A Data-Intensive Programming Model for FPGAs: A Genomics Case Study

Elliott Brossard, Dustin Richmond, Joshua Green, Carl Ebeling, Larry Ruzzo
Department of Computer Science and Engineering
University of Washington, Seattle, WA, USA
{snowden, watertav, greenj5, ebeling, ruzzo}@cs.washington.edu

Corey Olson, Scott Hauck
Department of Electrical Engineering
University of Washington, Seattle, WA, USA
coreybolson@gmail.com, hauck@uw.edu

***Abstract*—Genomics computing is indispensable in basic medical research as well as in practical applications such as disease prevention, pharmaceutical development, and criminal forensics. DNA sequencing, assembly and analysis are key components of genomics computing. Coupled with the increased use of computation for both synthesis and analysis of data in genomics is the astounding increase in the rate at which next-generation sequencing platforms are producing genomic data. Keeping up with the combination of increasing levels of algorithmic demands and an exponential increase in data represents a huge computational challenge that requires a corresponding revolution in how we process the data.**

Field Programmable Gate Arrays (FPGAs) are particularly well suited to the type of highly parallel, bit-level computations found in genomics algorithms. Unfortunately, the use of FPGA platforms among genomics researchers has been limited by the specialized hardware design expertise currently required to use these platforms. Another limiting factor has been a proliferation of FPGA platform architectures, each generally requiring a re-implementation of the algorithm.

This paper describes a new programming model called Elan and an associated compiler that we are developing for FPGA-based genomic applications. The Elan model and compiler allow a programmer to use familiar concepts from parallel and distributed computing to develop an application at a relatively high level of abstraction, which can then be compiled automatically to large-scale FPGA platforms. One of the goals of Elan is to allow an application to be run seamlessly across both the CPU and FPGA portions of a platform, and to be parallelized easily across a system comprising many FPGAs and CPUs. We use the short read alignment application as the motivating example.

Keywords-FPGAs; configurable computing; bioinformatics

I. INTRODUCTION

Next generation sequencing systems—devices that can rapidly read DNA sequences—are going through an exponential increase in throughput and decrease in processing costs that will radically change most biology-related fields. Genomics computing, which covers the computational problems associated with processing these flows, is indispensable in basic biological research as well in for practical applications such as disease prevention, pharmaceutical development, and criminal forensics. Genomics computing also plays a critical role in agriculture, enabling faster development of high-yield disease-resistant crops. DNA sequencing, assembly and analysis are key components of these flows. In recent years, there have been a number of academic open source software packages written that attempt to accelerate genomics sequence alignment and assembly. These packages can dramatically speed the analysis of genetic information, but they require large amounts of computing power in the form of CPU clusters or cloud computing platforms.

Keeping up with the combination of increasingly sophisticated algorithmic demands and exponentially increasing data represents a huge computational challenge. Currently the only answer is ever-larger CPU clusters, which are an expensive and power-hungry solution to a fine-grained computation problem. One alternative is GP-GPUs, but their SIMD operation and small on-chip storage capacities make them poorly suited to most genomics computations.

Reconfigurable computing, which uses Field-Programmable Gate Arrays (FPGAs) to accelerate important computing applications, has demonstrated huge performance increases and power consumption savings for computationally challenging tasks in signal processing, medical diagnostic imaging, networking, cryptography, scientific computing, and many other applications. FPGAs are particularly well suited to the type of highly parallel, bit-level computations found in genomics algorithms. Sequence alignment, for example, has long been computed efficiently on special-purpose FPGA platforms.

Unfortunately, the wider use of FPGA platforms among genomics researchers has been limited by the expertise required to use these platforms, which currently requires specialized hardware-design knowledge. This “programmability barrier” has been the focus of much research, but in spite of the substantial advances that have been made in behavioral synthesis and C-to-Hardware tools, today's tools are still inadequate for mapping large, complex applications to hardware. Our approach is to adopt a set of techniques and concepts that are used for programming parallel and distributed computers, and adapt them to the problem of programming hardware accelerators. This leverages a hardware-centric computation model that enables parallel algorithms to be described at a relatively high level of abstraction and then mapped automatically across a large number of computation nodes including processors and configurable hardware.

Another limiting factor has been a proliferation of FPGA platforms, each of which has a different system architecture that generally requires a re-implementation of the algorithm, even for next-generation boards from the same vendor. Figure 1 shows some example FPGA platforms with very different characteristics. These range from small, single-

FPGA systems to large multi-processor systems with attached FPGA accelerators. These accelerators are all attached to one or more processors, which typically perform a large part of the application, farming the data-intensive computation to the accelerator. Each platform provides a different interface, to which the application designer must adapt. One of the goals of our programming model is to provide clean, platform-agnostic ways to communicate between software and hardware components, and to communicate between components residing in different nodes.

Data-intensive applications also require large memories that can be accessed efficiently, and we assume accelerators are attached to large memories that are potentially shared with the host processor. Our model provides explicit support for large, distributed memories and allows the programmer use both pipelined and parallel memory accesses. Making the memory visible to the programmer allows very high memory bandwidth to be achieved.

