  Queue(unsigned int:32 queueCapacity) {
    currentSize = 0;
    queue = type[queueCapacity];
  }

  async push(type v) {
    if (currentSize + 1 < queue.length) {
      queue[currentSize++] = v;
    } // else call a function to report an error
  }

  sync type pop() {
    if (currentSize != 0) {
      type temp = queue[0];
      for (unsigned int:32 i = 0; i + 1 < currentSize; ++i) {
        queue[i] = queue[i + 1];
      }
      currentSize--;
      return temp;
    } else {
      // Report an error, then return a default value
      type t;
      return t;
    }
  }

  async clear() {
    currentSize = 0;
  }
}

Queue<double> q;
q.construct(64);
q.push(5.234);
// Execution will block until this call completes.
q.push(-4.79351).barrier();
double val = q.pop();
```

The syntax and semantics of functions are discussed in Section 5, but the main idea is that parametrized types can be used just like any other type within a module. In this example, the queue itself provides no synchronization mechanisms, but users of it may synchronize their accesses through `barrier` (see Section 5.4), and it is fully capable of storing any primitive type including local arrays of a fixed size.

5 Functions

Functions may be declared either in the global scope or in the scope of a module, and take zero or more parameters as arguments, which may be primitive types, module references, or array references. Primitive types are always passed by value as function arguments. Functions may read and modify local module state (if they are contained within a module), make function calls to other modules through module references, and read and write to arrays through array references. Functions can either be asynchronous, in which they may not have a return type and always return control to the caller immediately, or synchronous, in which case they may have a return value and the caller has to wait until the function completes until continuing. Functions are declared in any of the following formats:

```
async identifier(type_1 arg_1, ..., type_n arg_n) {
  statement-list
}

sync type identifier(type_1 arg_1, ..., type_n arg_n) {
  statement-list
}

inline type identifier(type_1 arg1, ..., type_n arg_n) {
  statement-list
}
```

Here `type` and all `type_i` specifiers are primitive types, where `type` may be `void` for `sync` functions. In `async` functions, or in `async` functions if `type` is `void`, the function may not return a value but can explicitly exit at any point through a `return` statement with no argument. In `sync` functions, if `type` is non-`void` (`boolean` or `int:32` for example), the function is required to have a `return` statement with an argument whose type agrees with `type`.

5.1 Asynchronous functions

Asynchronous functions are what drive parallelism in the Elan language, allowing a callee to initiate multiple simultaneous actions such as memory accesses and function calls. An example of their use is as follows:

```
unsigned int:32 sum = 0;
async addValue(unsigned int:32 value) {
  sum += value;
}

async sumValues() {
  unsigned int:32 values[32];

  // ... assign values to the entries in the array.

  for (unsigned int:16 i = 0; i < values.length; ++i) {
    addValue(values[i]);
  }

  // Control may reach here before all calls to addValue
  // have completed.
}
```

The `async` keyword denotes functions that are asynchronous: the caller continues immediately, regardless of how long the function itself takes to execute. A first-in-first-out queue tracks calls to asynchronous functions so that when a caller invokes an asynchronous function, the callee removes a set of arguments from the queue and executes with them. This has the implication that only a single instance of a function executes at once,

so if multiple callers invoke a function in a short span of cycles, all callers will continue execution while the callee processes call invocations one after another until its call queue is empty, at which point it sleeps until receiving another call in its queue.

In the above example, `sumValues` invokes `addValue` 32 times in total, yet control may reach the end of the function before all calls to `addValue` have completed since some fraction of the calls may still be queued at that point, depending on how long execution of the function takes in comparison to adding a call to the call queue. Note that the parallelism of the above example comes from the ability to initiate many different calls to `addValue` and then to execute some other logic at the same time; the `addValue` method itself has no parallelism. Distributions (see Section 7) overcome this limitation by creating multiple instances of functions.

As the reader may notice in the above example, invoking `addValue` 32 times sequentially could potentially fill a call queue of that size or smaller. Call queues are of a fixed size, so if the call queue for a function fills, the caller will block. This could potentially lead to deadlock depending on data dependencies, so the developer may have to increase the size of the call queue for specific functions for some applications.

5.2 Synchronous functions

```
sync int:32 computeResult(int:32 a, int:32 b) {
  int:32 result = 0;

  // ... some long sequence of steps to compute a result.
  return result;
}

async doWork(int:32 a, int:32 b) {
  // ... do some work.
  sendNotification(a, b);

  // Wait for the result of a computation. This will
  // block until computeResult completes.
  int:32 result = computeResult(a, b);

  sendResult(a, b, result);
}
```

With synchronous functions, the caller must wait for the called function to complete before continuing. Synchronous functions do have call queues like asynchronous functions, which enforce first-in-first-out processing order, and similarly process a single call at a time. Synchronous functions signal completion to the caller once they finish, which then permits the caller to continue its execution, and they send with the signal a return value if `type` is non-`void`. In the above example, `doWork` blocks while waiting for the result of `computeResult`. Synchronous calls are mostly useful for initialization of module state and separation of logic, since intra-module calls that merely assign starting values prior to execution, for instance, can be inlined by the compiler and have the same effect while shedding the overhead of call invocation.

5.3 Inline functions

```
inline int:32 max(int:32 a, int:32 b) {
  return a > b ? a : b;
}
```

`inline` functions are simple functions that compute a value based on their inputs and return the result. Inline functions may only refer to variables outside their scope if the variables are of phase type `compile` or `config`, and are automatically inlined by the compiler at the time of compilation or initialization.

5.4 Barriers

Barriers provide a useful way to synchronize execution across multiple stages of computation. The `barrier()` function may be invoked on any asynchronous function call and has the effect of blocking execution of the caller until the function call and all functions invoked on its behalf complete. This can be useful, for example, when one module produces some amount of work and passes it to another module that filters the work before recording the result. The first module can wait until all steps in this process have completed by issuing a barrier on the first call that started it. In code, this might look like the following:

```
struct WorkItem {
  // ...
}

module WorkProducer {
  WorkConsumer workConsumer;
  WorkItem workItem[];