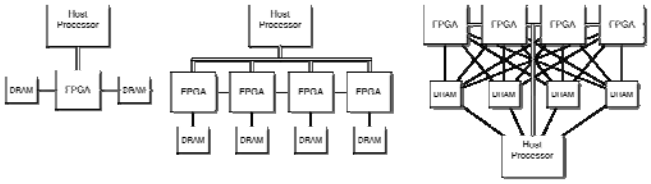

Figure 1. Example FPGA-based platforms

This paper starts with a description of the fundamental aspects of Elan. We then present the short read alignment application and show how it can be programmed in our model. Next we discuss advanced features of Elan that are used to extend a program to utilize multiple computational nodes and memories to achieve high data and computational bandwidth.

II. The ElanProgramming Model

We have defined a programming model called Elan for implementing large-scale, data-intensive applications on FPGA-based platforms. This section presents the salient features of Elan, including the execution model for direct hardware implementation and the features that allow the programmer to describe different parallel implementations of a program.

The Elan model adopts many parallel and distributed programming concepts that are used for large parallel systems. In so doing, we constrain the programmer to some extent, but such constraints are critical for managing and reasoning about the complexity of large parallel hardware/software systems. An advantage of applying concepts from the parallel programming world is that it allows applications to be partitioned and mapped to a combination of software and hardware resources using a single, consistent programming model. We rely on a modular, object-oriented approach that allows hardware and software implementations of a module to be interchanged, for modules to be reused across different applications and platforms, and for different hardware implementations to be generated based on performance characteristics of the target platform. Although Elan can be used for small systems comprising just a single FPGA, it is especially relevant for large systems with many CPUs and FPGAs.

A. Modules and Functions

The Elan programming model is based on hardware modules: A system is comprised of modules, each containing the data and functions that implement the functionality of that module. Modules are similar to software objects, except that they are implemented directly in hardware and are thus statically instantiated at compile time. Although modules can be virtualized and implemented on a processor, we focus on hardware modules and the specialized communication and computation structures used to implement them. Since modules are easily implemented in processors, a system can be built using a combination of hardware and software, and functionality can be moved between hardware and software by specifying whether a module is implemented virtually in a processor or physically in the FPGA fabric.

A module can have state in the form of data, which may be in registers or in privately owned memories, and can be manipulated or exposed through functions. Functions are computational units that may execute concurrently, but each function can only execute a single invocation at a time. Thus data synchronization for a single function is achieved automatically. Although conceptually only a single call is executed at a time, however, function execution can be pipelined to achieve greater performance, as long as that pipelining respects inter-call data dependencies.

Modules interact by calling each other's functions to initiate actions, cause computations to be done, or operate on data that belongs to the module. Function calls are by default asynchronous in the style of active messages: they do not return values, and the caller can continue execution immediately after calling an asynchronous function.

```
module Accumulator {
  int:32 sum;
  unsigned int:32 valuesAdded;
  sync void init() {
    sum = 0;
    valuesAdded = 0;
  }

  void add(int:32 value) {
    sum += value;
    valuesAdded++;
  }
}
```

Figure 2. Example accumulator module, with two methods

Asynchronous function calls are often used to implement pipelined dataflow graphs, streaming the data from one module to another via function calls. Synchronous function calls require the caller to wait for the function to complete, at which point the return value, if any, is surfaced to the caller. Synchronous functions are used sparingly, but they can be used to synchronize modules explicitly.

Figure 2 shows a simple module with two functions. The first function is synchronous and is used to initialize the

module data. The second function is called to update the module data. Even though many calls to `add` can be made concurrently, since call invocations are executed sequentially, there is no need to synchronize the module's variables.

B. Arrays and the Memory Model

Managing memory access to overcome the high latency of large memories is essential to achieving good performance for data-intensive applications. Memories can be highly pipelined to achieve very high data bandwidth, but each access may take many cycles to complete. For example, the Convey [1] memory system can achieve extremely high memory bandwidth via deep pipelining, but read accesses can take more than 400 cycles. Our explicit memory model allows the user to exploit deep memory pipelines via a combination of pipelined execution and parallel tasks.

Each array allocated to DRAM memory is associated with a memory port module that implements functions for reading and writing the array. These functions provide a pipelined split-phase protocol that takes advantage of the pipelined memory interface. Prototypes for these basic functions used to access arrays are shown in Figure 3. The `read` function takes an index used to read the array along with a function reference that is called with the data once it is available. The `write` function takes an index, the data to be written to the specified index, and optionally a write-complete callback. Since writes do not return a value, no callback function is necessary, but for synchronization purposes the caller may wish to know when the write has finished. The `atomicUpdate` function performs an atomic read-modify-write operation on the array using a function provided as one of the arguments. This allows the caller to specify an arbitrary update function to be performed on the memory location. These array functions can be augmented to support more complex memory models. For example, write flush is useful when synchronizing arrays between phases of a computation. The system infrastructure for a platform must provide the hardware implementations of the array function call interface provided by our model.

Function pointers like those used by the memory interface are a general mechanism that allows the calling function to specify the destination of the next call. Most applications use simple memory access patterns, so a static call graph can be constructed by the compiler and implemented using point-to-point communication. However, our mechanism allows the destination to be dynamically determined for more complex applications. These calls are implemented using a dynamic routing network.

Before describing the remaining features of the Elan programming model, we will describe the algorithm for the short-read alignment problem, and show how it can be implemented using Elan in a form that can be executed efficiently in hardware.

III. SHORT-READ ALIGNMENT ALGORITHM

DNA sequencing is used in a large and growing number of ways. One of the most visible is "human genome (re)sequencing", which is the process of determining the genome sequences of one individual, as distinct from the "human reference sequence", which is actually a mosaic drawn from several individuals. In this application, each read from the target genome is mapped (or "aligned") to its best match in the reference genome; the consensus of the aligned reads together define the target genome. This works since human genomes are extremely similar, differing in perhaps 1 in 1000 base-pairs. While many reads will match exactly to the reference genome, many do not, either because there was a read error or because there is a real difference between the reference and target genome at that location.

```
module Array<type> {
  void read(void &callback(type),
            unsigned int:32 index) {
    callback(memory[index]);
  }
  void write(unsigned int:32 index,
             type value) {
    memory[index] = value;
  }
  void atomicUpdate(type &update(), void
              &callback(type), unsigned
              int:32 index) {
    type temp = memory[index];
    memory[index] = update(temp);
    callback(temp);
  }
}
```

Figure 3. Example array using our memory model.

Next generation sequencing technologies have appeared in recent years that are completely changing the way genome sequencing is done. New platforms like the Solexa/Illumina and SOLiD can sequence billions of base pairs in the matter of days. However, the computational cost of accurately reassembling the data produced by these new sequencing machines into a complete genome is high and threatens to overwhelm the cost of generating the data [6]. Providing a low-cost, low-power, high-performance solution to the resequencing problem has the potential to make the sequencing of individual genomes routine [5].

To simplify somewhat, a DNA sample of the target genome is prepared by slicing the genome at random into many small subsequences, typically 35 to a few hundred base-pairs (ACGT) in length. Next generation sequencing machines then "read" the base-pair string for each of these subsequences, called "reads". These new sequencing machines can perform these reads in parallel to generate hundreds of millions of reads in a matter of days. This is done on many copies of the DNA sequence, resulting in many overlapping reads, which guarantees coverage of the entire reference sequence and allows a consensus to be achieved in the presence of errors in the reading process.

Our alignment algorithm finds the best positioning for each read, and is based on the BFAST algorithm [2]. The alignment is found in two steps. In the first step, a set of "candidate alignment locations" (CALs) is collected for the read using an index of the reference genome. This index is a hash table mapping all length N subsequences of the genome

to the set of locations where that sequence occurs [3]. To find the CALs for a read, we look up in the index each subsequence of length N in the read, called a seed. This returns a list of all the locations where that seed is found in the reference. The set of all CALs for the short read is formed by finding the locations of all the seeds in the read. If at least one of these seeds is free of mismatches and indels (insertions or deletions), then one of these CALs will give the actual location of the read in the reference genome.

In the second step, the read is compared to the genome at each of the candidate locations using a full Smith-Waterman [9] scoring algorithm, and the location that has the best match to the read is reported. The Smith-Waterman algorithm is a dynamic programming algorithm for performing approximate string matching that can be mapped very efficiently to a systolic array implementation in hardware. It is important both when finding the set of CALs, and when evaluating the match for each CAL, that the algorithm handle multiple mismatches caused by single nucleotide polymorphisms (SNPs; i.e., single differences between the target and reference genomes) and read errors, as well as insertions and deletions (indels). Although these occur infrequently, it is these cases that are the most biologically interesting and thus most important to identify. Using the full Smith-Waterman algorithm is the best way to find alignments in the presence of all these features.

A typical short read alignment problem for the human genome involves aligning 600 million short reads, which is about 20–40 GB of data, to a genome of 3 billion base pairs. The size of the index table is about 20GB, and the size of the reference genome is 1–4 GB depending on the data representation. Performing an alignment in software using a multi-threaded program on a high-performance workstation takes about 9-12 hours. This section describes how the Elan programming model is used to describe a parallel hardware implementation of a short read alignment algorithm.

Figure 4 gives a high-level program for this alignment algorithm. The CALs for each seed are retrieved using an index of the reference comprised of two tables, the `PtrTable` and the `CalTable`. The `PtrTable` is a hash table that maps each seed to a pointer into the `CalList` that gives the list of CALs for that seed. A tag is used to identify the CALs associated with the seed since the CALs for multiple seeds are stored in one entry of the hash table. The size of the `PtrTable` is adjusted to keep it small and efficient while avoiding too many unnecessary lookups in the `CalList`. Many of the details of this program [7][4][8] have been elided for clarity. While these details are important, they do not affect the explanation of the Elan model.

Using hardware to accelerate this program requires addressing several different issues. The first is that the large-scale multi-threaded parallelism used by software programs is not appropriate for hardware. The outer loop of this program is a forall loop—each iteration is independent and can be executed by a separate thread, which may be interleaved on a single core and/or allocated across many cores and processors. A big difference between software and hardware execution is that threads are virtual in software, while they are physical in hardware. That is, computation is mapped directly into components that execute that computation. High performance is achieved on a single thread both by executing operations in parallel (instruction-level parallelism), and also via fine-grained pipelining. Coarse-grained task-level parallelism is expensive in hardware since each task is mapped to physical hardware. In our model we try to achieve as much performance as possible via instruction-level parallelism and pipelining before resorting to multiple-thread, task-level parallelism.

```
for (i=0; i < NUMREADS; i++) {
  shortRead = shortReads[i];
  filter.init();
  maxScore.init();
  for (j=0; j < NUMSEEDS; j++) {
    seed = shortRead.seed(j);
    (index, seedKey) = hash(seed);
    calList = PtrTable[index];
    for (k=0; k < calList.size; k++) {
      (cal, key) = CalList[k];
      if (key == seedKey &&
          !filter.contains(cal)) {
        filter.add(cal);
        ref = reference[cal];
        (pos, score) =
          SmithWaterman(shortRead, ref);
        maxScore.enter(pos, score);
      }
    }
  }
  host.report(maxScore.max());
}
```

Figure 4. Pseudocode for short read alignment.