  WorkProducer() {
    // ...
  }

  async start() {
    while (/*there are more phases to do*/) {
      produceWork().barrier();
    }
  }

  async produceWork() {
    for (unsigned int:32 i = 0; i < numWorkItems; i++) {
      workConsumer.consumeWork(workItem[i]);
    }
  }
}

module WorkConsumer {
  async consumeWork(WorkItem workItem) {
    if (/*some condition is met*/) {
      recordResult(workItem);
    }
  }

  async recordResult(WorkItem workItem) {
    // ...
  }
}
```

Invoking a barrier on the `produceWork` function allows the `start` function to synchronize its work phases in accordance with when the function calls invoked as part of a phase complete.

The implementation of barriers requires work-accounting for every call initiated as part of a barrier; the `produceWork` function tracks how many times it invokes `consumeWork` before returning and reports this count to `consumeWork`, and similarly `consumeWork` tracks how many times it invokes `recordResult` and then reports this count to `recordResult` once it returns. Having completed all expected invocations of `recordResult`, the `recordResult` function reports to `consumeWork` that it has finished. Once `consumeWork` learns from all the functions it called during execution that they have completed, which in this case is just `recordResult`, and has completed all of its expected invocations itself, it notifies `produceWork` that it is finished. Once `produceWork` learns that all invocations of `consumeWork` have completed, it reports this to the `start` function, which now continues execution. Since different barriers may involve the same functions, barrier-related messages should be tagged with an identifier and sequence number of the originating function in order to ensure correct routing.

This process becomes much more complicated in other scenarios, but the idea of each function tracking the functions it invokes and when they complete is the same. Arrays, which are introduced in Section 6, must also facilitate barriers by forwarding information to the callback function (if any) about the caller so that it can properly report back once it completes, and must themselves track how many calls were routed to which distributed modules instances, introduced in Section 7.

In addition to barriers, fences are a planned feature of the Elan language. Their syntax and semantics have not been finalized, but they overcome the restriction that a function may only execute a single barrier at a time. Fences block the caller until calls at some level of execution have completed, at which point another fence is allowed to proceed. This is useful when only the last few levels of a call tree/graph require synchronization but not the rest.

5.5 Function templating

Much like modules, functions may also be templated. Function templates take the following form:

```
async identifier<template-type1, ..., template-typen>(...) {
  ...
}

sync type identifier<template-type1, ..., template-typen>(...) {
  ...
}

inline type identifier<template-type1, ..., template-typen>(...) {
  ...
}
```

These template types may be primitive types or reference types including modules and arrays, and may be partially specified by templated type(s) from an enclosing module. Local arrays used as templated types must have a fixed size. Returning to the inline function example, one could alternatively write and invoke a `max` function as:

```
inline type max<type>(type a, type b) {
  return a > b ? a : b;
}

highScore = max<unsigned int:32>(score, highScore);
```

A caveat of templated functions is that an asynchronous or synchronous function referenced twice with different types will actually create two separate instances of the function; with a templated asynchronous function, for example, two functions with separate call queues will be created.

6 Arrays

Implementing a strong memory model for a new platform is an expensive undertaking time-wise, and some of the memory features present on one platform might not be available on another. The Elan language addresses this problem by providing a single memory model that is flexible enough to accommodate a variety of needs while at the same time espousing efficiency. The tradeoff is that such a memory model can never hope to be as efficient as a hand-crafted platform-specific model, but the hope with Elan is that such a tradeoff will easily be outweighed by the speed with which applications can be built.

Arrays in Elan provide a high-performance, statically-typed interface to raw memory. Each array must be

parametrized by a primitive type and optionally a distribution map, as we discuss in Section 7. Arrays are a defined as a standard Elan module whose precise implementation is platform-specific, yet they expose `read`, `write`, `writeNotify`, `writeFlush`, and `atomicUpdate` functions with the same semantics, as given in pseudocode below, across all platforms:

```
module Array<type> {
  type array[];
  Array(unsigned int:64 length) {
    array = type[length];
  }

  async read(unsigned int:64 index, async(type) callback) {
    assert(index < array.length);
    callback(array[index]);
  }

  async write(unsigned int:64 index, type value) {
    assert(index < array.length);
    array[index] = value;
  }

  async writeNotify(unsigned int:64 index, type value, async() callback) {
    assert(index < array.length);
    array[index] = value;
    callback();
  }

  async writeFlush(unsigned int:64 index, type value, async() callback) {
    assert(index < array.length);
    array[index] = value;
    array.flushWrites();
    callback();
  }

  async atomicUpdate(unsigned int:64 index,
                     inline<type>(type) updateFunction,
                     async(type) callback) {
    assert(index < array.length);
    type previousValue = array[index];
    array[index] = updateFunction(previousValue);
    callback(previousValue);
  }
}
```

Here `type` is the parametrized type, which is provided upon instantiation of Array and as previously stated may be any primitive type. The `read` function fetches the value of the array from the indicated index and passes its value back to the specified asynchronous callback function. `write` simply writes a given value to the specified index, while `writeNotify` performs a write and then invokes a callback to indicate that the write completed. `writeFlush` performs a write operation, flushes all pending writes, and then invokes a callback to indicate completion. Finally, `atomicUpdate` performs an atomic update on the value stored at the indicated location using the given `updateFunction`, then returns the previous value via the callback function.

The `inline` and `async` function notation used above indicates the prototypes expected for those parameters. `inline<type>(type) updateFunction`, for example, refers to an `inline` function that returns a result of type `type` as specified by `<type>` and takes a parameter of type `type` as specified by `(type)`. As a caveat, the `async` callback functions used above are passed by reference by the callee, whereas the `inline` update function is passed by value, so it is actually evaluated at the memory interface.

With this form of the array that does not use a distribution map, instantiating an array ties it to a single read/write port (or one read and one write port, if the platform separates read from write ports) to DRAM

on the FPGA, which may have to be shared with other arrays depending on system constraints. All reads and writes to a particular index on a particular port are sequentialized, and an `atomicUpdate` operation blocks writes until it has read the previous value, computed the new value, and written the new value to the array. A caveat is that in arrays whose elements are larger than the platform's block size lose the guarantee of consistency; since such writes actually require multiple write operations, a read operation performed while the write is still in flight may return inconsistent data. This can be mitigated in part by using `writeFlush` before attempting to read any data larger than the block size from a section of the array that is thought to have been written to recently, although having to wait for a flush might be undesirable in terms of performance. Checksums can also be used to validate that large reads from an array return consistent data, though this introduces a computational overhead and the potential to have to re-issue reads, which is also costly.

One useful feature that the Elan memory model does not support is range-based and index-based locking of arrays. In some applications, it may be desirable to prevent other modules with a port to an array from writing to it while one module is updating its contents. Such a scenario might arise when the elements of the array are larger than the memory block size, when there are interdependencies between elements of the array, or when memory is the main communication medium between multiple FPGAs and the application requires exclusivity of some sort. In the former two cases, the modules involved should communicate to establish the module currently permitted to write to the array and when it is safe to read data. In the latter case, the `atomicUpdate` function can be used on a shared array to acquire a (spin-)lock, and the other FPGAs involved are designed such that they cannot write and/or read certain indexes while the lock is held by another module.

6.1 Array declaration

Declaration of arrays is similar to the declaration of modules. Arrays must be constructed at compile- or initialization-time, so their size must be of phase type `compile` or `config`. To declare an array, one simply specifies the type and size:

```
1 compile unsigned int:64 numberOfSums = 1000;
2 Array<int:32> sums;
3 sums.construct(numberOfSums);
```

The array can now be accessed through its functions like any other module.

6.2 Array ports

When an array is declared, it is assigned a single port through which it can be accessed, which will be shared with all functions that invoke the array's functions. For higher throughput, one may create additional ports to an array by invoking the `createNewPort` function on it:

```
1 compile Array<int:32> secondSums = sums.createNewPort();
```

Here `createNewPort` creates a new port to the `sums` array, which `secondSums` can now access.

6.3 The array read result

The special `_result` variable is a reference to the data that will be read back from invoking `read` on an array. It can be used in combination with inline arithmetic or an inline function to manipulate values used

in the function callback or even the callback itself. For example, one could route a callback to one module if the read result is even, or another module if the result is odd:

```
array.read(index, _result % 2 == 0 ? &firstFunc() : &secondFunc());
```

Functions involving the `_result` variable are evaluated at the array interface itself after reading the value from it, which enables this somewhat-dynamic behavior. The `_result` variable can also be combined with distributed module function calls (see Section 7.2) to route the callback to a module instance in correspondence with the value read from the array:

```
valueArray.read(i, &this[_result % 5].sumCallback());
```

If `this` is a distributed module, then the `sumCallback` function invoked with the result will be that of module instance `_result % 5`.

7 Distributions

Distributions are a major construct of parallelism in Elan. In the simplest case, distributions can be used to create arrays, and in more complicated cases, distributions can be used to map many complex keys (such as structs) to specific modules to facilitate the sharing of work or the partitioning of data.

When a distribution is applied to a module via the `maps` keyword, a number of copies of the module are created upon instantiation of the module, which we will refer to as the module instances. Particular module instances are accessed via a key, which is passed as an argument to the bracket operator on a distributed module, such as in `m[key].distFunc()`. A function may be invoked on all module instances represented by a distributed module by using standard function invocation, as with `m.distFunc()`.

The map function that determines the key-module instance relationship is specified by the distribution applied to a module. It can be an bijective map, in which case there are equally many keys and module instances, or a surjective (but not injective) map, in which case there are more keys than module instances. This latter case is the most common, as it can be used to dole out portions of a large data set to multiple module instances. The key used for a distribution can be any primitive type, such as integers, floating point types, and structs, although integers are easiest to reason about. The value returned by the map function is a partition number, which is the index of the module index to which the call for a key will be routed.

The most useful feature of distributions from within a module instance is the distribution's key iterator. Each module instance is able to iterate through the keys particular to its index within the list of distributed module instances, i.e. those keys that map to it. This is a powerful tool for initiating distributed computations that take a key set as input, as we will show in an example using accumulators later in this section.

When a distribution is applied to an array via the `maps` keyword, the array is partitioned so that separate memory ports access separate portions of the array. This is useful for creating a one-to-one correspondence between module instances and array partitions when the same distribution is applied to both a module and an array. If the keys used in the distribution are array indexes, then each module instance has ownership of a certain set of elements in the array as determined by the map, which can be used to enforce exclusivity or to increase performance by having each module instance use its port to the array to perform reads and writes on those indexes.

Definitions of distributions take the following form:

```
distribution identifier<key = keytype> {
  inline keytype firstKey(unsigned int:32 partition) {
    // Return the first key in the key space of
```

```
    // the distribution for the given partition.
  }

  inline keytype lastKey(unsigned int:32 partition) {
    // Return the last key in the key space of
    // the distribution for the given partition.
  }

  inline keytype nextKey(keytype k) {
    // Return the key after k in the key space of
    // the distribution.
  }

  inline unsigned int:32 numPartitions() {
    // Return the number of partitions associated
    // with the distribution.
  }

  inline unsigned int:32 numKeys() {
    // Return the number of keys associated with
    // the distribution.
  }

  inline unsigned int:32 map(keytype k) {
    // Map the key to a partition.
  }

  inline unsigned int:32 mapIndex(unsigned int:32 index) {
    // Map an index that is at least 0 and less
    // than the number of keys to a partition.
  }
}
```

`keytype` is the type of the key used in the distribution map, such as a struct, an integer, etc. The `map` function maps keys of this type to the index of a partition that is at least `0` and is less than `numPartitions()`. The `mapIndex` function maps an index that is at least `0` and less than `numKeys()` to a partition–this is used by arrays to relate indexes to partitions in correspondence with the key set. For distributions intended to be applied only to arrays, defining `numKeys` and `mapIndex` is optional. For distributions intended to be applied only to modules, defining `firstKey`, `lastKey`, and `map` is optional.

Regarding the other defined functions, `firstKey(partition)` returns the first key in the key space of the distribution for the given partition and `lastKey(partition)` returns the last key for the given partition. `nextKey(k)` returns the key following the given key `k`, although its behavior is undefined for keys `k` that cannot be expressed as a combination of `firstKey` and `nextKey`. If the keys were unsigned 32-bit integers, the key space ranged from 0 to 99, and the distribution was a striped distribution with `numPartition` partitions, for example, `firstKey(partition)` would return `partition`, `lastKey(partition)` would return `100 - numPartitions + partition`, and `nextKey(k)` would return `k + numPartitions`. In this distribution, `map(k)` would return `k % numPartitions`. The `firstKey`, `lastKey`, and `nextKey` functions must have the following properties:

1. The following function terminates for any partition with index `partition` that is at least `0` and less than `numPartitions()`:

```
async enumerate(unsigned int:32 partition) {
  key k = firstKey(partition);
  while (k != lastKey(partition)) {
    k = nextKey(k);
  }
}
```

2. For each partition with index `partition`, define `L(partition)` as the enumeration of all keys between `firstKey(partition)` and `lastKey(partition)`. For any partition index `partition2` with `partition2 != partition`, the intersection of `L(partition)` and `L(partition2)` must be empty.

There are two built-in distributions in the Elan language, `BlockDistribution` and `StripeDistribution`. `BlockDistribution` maps approximately equal-sized blocks of keys to partitions, while `StripeDistribution` maps keys to partitions in a striped, or interleaved, pattern. Use of either requires a numeric type as the key, such as an integer. Appendices A and B contain the source code for these two types of distributions.

Distributions are like modules in that they may only be constructed during compilation or initialization, though they have the additional restriction that they may only possess a constructor and inline functions–no asynchronous or asynchronous functions–and they may not modify their internal state after invocation of the constructor, which is a side effect of the restriction on functions they may define. The number of partitions (as given by `numPartitions()`) must be known at compile-time, so in the block distribution built into the language (see Appendix A), for example, the argument passed to the constructor for the number of partitions must be of phase type `compile`.

7.1 Array distributions

Arrays can have distributions applied to them to associate certain indexes with certain memory ports, which are the partitions of the distribution. When a distribution is applied to an array, the `mapIndex` function is used to map array indexes to partitions, and the `numKeys` function is used to determine the length of the array. The usage of distributions with arrays is:

```
Array<type> identifier maps distribution-identifier;
```

When a distribution is applied to an array, `construct` is invoked implicitly with the size specified by the distribution. All calls to functions using the array identifier will use the distribution map to determine the port to which to route the access; that is, `array.read(index, func)` is translated to use `dist.mapIndex(index)` at the memory interface.

7.2 Module distributions

Distributions can be used to create multiple instances of modules and relate a section of the key set to each one, as discussed above. When `maps dist` is applied to a module definition, where `dist` is the name of a distribution variable, `dist.numPartitions()` copies of the module are created upon a declaration of it each with a separate index number. Each distributed instance of a module is able to query the distribution for its local index, via `dist.getIndex()`, and the set of keys that correspond to it, via `dist.firstKey(getIndex())`, `dist.nextKey(key)`, and `dist.lastKey(getIndex())`. One could create and use an instance of BlockDistribution making use of these functions as follows:

```
BlockDistribution<unsigned int:32> dist;
Array<unsigned int:32> valueArray maps dist;
// Create a location in memory to store the
// total sum.
Array<unsigned int:32> totalSum(1);
boolean done;

module Accumulator maps dist {
  unsigned int:32 sum, valuesToAdd;
  Result result;
  Accumulator(Result result) {
```

```
12      this.result = result;
13    }
14
15    async computeSum() {
16      sum = 0;
17      // Both BlockDistribution and StripeDistribution define
18      // a numKeysForPartition function.
19      valuesToAdd = dist.numKeysForPartition(getIndex());
20      for (unsigned int:32 i = dist.firstKey(getIndex());
21           i <= dist.lastKey(getIndex()); i = dist.nextKey(i)) {
22        valueArray.read(i, &sumCallback());
23      }
24    }
25
26    async sumCallback(unsigned int:32 value) {
27      sum += value;
28      valuesToAdd--;
29      if (valuesToAdd == 0) {
30        // Report the total sum to a collector of
31        // the results.
32        result.reportSum(sum);
33      }
34    }
35  }
36
37  module Result {
38    unsigned int:32 totalSum, sumsToReport;
39
40    // Initialize the result module with an expected
41    // number of sums to be reported. Alternatively,
42    // this could have been in the constructor, but
43    // making this a method allows for re-use of this
44    // module in future computations.
45    sync init(unsigned int:32 sumsToReport) {
46      totalSum = 0;
47      this.sumsToReport = sumsToReport;
48    }
49
50    async reportSum(unsigned int:32 sum) {
51      totalSum += sum;
52      sumsToReport--;
53      if (sumsToReport == 0) {
54        totalSum.writeNotify(0, totalSum, &totalWritten());
55      }
56    }
57
58    async totalWritten() {
59      // Indicate that the total has been written to
60      // memory.
61      done = true;
62    }
63  }
64
65  // Create a block distribution with 5 partitions
66  // and 1000 indexes.
67  dist.construct(5, 1000);
68
69  sync init() {
70    // valueArray is initialized during the
71    // initialization phase.
72  }
73
74  sync main() {
75    done = false;
76
```

```
77    // Create 5 accumulators, one per partition
78    // using the distribution applied to Accumulator.
79    Accumulator distributedAccum;
80    Result result;
81
82    // Invokes the constructor of all 5 modules.
83    // It does not matter if result has been
84    // constructed when this constructor is
85    // invoked since result was already instantiated.
86    distributedAccum.construct(result);
87    result.construct();
88
89    // Notify the result module of how many sums
90    // to expect.
91    result.init(dist.numPartitions());
92
93    // Invoke computeSum on all of the accumulator
94    // modules.
95    distributedAccum.computeSum();
96
97    // Alternatively, instead of using a multicast
98    // call like the above, one can access a particular
99    // distributed module via a key (not index!) that
100   // corresponds to it; e.g.:
101   // for (unsigned int:32 i = 0; i < dist.numPartitions(); i++) {
102   //   distributedAccum[dist.firstKey(i)].computeSum();
103   // }
104 }
```

In the above example, a distribution with 5 partitions and 1000 keys is applied to both an array and a module. This has the effect of creating 5 ports to the array, with a one-to-one correspondence between ports and distributed instances of the Accumulator modules, of which there are also 5. Each accumulator computes the sum of the values in its range of the array by iterating through the index set associated with it and summing the values. Once an accumulator finishes, it sends its sum to the result module, which stores the sum in memory.

7.3 Dynamic mappings

Elan provides a mechanism for the "dynamic" allocation of modules that creates and manages multiple copies of modules. Dynamic modules, which are modules that can be allocated dynamically, generally have a well-scoped lifespan and perform a specific task, such as computing a value or acting as a semi-persistent container for data. To make a module dynamic, one must create a `dynamic` mapping, which must be instantiated and used with the `maps` keyword on a module to make it dynamic. `dynamic` mappings take the following form:

```
dynamic identifier<key = keytype> {
  inline unsigned int:32 hash(keytype k) {
    // Hash the given key to an unsigned int:32.
  }

  inline unsigned int:32 numInstances() {
    // Return the number of instances that can be
    // allocated by the dynamic mapping.
  }
}
```

In comparison to distributions, dynamic mappings provide no functionality for iterating over key sets like distributions do, and furthermore they only require the ability to hash keys, whereas distributions required the ability to map a key to a particular partition. The `hash` function defined by dynamic mappings should

produce as few collisions as possible for the key set, since otherwise lookup and allocation of modules for certain keys will take a disproportionately long number of cycles.

Instances of a module to which a dynamic mapping have been applied are allocated automatically whenever their key is used. When a call is made to a dynamic module with a key for which no module instance has been allocated, an instance of the module is removed from the ready queue and a reference to it stored in the allocated modules hash table along with its key. Upon receiving a call to the dynamic module using the same key, the dynamic module will perform a lookup into the allocated modules hash table and direct the call to the appropriate module instance. If a call is made for a key to which no module instance has been allocated and there are no more available module instances in the ready queue, the caller will block until one is available.

Dynamic module instances may deallocate themselves through the `deallocate` function inherent to dynamic modules, which takes no arguments and simply puts the module instance back on the ready queue. After deallocation, a call to the dynamic module using the key with which it was previously associated will allocate a new instance. The following is an example of creating and using a dynamic mapping.

```
struct InputStruct {
  int:32 value1;
  boolean value2;
}

dynamic SimpleMapping<key = InputStruct> {
  unsigned int:32 numInstances;
  SimpleMapping(unsigned int:32 numInstances) {
    this.numInstances = numInstances;
  }

  unsigned int:32 map(InputStruct k) {
    return (k.value1 << 1) | (k.value2 ? 1 : 0);
  }

  unsigned int:32 numInstances() {
    return numInstances;
  }
}

SimpleMapping simpleMapping;

module LengthyComputation maps simpleMapping {
  async computeResult(InputStruct inputData) {
    // ... perform some long computation.
    reportResult(result);
  }
}

compile unsigned int:32 numInstances = 15;
simpleMapping.construct(numInstances);
LengthyComputation dynamicComputation;

// This call to construct sets up 15 instances of the module.
dynamicComputation.construct();

sync main() {
  InputData inputData;
  for (unsigned int:32 i = 0; i < 1000; i++) {
    // Assign values to inputData's fields before
    // making the call.
    inputData.value1 = ...;
    inputData.value2 = ...;
    dynamicComputation[inputData].computeResult(inputData);
  }
```

```
46 }
```

In this example, 1000 invocations of the computation are executed in total, with up to 15 executing simultaneously. While the same effect could be had using a distribution with 15 instances, it would require some effort to ensure that each key mapped to a module instance that was unlikely to be in use. Another difference between this dynamic mapping approach and one involving a distribution is that calls to `dynamicComputation` block automatically in the above example when there are no free module instances to allocate, whereas using distributions, each call would be delivered to a module instance immediately, which will block if the called function is synchronous and will not block if the called function is asynchronous.

8 Breadth-first search

Here we develop a full example of an application–parallel breadth-first search–using the Elan language. The breadth-first search algorithm is split into multiple phases, where in each phase, the algorithm processes all nodes in the current queue by enumerating their neighbors. For each neighbor that has not yet been visited, the algorithm marks the neighbor as visited and places it on the queue for the next phase.

In parallel breadth-first search, multiple instances of this algorithm execute in parallel on a common set of nodes that have been divided up under some scheme. There is a race condition on the update to the visited status of nodes, as multiple agents might check the visited status of a node at the same time and enqueue it more than once, but there is no harm in this aside from producing extra (duplicate) work to be processed in the next phase.

8.1 Division of work

In breadth-first search, we will separate the work into multiple distinct stages. The purpose of stages is to pipeline array accesses, as these are the most costly operations performance-wise and in general the more that can be issued simultaneously the better performance the search algorithm will have.

The first stage is reading the nodes from the queue of nodes to be explored–in the very first phase of the algorithm, this queue will most likely contain just a single node index, whereas in later phases it will contain many, and finally zero nodes indexes at the end. Data produced by this stage is a sequence of integers corresponding to node indexes, which is fed as input to the next stage.

The second stage is looking up the adjacency list of each node index given as input. The output of this stage is indexes of adjacent nodes; there will be $\sum_{i=0}^{n-1} m_i$ such indexes produced by this stage for n nodes that have m_i neighbors, where $i \in \{0..n-1\}$.

The third stage is checking whether each node given as input has been visited in a previous phase of breadth-first search. For each node index, this stage looks up the visited status of the node. As output, this stage produces node index-visited status pairs, where the visited status is an integer indicating the search phase in which the node was visited (-1 for yet-unvisited).

The fourth stage is enqueueing node indexes given as input that have not been visited in a previous phase. This stage produces no output, but it writes all indexes of nodes to be explored to a specific queue array stored in DRAM. Once the fourth stage has completed, the first stage recommences with an incremented phase number. The algorithm terminates when the queue of nodes to explore is discovered to be empty in the first stage.

8.2 Parallelization

To parallelize these stages, we will apply a distribution using the nodes' index as the key, and we will create multiple ports to the arrays in which data is stored using the same distribution. Each partition will "own" a set of nodes as decided by the distribution, and these are the only nodes on which it will perform operations. The termination condition for the algorithm with multiple partitions is that every partition's queue is discovered to be empty in the first stage; i.e. the entire graph has been explored. The termination condition could also be set to something like reaching a specific node index or a specific set of node indexes without much extra work.

The goal of parallelization is to alleviate performance bottlenecks by maximizing throughput of constrained resources. In the case of breadth-first search, the bottleneck is memory accesses, and the more reads and writes than can be issued for a period of time the better. The distribution applied in the Elan implementation, to be effective, should allow many module instances to process work in parallel while issuing a constant stream of array accesses in order to saturate the memory interface.

8.3 Java pseudocode

Here we present Java code for sequential breadth-first search using the division of work described above, which is what we will use as a basis for the Elan implementation. We restrict ourselves to using only arrays as opposed to resizable data structures to make the implementation more in line with the Elan version.

```
1  public class BFS {
2    private static final int QUEUE_CAPACITY = 10000;
3    private final int numNodes;
4    private int[][] adjacencyListArray;
5    private int[] visitedStatusArray;
6
7    private int currentPhase;
8    private int currentQueueSize, nextQueueSize;
9    private int[] currentQueue, nextQueue;
10
11   public BFS(int numNodes) {
12     this.numNodes = numNodes;
13     adjacencyListArray = new int[numNodes][4];
14     visitedStatusArray = new int[numNodes];
15     currentQueue = new int[QUEUE_CAPACITY];
16     nextQueue = new int[QUEUE_CAPACITY];
17   }
18
19   public void init() {
20     // Initialize the adjacency lists with some random values.
21     for (int i = 0; i < numNodes; i++) {
22       int listLength = (int) (Math.random() * adjacencyListArray[i].length);
23       for (int j = 0; j < adjacencyListArray[i].length; ++j) {
24         adjacencyListArray[i][j] = (int) (Math.random() * numNodes);
25       }
26       adjacencyListArray[i][listLength] = -1;
27     }
28
29     // Set the visited status of every node to unvisited.
30     for (int i = 0; i < numNodes; i++) {
31       visitedStatusArray[i] = -1;
32     }
33
34     currentQueueSize = 0;
35     currentPhase = 0;
36
```