The next issue is memory latency and bandwidth. Three arrays in this program, `PtrTable`, `CalList` and `reference`, are very large, multi-gigabyte arrays that must be stored in DRAM. These arrays are accessed mostly in random order, which means that caching is not effective for reducing latency. Thus, the time to execute each iteration of this program is dominated by the memory latency, which may be 10s to 100s of clock cycles. If memory accesses can be pipelined along with the rest of our program, then we can hide this latency.

The final issue is the call to the `SmithWaterman` function, which invokes a Smith-Waterman[9] string comparison between the short read and a substring of the reference. This is a complex function that is very slow in software. However, there is a well-known dynamic programming algorithm that maps well to hardware and achieves very good performance.

The goal of the Elan programming model is to allow a programmer, or perhaps a smart compiler, to transform a program like this into an efficient hardware implementation running on an FPGA accelerator.

IV. Program Transformation

In this section, we will transform the short read program of Figure 4 into a program written in the Elan programming model that can be directly compiled into hardware. This

transformed program will retain the structure of the original program, but will use deep pipelining to achieve high memory bandwidth and performance.

Figure 5 shows the first module in the pipeline, which implements the outermost two loops of the program of Figure 4. This module has two functions. The first function, `getReads`, is called by a host process to begin execution of the program. It performs the first step of each iteration, which reads each short read from the `shortRead` array. Since the `read` function is asynchronous, `getReads` does not wait, but can start all the array accesses for the short reads at a rate of one per clock cycle.

The call to `read` provides a pointer to the `newRead` function as an argument. As data is read from the DRAM, it is sent to the `Main` module via a call on the `newRead` function. This function executes the second loop of the program, which divides each short read into a set of seeds, and initiates a read of the pointer hash table for each seed. Note that these two functions execute concurrently, although the `newRead` function will be working on a much earlier iteration than the `getReads` function because of the pipelined memory accesses. Once the memory read pipeline has filled, however, memory accesses will occur at the maximum rate supported by the memory.

```
module Main {
  ...
  void getReads() {
    for (int:32 i=0; i<NUMREADS; i++) {
      shortRead.read(&newRead(), i);
    }
  }
  void newRead(ShortRead shortRead) {
    smithWaterman.newRead(shortRead);
    for (int:32 j=0; j<NUMSEEDS; j++) {
      Seed seed = shortRead.seed(j);
      (int:32 index, int:32 key) =
          hash(seed);
      ptrTable.read(
        &filter.ptrData(key), index);
    }
  }
}
```

Figure 5. The `Main` module. It executes the start of each iteration, beginning the flow of data through the pipeline.

Again, the `ptrTable.read` function call provides a pointer to the function `filter.ptrData` that the memory calls to return the data. This function is part of the `Filter` module, which is shown in Figure 6. This `ptrData` function requires two arguments, while the memory read function expects a pointer to function of only one argument. The call to `read` supplies the first argument and partial evaluation uses this to turn the `ptrData` function into a function of one parameter as expected by the `read` function. This allows the `newRead` function to forward the key value to the `Filter` module along with the memory access.

```
module Filter {
  ...
  void ptrData(int:32 key,
               CalList calList) {
    for (int:32 k=0; k < calList.size;
         k++) {
      calTable.read(&calData(key),
                    calList.ptr + k);
    }
  }
  void calData(int:32 seedKey,
               CalEntry cal) {
    if (seedKey == cal.key) &&
     (!filter.contains(cal.pos)) {
      filter.add(cal.pos);
      refArray.read(
        &smithWaterman.refData(),
        cal.pos);
    }
  }
}
```

Figure 6. The `Filter` module reads the CALs, filtering out repeats. It then calls the Smith-Waterman computation.

The `Filter` module uses the information returned by the hash table to initiate the reads of the `calTable`. The key value is again forwarded with the data read from the `calTable` to the `calData` function, which now uses it to determine which CALs belong with the current seed.

```
module SmithWaterman {
  Read shortRead;
  void newRead(Read read) {
    this.shortRead = read;
  }
  void refData(Reference ref) {
    (pos, score) = SW(shortRead, ref);
     maxScore.enter(pos, score);
  }
}
```

Figure 7. The Smith-Waterman unit itself.

The SmithWaterman module of Figure 7 performs the Smith-Waterman comparison between the short read and the reference subsequence identified by the current candidate location. This module has two methods. The first, `newRead`, is called by the `Main` module and provides the current short read to be compared. The second, `refData`, is called by the `RefArray` module with the result of reading the reference array at the current CAL position. The `SW` function refers to a hardware function that performs the pipelined Smith-Waterman calculation and returns the best score and associated alignment in the reference subsequence. This function takes on the order of 100-200 FPGA clock cycles, and thus may represent a bottleneck depending on how often the `Filter` module finds a new, unique CAL to process. The `SmithWaterman` module calls the `MaxScore` module, shown in Figure 8, with the result of the comparison to enter this alignment for the current read.