```
37      // Enqueue the first node.
38      visitedStatusArray[0] = 0;
39      nextQueue[0] = 0;
40      nextQueueSize = 1;
41    }
42
43    public void startSearch() {
44      while (nextQueueSize != 0) {
45        int[] tempQueue = currentQueue;
46        currentQueue = nextQueue;
47        nextQueue = tempQueue;
48        currentQueueSize = nextQueueSize;
49        nextQueueSize = 0;
50        processQueue();
51        currentPhase++;
52      }
53    }
54
55    private void processQueue() {
56      for (int i = 0; i < currentQueueSize; i++) {
57        readNeighbors(currentQueue[i]);
58      }
59    }
60
61    private void readNeighbors(int nodeIndex) {
62      checkVisited(adjacencyListArray[nodeIndex]);
63    }
64
65    private void checkVisited(int[] adjacencyList) {
66      for (int i = 0; adjacencyList[i] != -1; i++) {
67        enqueueUnvisited(adjacencyList[i], visitedStatusArray[adjacencyList[i]]);
68      }
69    }
70
71    private void enqueueUnvisited(int nodeIndex, int visitedPhase) {
72      if (visitedPhase == -1) {
73        visitedStatusArray[nodeIndex] = currentPhase;
74        nextQueue[nextQueueSize++] = nodeIndex;
75      }
76    }
77
78    public static void main(String[] args) {
79      BFS bfs = new BFS(100000);
80      bfs.init();
81      bfs.startSearch();
82    }
83  }
```

In this example, the current queue and next queue are required to be quite large in relation to the number of nodes (one-tenth the size), which is clearly an issue for a hardware implementation–the queues will need to be kept in block RAM or DRAM, since clearly they are far too large to fit in local registers. Splitting up the queues between multiple units of computation, as we will do in the Elan implementation of breadth-first search, will help to reduce the size required per module for these arrays.

One key difference between this implementation and an Elan version are the function calls. In the above code, when a method is invoked, the caller blocks until the method returns, which makes synchronization of phases much easier. In the Elan implementation, we will rely on the `barrier` function (see Section 5.4) to accomplish a similar effect.

8.4 Elan Implementation

We will first define a manager module (of which there is only one) that will be responsible for initiating new phases of the search and deciding when the search has finished.

```
module BFSManager {
  unsigned int:32 numPartitions, reportedQueueEmpty;
  unsigned int:32 currentPhase;
  BFSModule bfs;
  boolean done = true;

  BFSManager(unsigned int:32 numPartitions, BFSModule bfs) {
    this.numPartitions = numPartitions;
    this.bfs = bfs;
    reportedQueueEmpty = 0;
    currentPhase = 0;
    done = false;
  }

  async startSearch() {
    while (!done) {
      reportedQueueEmpty = 0;
      // This will call processQueue on all BFS modules
      // in the distribution and will wait until all call
      // invocations performed on behalf of processQueue
      // for each partition have completed.
      bfs.processQueue(currentPhase).barrier();
      level++;
    }
  }

  async reportQueueEmpty(unsigned int:32 index) {
    reportedQueueEmpty++;
    // The algorithm is done once all queues are empty
    // across partitions.
    if (reportedQueueEmpty == numPartitions) {
      done = true;
    }
  }
}
```

Here `BFSModule` is the module we have yet to define that encapsulates all of the BFS phase operations. The two functions within this manager class that are used by BFS modules are `reportQueueEmpty` and `reportDone`, which signal to the manager that the BFS module with the given index is done with its work for this level either because its queue is empty or it has finished all its work.

As setup for the application, we will declare a block distribution and some constants related to the number of partitions, nodes, and capacities.

```
// The number of partitions to use in the search;
// the higher this number, the greater the degree of
// parallelism that will be used.
compile unsigned int:32 numPartitions = 10;
// The total number of nodes whose indexes are
// represented in the search, though not every node
// index has to be part of the graph.
config unsigned int:32 numNodes;
// The number of node indexes that each module
// is able to store in its current and next queues.
config unsigned int:32 queueCapacity;

// The block distribution that will be applied to the
```

```
// arrays and modules.
BlockDistribution<unsigned int:32> blockDist;

// Store up to 100 nodes in the adjacency list for
// each node--this is the connectivity of the graph.
// -1 is designated as terminating the list of nodes.
config unsigned int:32 adjacencyListCapacity = 100;
Array<int:32[adjacencyListCapacity]>
  adjacencyListArray maps blockDist;
Array<int:32> visitedStatusArray maps blockDist;

sync init() {
  // Assign a value to numNodes.
  // Construct adjacencyListArray and visitedStatusArray.
  // Set up the adjacency list array with the neighbors of each
  // node, and set each index in the visited status array to -1
  // to indicate that the node at that index has not yet been
  // visited. Set the visited phase of node 0 to 0, as this will
  // be the first node to be explored.

  // Set up the block distribution.
  blockDist.construct(numPartitions, numNodes);
}

sync main() {
  // Construct the distributed BFS modules and the BFS manager,
  // which will oversee the search.
  BFSModule bfs;
  BFSManager manager;
  bfs.construct(manager);
  manager.construct(bfs);

  // Commence the search.
  manager.startSearch();
}
```

Finally, we will define the BFS module itself. Within the `BFSModule` module, we will define four functions, one for each stage, named `processQueue`, `readNeighbors`, `checkVisited`, and `enqueueUnvisited`, in correspondence with the four stages outlined above in Section 8.1.