The `MaxScore` module just keeps track of the highest score found for the current short read. In some applications,

it could simply filter all scores below some threshold, or keep a list of the highest N scores for the short read.

```
module MaxScore {
  int:32 maxScore = 0;
  int:32 maxPos;
  void enter(int:32 pos, int:32 score) {
    if (score > maxScore) {
      maxScore = score;
      maxPos = pos;
    }
  }
}
```

Figure 8. The `MaxScore` module, which aggregates the individual scores for all CALs of a give read and saves only the highest scoring location.

A. Pipeline Synchronization

Our transformed program executes the original program in a pipelined fashion. Even though this pipeline is composed of concurrently executing function calls, it is useful to think of it as a single thread that is executing via hardware pipelining as opposed to software pipelining.

The transformed program correctly executes a single iteration of the original program in a pipelined fashion; however, it does not handle multiple iterations correctly. That is, there is no mechanism for determining where one iteration ends and the next iteration begins. For example, the `MaxScore` module must re-initialize the best score for each short read. We specify this synchronization by putting a barrier at the end of the outer loop, which separates one iteration from the next in the computation.

There two types of barriers. The first is a true barrier, which must be used if one iteration depends on data produced from an earlier iteration. These barriers require that the pipeline be flushed between iterations. One might think that a barrier might greatly limit performance, but if the loop body contains a large computation and the pipeline flush happens only infrequently, then the performance loss can be minimal. There is also the opportunity for "relaxed barriers", where the next iteration can begin when the data required by loop-carried dependencies becomes available.

We call the second type of barrier a "fence", which can be used by for-all loops such the loop in this program. A fence simply requires that two iterations be separated in the pipeline. This fence notifies each module when one iteration has ended so it can begin the next.

There are two ways to implement barriers and fences. The first method attaches a system flag to each function call, and the end of an iteration is denoted by setting this flag. A function can test this flag to determine whether a barrier has been reached and can forward the barrier via its own function calls.

The second version uses dataflow accounting, which keeps track of the amount of work entering the pipeline and the amount of work completed. This is useful when the amount of work can be statically determined. For example, in our program the number of seeds in each read is constant, so the barrier/fence can be implemented by a simple count. Dataflow accounting is more complicated in the general case, but can be useful in systems where calls may be delivered out of order.

In our program, a combination of barrier flags and dataflow accounting is used to implement the fence at the end of the outer loop. (These have not been included in the example code in the interest space and clarity.) No fence call is required by the `Main.getReads` method since exactly one call is made per iteration on the `Main.newRead` method. Similarly, the `Main.newRead` method does not need to make a fence call since `Filter.ptrData` knows that exactly `NUMSEEDS` calls (the number of seeds per read) will be made per iteration.

`Filter.ptrData` does need to make a fence call on `calTable.read` after it completes processing the last call for the iteration, since the number of CALs per seed is not known at compile time. This fence is forwarded to `Filter.calData`, which then forwards it through `RefArray.read` to the `SmithWaterman.refData` function. Since there is one short read per iteration, the `SmithWaterman` module waits for the fence before executing the next `newRead` function call. This synchronization between the `newRead` and `refData` functions is accomplished using a shared full/empty flag.

When the fence finally makes it to the `MaxScore` module, it causes the computed maximum score and alignment for the current short read to be reported via a function call to the Host processor, and the values to be re-initialized.

V. Parallelization via Module Replication and Distribution

There is already substantial parallelism in the implementation as described since the operation of the modules and the memory accesses are pipelined to take full advantage of available memory bandwidth. Increasing the performance of this pipeline requires understanding the bottlenecks and introducing concurrency to remove them. In general, increasing the performance of an application revolves around removing three bottlenecks: computational bandwidth, memory bandwidth, and communication bandwidth. In the short read alignment application, there are two primary bottlenecks: the memory bandwidth for reading the CALs from the `calTable` array, and the computational demands of the Smith-Waterman comparison. Depending on the seed length, tens to hundreds of CALs may be read from memory for a short read and filtered down to only a few unique CALs. Each unique CAL then results in one Smith-Waterman unit comparison, which then takes roughly 200 clock cycles to complete. Based on the ratio of unique CALs produced to the total number read from the `calTable`, which depends on the seed length and indexing parameters, one or the other of these bottlenecks will be encountered.

Computational bandwidth is increased by replicating modules so that independent computations can proceed in parallel. Memory bandwidth is increased by partitioning arrays across multiple memories or memory ports. In both cases, this partitioning and replication may occur within a

single node, or across multiple nodes, depending on the memory and computational resources available in a platform. Communication bandwidth is increased by taking advantage of locality, that is, by making sure modules are located near the memories and other modules with which they communicate.

A. Memory Partitioning

Arrays are partitioned across multiple memory modules by specifying a *partition map function* that maps array indices to memory partitions. Partitions are then distributed across memory modules via a "locale" map function. Once partitioned, array access method calls are automatically directed to the appropriate partition using the partition and locale map functions.

Assuming that our platform has multiple memories or memory ports, we can address the memory bandwidth bottleneck posed by the `calTable` by partitioning it across multiple memory modules or ports. For example, the Convey platform provides 16 memory ports in each FPGA, and we might allocate 4 or 8 of these ports to the `calTable`. This requires only one change to the program: the `calTable` array declaration is declared a "partitioned" array with a partition map function that describes how the array is partitioned. Calls to the array access functions are automatically sent to the appropriate memory module or port using the map function and the array index.

For the short read application, the mapping function for the `calTable` and `ptrTable` would be blocked or interleaved based on the array index, since the hash function already distributes the entries relatively uniformly across regions of the tables. For arrays with more non-uniform accesses (such as raw DNA sequences which can have unequal base distributions), hashing the address or the use of caching can reduce hotspots in the memory access patterns.

We adopt a partitioned memory model that allows multiple memories and memory ports to be used to increase memory bandwidth without the overhead of coherence protocols. This means that each memory port only ensures that memory accesses to that port are ordered correctly.

B. Module Replication

Computational bandwidth is increased by replicating modules that pose a computational bottleneck. For example, in our program, we would replicate the `SmithWaterman` module to allow multiple `SmithWaterman` comparisons to proceed in parallel. This is done by changing the declaration of the `SmithWaterman` module to specify that it is a "distributed" module, specifying how many copies of the module should be constructed, and defining a *distribution map function* that describes how a given parameter, or set of parameters, shared by all function calls, are mapped to a given module. This creates an array of modules and specifies how function calls are sent to the appropriate module of this array.

The programmer must decide how best to partition, replicate and distribute the memories and modules. In this program, we could use a distribution map function on the `shortRead` ID number to allocate `<shortRead, ref>` comparisons to `SmithWaterman` units. This would map all comparisons for one read to the same `SmithWaterman` unit. This allows better pipelining (since the next `ref` can begin processing before the previous one is done since the `shortRead` is the same), but can cause load-balancing and reordering costs. Alternatively, we could simply allocate each comparison to the next free `SmithWaterman` unit by using a map function over both the `shortRead` ID number and the reference CAL. This type of allocation of replicated modules is described in the next section.

The programmer must design the appropriate array partitioning and accompanying replication of computational modules to achieve the best performance by maximizing memory bandwidth and minimizing inter-node communication. However, distributions provide a high-level way to describe these partitioning and replication map functions. The programmer describes a parallel implementation simply by defining these map functions and the compiler automatically partitions the memory, duplicates the modules, implements method calls using the most efficient means, and routes method calls to the appropriate modules.

Distributions allow the partitioning and parallelization of an implementation across a large number of nodes to be described concisely and implemented automatically. Changing the parameters and the shape of a parallel implementation is thus easy for the programmer to specify. This greatly reduces the effort to tune an implementation and port applications to new platforms—the algorithm is cleanly separated from the platform-specific implementation.

C. Dynamic Modules and Dynamic Module Allocation

It is often convenient to allocate modules to tasks automatically. For example, it is convenient to treat a set of `SmithWaterman` modules as a fixed-size pool of modules that can be allocated automatically as they are needed. Our model supports this automatic allocation and de-allocation by allowing modules to be declared as dynamic. Dynamic modules are like other modules, but are kept in a pool by the runtime system and allocated dynamically as needed when method calls are made. Each method call to a dynamic module specifies a module ID. When a method is called with a new module ID, a module with that ID is automatically allocated from the pool. Subsequent method calls to a dynamic module with the same ID are handled by the same module, and when the module has completed its task, it de-allocates itself and is placed back in the pool. This mechanism allows the programmer to think of there being a large number of modules, one for each task, and lets the runtime system automatically multiplex tasks across the set of available hardware modules.

In the short read application, we could define the `SmithWaterman` module as a dynamic module with the `(read ID, CAL)` pair as the dynamic module ID. This would cause each Smith-Waterman comparison to be allocated to a new module from the pool. When the comparison is completed and the result has been reported via the `newScore` function call, the module would deallocate itself

and be returned to the pool. In this way, the number and location of actual `SmithWaterman` modules can be changed without having to otherwise change the program.

VI. Compiling Programs to Hardware

A program written using the Elan programming model is converted to hardware in three steps. In the first step, all hardware modules are constructed. Each module has a set of input ports that correspond to the function call interfaces defined by the module. Each of these function call ports is connected to a FIFO that buffers incoming function calls. An output port is added to the module for each function call made by the module's functions. Each function is implemented using a finite state machine that performs the computation of that function. This FSM waits in an idle state until a function call appears in the call FIFO, and then executes the function using the parameters in the call FIFO. Any function calls made by the function are accomplished by putting the arguments on the output port and initiating a handshake that sends the call as a packet. When done, the FSM returns to the idle state. Where data dependencies allow, the FSM can be pipelined so that several function calls can be executed concurrently.

The second step constructs the call graph that connects the modules. In most cases, this call graph can be determined statically. That is, it can generally be determined at compile time exactly which module a function call is being invoked on. In this case, a point-to-point link is made in the call graph. For distributed modules and partitioned memories, a function call may be invoked on any one of a set of modules. In this case, each call has a fanout to a set of modules.