```
module BFSModule maps blockDist {
  BFSManager manager;
  unsigned int:32 phase;
  unsigned int:32 currentQueueSize, nextQueueSize;
  // These local arrays will be stored in block RAM.
  unsigned int:32 currentQueue[queueCapacity];
  unsigned int:32 nextQueue[queueCapacity];

  BFSModule(BFSManager manager) {
    this.manager = manager;
    phase = 0;
    currentQueueSize = 0;
    nextQueueSize = 0;
    if (getIndex() == 0) {
      // Enqueue node 0 in the first partition. The search
      // will start with this node.
      nextQueue[0] = 0;
      nextQueueSize = 1;
    }
  }

  async processQueue(unsigned int:32 phase) {
    this.phase = phase;
    currentQueueSize = nextQueueSize;
```

```
25      nextQueueSize = 0;
26      // Swap the current and next queue.
27      unsigned int:32 tempQueue[] = nextQueue;
28      nextQueue = currentQueue;
29      currentQueue = tempQueue;
30
31      if (currentQueueSize == 0) {
32        manager.reportQueueEmpty(getIndex());
33      } else {
34        for (unsigned int:32 i = 0; i < queueSize; i++) {
35          unsigned int:32 nodeIndex = currentQueue[i];
36          this[nodeIndex].readNeighbors(nodeIndex);
37        }
38      }
39    }
40
41    async readNeighbors(unsigned int:32 nodeIndex) {
42      // Use the result of the read from the array to determine
43      // the distributed module instance on which to invoke
44      // the checkVisited function.
45      adjacencyListArray.read(nodeIndex, &this[_result].checkVisited());
46    }
47
48    async checkVisited(unsigned int:32 nodeIndex[adjacencyListCapacity]) {
49      for (unsigned int:32 i = 0; nodeIndex[i] != -1; i++) {
50        visitedStatusArray.read(nodeIndex[i],
51          &this[nodeIndex[i]].enqueueUnvisited(nodeIndex[i]));
52      }
53    }
54
55    async enqueueUnvisited(unsigned int:32 nodeIndex,
56                           int:32 visitedStatus) {
57      if (visitedStatus == -1) {
58        visitedStatusArray.write(nodeIndex, phase);
59        nextQueue[nextQueueSize++] = nodeIndex;
60      } // else ignore this node.
61    }
62  }
```

8.5 Analysis

This code corresponds almost exactly to the Java pseudocode for breadth-first search given in Section 8.3. Key differences include the distribution applied, which is responsible for parallelizing the algorithm, pipelined accesses to arrays, and the use of `barrier` to synchronize between phases of the search.

The distribution type and configuration can have a dramatic effect on the efficiency of the above implementation, since with a block distribution, if the data set used has nodes that are likely to be connected to node indexes close to their own, the workload over the course of the search will be highly unbalanced. Module instances that have lower partition numbers will end up doing most of the work in the beginning and those with higher partition numbers will end up doing most of the work toward the end. This could be addressed with relative ease by using a stripe distribution instead, although similarly there are cases where stripe distributions would end up doling out an unbalanced workload as well.

Synchronization using a barrier in the Elan implementation of breadth-first search is unnecessarily expensive, as the careful reader might notice. Within the four stages of node processing in `BFSModule`, only the first and final stages modify any state. It is actually possible to overlap execution of search phases in order to achieve an even higher level of throughput, for which fences (see note in Section 5.4) would be useful. We will not, however, explore the details of such modifications to the search code since the behavior of fences is

not yet well-specified.

9 Comparison of Elan versus Chapel

Chapel, which is a parallel programming language developed by Cray, Inc. for the simplified creation of distributed applications on multicore systems, had a strong influence on the design of the Elan language. In particular, Chapel enables developers to maintain a clean separation between their algorithms and the platforms on which they are designed to run and makes inter-node communication implicit, as opposed to the commonly-used MPI, which requires communication to be specified explicitly. One of the most powerful ways in which Chapel separates platform characteristics from algorithm implementation is in its use of distributions as applied to key sets (domain spaces), which can be used to map keys to nodes (locales) through various schemes. In Elan, this idea manifests as distributions, which can be applied to modules and arrays in order to partition them in accordance with a data set. Changing the level of parallelism or the partitioning of a data set across modules is as simple as changing configuration or the type of the distribution used.

There are multiple significant differences between the requirements of Elan and Chapel that necessitated the creation of a new language specifically for FPGAs. Dynamic allocation (aside from pseudo-dynamic allocation of module instances using a dynamic mapping) is not feasible on FPGAs, so in Elan we permit only compile-time instantiation of modules in contrast to Chapel, which relies on dynamic allocation. Chapel does not expose an explicit memory interface, and in absence of a compiler that can determine the most efficient way in which to pipeline and parallelize memory accesses, we implemented an array type in Elan with the key functionality required to facilitate memory-intensive FPGA applications. Functions are also notably different in semantics between Chapel than in Elan due to the inability to create new instances of functions dynamically in hardware; multiple calls to the same (asynchronous) function are permitted in Elan as backed by queues.

While we considered modifying the Chapel compiler to address these differences while capitalizing on the existing language base, we eventually concluded that the differences were significant enough to merit the creation of a new language, Elan, using many of the ideas advocated by Chapel yet also implementing those ideas specific to hardware constraints and performance.

10 Origins of the Elan language

The Elan language is the result of multiple years of experimentation with the high-level design of applications for FPGAs. We originally devised distributed and dynamic objects and asynchronous functions as the means for creating parallelism in FPGA-based applications, though the syntax of these changed greatly over time. We developed a sandbox for experimentation in Java that provided the key functionality of the language for describing distributed applications that we sought, which uses Java's thread library, blocking queues, and a variety of adapter classes to provide the language framework. After writing parallel applications such as sparse-matrix multiplication, breadth-first search, and short-read genome sequencing within this framework, we implemented them in hardware using the same semantics of the then-abstract language.

Having developed and ported multiple applications to hardware in this manner, we also began to experiment with Chapel as a potential substitute for creating an entirely new language. We found that the tenets of Chapel were in nearly complete agreement with those of our own language, and we implemented some of the same applications in Chapel in order to facilitate a three-way comparison between the Chapel code, the emulated Java sandbox code, and the generated hardware that used the language semantics. We found that while Chapel made for an extremely convenient environment in which to develop parallel applications, it did not and could not expose the constructs required to emulate how applications run on FPGAs in our model.

It was at this point that we began to formalize a language for parallel applications on FPGAs, which we named Elan.

11 Future work

Though we have formalized much of the Elan language and how it will be implemented, we do not have an Elan-to-hardware compiler. The creation of such a compiler is the logical next step in this project, and we plan to use a tool such as Auto-ESL to facilitate the conversion of C code generated by the Elan compiler to Verilog. We have written applications in "pure" Elan code, in Java code using the Java sandbox emulator, and in hardware for the Pico and Convey platforms, so we have multiple examples of the language itself and how it should execute in hardware. We also have some preliminary results for the performance of hardware applications written using the functionality of the Elan language; in parallel breadth-first search on the Convey platform, we achieved 70% of the throughput compared to Convey's custom-designed version, and in short-read genome sequencing on the Convey platform, we achieved approximately a 100-times speedup for a single FPGA versus a dual-processor Intel Nehalem system with 16 cores in total using the same algorithm, and approximately a 350-times speedup for four FPGAs.

12 Acknowledgments

I would like to thank Professor Carl Ebeling for encouraging me to join the Elan project and for overseeing my research efforts related to it. He and Professor Scott Hauck, of the Department of Electrical Engineering at the University of Washington, cemented much of the language in part through discussion with me, and their ideas and suggestions appear throughout the above work. Dustin Richmond, Joshua Green, Stephen (Ross) Thompson, and Sean Cowan, who are undergraduate students at the University of Washington in the CSE and electrical engineering departments, have done a significant amount of work on the hardware implementation of Elan and its memory model by creating or porting applications using an Elan-like framework, and their findings and results influenced the direction of the language.

Appendices

Appendix A BlockDistribution

A block distribution maps blocks of keys to partitions. For example, a block distribution using the integers 0 through 7 as a key set with 4 partitions would map 0 and 1 to partition 0, 2 and 3 to partition 1, and so forth. The following `BlockDistribution` is built into the Elan language and supports numeric types as keys. It accounts for uneven allocation of keys where the number of partitions does not evenly divide the number of keys.