In the final step, the modules and the call graph are mapped to the physical resources of the hardware platform. The programmer specifies how modules should be mapped to nodes in the system, and the compiler maps the call graph to system communication resources. Each function call is a packet comprising the function arguments, sent from the source module to the destination module. This packet delivery can be done in the simplest case using dedicated wires and a handshake with the destination call FIFO, or in the most general case using a packet-switched network spanning multiple nodes. Other options include using shared, time-multiplexed buses for low-bandwidth links, and independent switched networks for independent parts of the call graph. Optimizing the system communication using an analysis of the data bandwidth on each link to map the call graph efficiently to the communication resources of the platform is a major research challenge.

We do not yet have a full compiler for Elan. We are experimenting with Elan by writing programs in a Java "sandbox" that implements the programming model using Java threads and implicit call FIFOs. Programs run in this sandbox using concurrent execution that allows us to test and debug programs before mapping them into hardware.

After a program has been debugged, it is run through a simple parser that generates the Verilog for the system architecture. This uses annotations added to the Java program that specifies the bit-width of values so that the appropriate Verilog declarations can be made. This system architecture includes all module interfaces, including call FIFOs, and the complete implementation of the call graph using switching networks for those cases where calls fan out to multiple modules, and where calls fan in to a single module.

The final step is to write the finite state machines for each of the module functions. This is currently done manually, although we are starting to experiment with using AutoESL [10] from Xilinx to do this automatically.

Although we do not yet have a full compiler, we have been able to experiment with a number of applications including vector add, breadth-first search, sparse matrix-vector multiply and short read alignment. We have targeted both the Pico Computing M503 FPGA board and the Convey HC-1, which have very different platform architectures. These experiments have allowed us to analyze the efficacy of Elan for hardware systems, understand the compiler optimization issues and opportunities, and estimate the cost and performance of the resulting hardware systems.

A. Mapping the Short Read Application to Hardware

We have designed the hardware infrastructure required by our model for both the Pico Computing M501 platform and the Convey HC-1 FPGA-based supercomputer. This infrastructure supports the pipelined memory interface of the individual platforms, a procedure call-based host interface, and the routing network needed for partitioned arrays and distributed modules.

The Convey HC-1 computer is an FPGA-based supercomputer comprising an Intel Xeon quad-core processor and 4 Xilinx Virtex5 LX330 FPGAs [1]. The processor and FPGAs are connected by an interface that allows them to share up to 64 GB of memory. Each FPGA has 16 ports to this memory; each port supports 64-bit pipelined memory accesses to random addresses at the rate of 150MHz.

We mapped the program described above to the Convey platform via the process described above. We first tuned the program for one FPGA, and then replicated it across all four FPGAs, giving each FPGA ¼ of the short reads to process. Because all 4 FPGAs on the Convey platform share the same memory, the index and reference arrays can be shared by the four replicas. We can fit 4 `SmithWaterman` modules in each FPGA for short read lengths of 100 base-pairs. In our current implementation, we do not use dynamic modules, but rather allocate `SmithWaterman` modules to each short in round-robin order as described in Section V.B.

We allocate 1 memory port for reading the short read array, a second port for reading the `PtrTable`, 8 ports for reading the `CalTable` and 1 port for writing the score results back to memory. The pipelined memory performance of the Convey is such that the bottleneck becomes the Smith-Waterman computation. The results so far show that the performance and cost of this implementation are competitive with a manually written Verilog implementation of the short-read algorithm. In particular, the system infrastructure used by the compiler does not cause any performance degradation and allows the program to achieve the full memory interface bandwidth. With 4 FPGAs and 4 Smith-Waterman units per FPGA, and 100 base-pair short reads, we can perform about 15 million full Smith-Waterman comparisons per second.

For a seed length of 22 base-pairs, the expected computation rate is over 2 million short reads per second. By comparison, BFAST running on a 6-core Intel Xeon processor processes about 10,000 short reads per second[1].

Retargeting this program to the Pico M503 platform **Error! Reference source not found.** is straightforward. The Pico M503 platform is a single board with one Xilinx Virtex6 LX240T FPGA connected to 8 GB of DRAM and 24 MB of SRAM, which is not used in this application. The M503 provides two multi-ported interfaces to two 4 GB DRAM modules. Since these ports compete with each other, unlike on the Convey platform, there is no advantage to assigning more than one port to each array. In addition, the short read array is read from disk by the host, and thus the `Main` module is split into two modules: the `getReads` function in one module running on the host processor, and the `newRead` function in a second module running on the FPGA. The function call to `newRead` is automatically carried out using the streaming interface provided by the Pico platform.

Compiling the program to a platform comprising multiple Pico boards, each with its own FPGA and memory module, requires the system infrastructure for delivering the call packets between FPGAs. We have not yet implemented this infrastructure, which would use the PCI-express interconnect between the boards, a set of dedicated inter-FPGA connections or a combination of the two. A program to use multiple FPGAs without shared memory would need to either replicate the index across the boards, which would waste memory, or partition the index across the boards, which would require each short read to be processed by all boards and for the results to be combined.

VII. Related Work

There are many short read mapping software tools that tackle the problem of processing the enormous amount of data produced by the next-generation sequencing machines. These alignment solutions tend to fall into two main algorithmic categories.