```
distribution BlockDistribution<key> {
  unsigned int:32 numPartitions, numIndexes;
  key keyStart, keyEnd;
  BlockDistribution(unsigned int:32 numPartitions,
                    unsigned int:32 numIndexes) {
    this.numPartitions = numPartitions;
    this.numIndexes = numIndexes;
    keyStart = 0;
    keyEnd = numIndexes - 1;
  }

  BlockDistribution(unsigned int:32 numPartitions,
                    key keyStart,
                    key keyEnd) {
    this.numPartitions = numPartitions;
    numIndexes = keyStart - keyEnd;
    this.keyStart = keyStart;
    this.keyEnd = keyEnd;
  }

  inline unsigned int:32 numKeysForPartition(
      unsigned int:32 partition) {
    return numKeys / numPartitions +
      (numKeys % numPartitions >= 1 ? 1 : 0);
  }

  inline key firstKey(unsigned int:32 partition) {
    // This computation accounts for off-by-one errors when
    // numIndexes is not divided evenly by numPartitions.
    // The first numLarge partitions will have one more
    // key attributed to them than the other partitions.
    unsigned int:32 numLarge = numIndexes % numPartitions;
    unsigned int:32 smallSize = numIndexes / numPartitions;
    unsigned int:32 largeSize = smallSize + 1;
    if (partition < numLarge) {
      return partition * largeSize;
    } else {
      return numLarge * largeSize +
        (partition - numLarge) * smallSize;
    }
  }

  inline key lastKey(unsigned int:32 partition) {
    return firstKey(partition) + numKeysForPartition(partition);
  }

  inline key nextKey(key k) {
    return k + 1;
  }

  inline unsigned int:32 numPartitions() {
```

```
52      return numPartitions;
53    }
54
55    inline unsigned int:32 numKeys() {
56      return numIndexes;
57    }
58
59    inline unsigned int:32 map(key k) {
60      return k / numPartitions;
61    }
62
63    inline unsigned int:32 mapIndex(unsigned int:32 index) {
64      return k / numPartitions;
65    }
66  }
```

Appendix B StripeDistribution

A stripe distribution maps keys in a round-robin fashion to partitions. For example, a stripe distribution using the integers 0 through 7 as a key set with 4 partitions would map 0 and 4 to partition 0, 1 and 5 to partition 1, and so forth. The following `StripeDistribution` is built into the Elan language and supports numeric types as keys. It accounts for uneven allocation of keys where the number of partitions does not evenly divide the number of keys.

```
1   distribution StripeDistribution<key> {
2     unsigned int:32 numPartitions, numIndexes;
3     key keyStart, keyEnd;
4     BlockDistribution(unsigned int:32 numPartitions,
5                       unsigned int:32 numIndexes) {
6       this.numPartitions = numPartitions;
7       this.numIndexes = numIndexes;
8       keyStart = 0;
9       keyEnd = numIndexes - 1;
10    }
11
12    StripeDistribution(unsigned int:32 numPartitions,
13                       key keyStart,
14                       key keyEnd) {
15      this.numPartitions = numPartitions;
16      numIndexes = keyStart - keyEnd;
17      this.keyStart = keyStart;
18      this.keyEnd = keyEnd;
19    }
20
21    inline unsigned int:32 numKeysForPartition(
22        unsigned int:32 partition) {
23      return numKeys / numPartitions +
24        (numKeys % numPartitions >= 1 ? 1 : 0);
25    }
26
27    inline key firstKey(unsigned int:32 partition) {
28      return keyStart + partition;
29    }
30
31    inline key lastKey(unsigned int:32 partition) {
32      return firstKey(partition) + partition +
33        numPartitions * (numKeysForPartition(partition) - 1);
34    }
35
36    inline key nextKey(key k) {
```

```
37      return k + numPartitions;
38    }
39
40    inline unsigned int:32 numPartitions() {
41      return numPartitions;
42    }
43
44    inline unsigned int:32 numKeys() {
45      return numIndexes;
46    }
47
48    inline unsigned int:32 map(key k) {
49      return k % numPartitions;
50    }
51
52    inline unsigned int:32 mapIndex(unsigned int:32 index) {
53      return k % numPartitions;
54    }
55  }
```

Appendix C Reserved words

The following is a list of reserved words in Elan. Those related to logic and control flow have the same syntax and semantics as in C, C++, and Java.

`compile`, `config`, `unsigned`, `int`, `boolean`, `float`, `double`, `struct`, `module`, `Array`, `_result`, `barrier`, `distribution`, `dynamic`, `void`, `construct`, `sync`, `async`, `return`, `for`, `while`, `do`, `switch`, `case`, `default`, `break`, `continue`, `if`, `else`, and `maps`.

Appendix D Operators

The following is a list of operators in Elan.

`.` (field or function accessor), `[]` (local array operator), `<>` (Array primitive type), `()` (method argument list), {} (struct, module, function, or block scope specifier), `=` (assign), `+=` (increment by value), `-=` (decrement by value), `*=` (multiply by value), `/=` (divide by value), `++` (prefix and postfix increment), `--` (prefix and postfix decrement), `?` and `:` (ternary operators), `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus), `!` (logical not), `==` (reference or value equality), `!=` (logical reference or value inequality), `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), `&&` (logical and), `||` (logical or), `&` (bitwise and, function reference), `|` (bitwise or), `^` (bitwise xor), `>>` (signed bit shift right), `>>>` (unsigned bit shift right), and `<<` (bit shift left).