The first category of solution is based upon a block sorting data compression algorithm called the Burrows-Wheeler Transform (BWT) [14]. This solution uses the FM-index [15] to efficiently store information that allows all locations of a given substring to be found in time proportional to the length of the substring. Unfortunately, the running time of this class of algorithm is exponential with respect to the number of differences between the short read and the reference; therefore BWT-based algorithms cannot afford to be as sensitive as the other class of algorithm, making them inappropriate for some genomic applications. Bowtie [12] and BWA [13] are examples of programs based on BWT indexing.

The alignment algorithm we have described falls into the second category of algorithm, which uses an index of short subsequences called seeds to find candidate locations in the reference. The Smith-Waterman algorithm is used to compare the short read to the candidate locations in the reference, which allows many mismatches and insertions/deletions to be tolerated. BFAST [2] is an example program based upon this algorithmic approach.

A. FPGA-Based Short Read Alignment

Our group has published an FPGA-based implementation of the short read alignment application that is very similar to that described in this paper [8]. However, that implementation was designed entirely by hand for the Pico M503 FPGA platform and required over a man-year of effort. The goal of the research described in this paper is to reduce the time and effort required to implement algorithms using FPGA accelerators by at least an order of magnitude, while remaining competitive in both cost and performance.

Two other attempts to accelerate short read alignment on FPGAs used a brute-force approach to compare short sequences in parallel to an entire reference genome. They stream the reference genome through a system that matches of the short reads to the reference [16][17]. Reference [16] demonstrates a greater sensitivity to genetic variations in the short reads than Bowtie and MAQ, but the mapping speed was approximately the same as that of Bowtie. This system also demonstrated mapping short reads to only chromosome 1 of the human genome. Reference [17] demonstrates between 1.6x and 4x speedup versus RMAP [18] and ELAND [19] for reads with between 0 and 3 differences, for the full human genome.

In both implementations, the number of short reads that can be aligned in a single pass of the reference genome is limited by the number of block RAMs on the FPGA. Scaling to a larger number of short reads (the previously cited works mapped only 100,000 50-base and 1,000,000 36-base reads respectively) would require multiple passes of the reference genome, with consequent increase in runtime.

There have been many efforts to compile high-level languages like C to hardware and these are now becoming widely available (see the excellent survey by Cardoso, Diniz and Weinhardt [20]). By contrast, our work is focused on the system level; in fact we assume the existence of a "module compilers" like AutoESL to map simple C programs to hardware. Our model is similar in spirit to several other models that address the system level, particularly TDM-MPI [21], IBM's Lime [22], and HThreads [23], but differs in the use of higher-level abstractions for describing parallelism.

In most cases, an application can be described using a relatively small number of concurrent objects. Generating a parallel implementation involves partitioning the data and computation and distributing these across a large number concurrent objects. We borrow the idea of "distributions" from parallel programming languages like Chapel [24] and X10 [25] to describe how objects in our model are duplicated and distributed to achieve large parallel implementations.

[1] We are currently finalizing the hardware implementation for a single FPGA. The final version of the paper will contain the performance results of running the full 4-FPGA system on full representative data sets for the human genome. We will generate a detailed comparison of the performance of our system to a manually designed short-read alignment implementation, as well as the software version running on a multi-core server.

Distributions allow the partitioning and parallelization of an implementation across a large number of nodes to be described concisely and implemented automatically.

VIII. Conclusions

In this paper, we have presented the Elan programming model and compiler used to target systems with multiple FPGAs and CPUs. By harnessing concepts from parallel computing, we provide an efficient method for specifying high-performance applications. We also have mechanisms for increasing the parallelism in computations and have efficient memory interfaces to support these systems.

The key to the Elan model is the ability of the programmer to specify parallelization decisions conveniently via the distribution and locale map functions. The compiler uses these to partition and allocate arrays to memory, replicate modules, and generate the necessary communication channels between these modules. Tuning the structure of the computation to the application and the platform is relatively straightforward and transparent compared to that required using traditional hardware design tools. This also makes it easier to port applications from one platform to another.

System-level design is also simplified by our model, since modules can be implemented in software or hardware. This means an application can be designed as a single program and split between the processor nodes and accelerator nodes depending on the platform architecture.

We have also demonstrated how we have used this environment to develop an implementation of the short-read alignment problem, a key step in modern genomics applications. This paper demonstrates a viable direction for the programming of high-performance accelerators in bioinformatics and other streaming application domains.

Acknowledgments

This work was supported in part by NSF grant #CCF-1116248, as well as Pico Computing, Convey Computing, Xilinx, and the Pacific Northwest National Lab.

References

[1] http://www.conveycomputer.com/

[2] N. Homer, B. Merriman, S. F. Nelson, “BFAST: An Alignment Tool for Large Scale Genome Resequencing”, *PLoS ONE*, Vol. 4, No. 11, 2009.

[3] D. S. Horner, G. Pavesi, T. Castrignan, P. D. D. Meo, S. Liuni, M. Sammeth, E. Picardi, and G. Pesole, “Bioinformatics approaches for genomics and post genomics applications of next generation sequencing”, *Briefings on Bioinformatics*, vol. 11, no. 2, 2010.

[4] Maria Kim, *Accelerating Next Generation Genome Reassembly in FPGAs: Alignment Using Dynamic Programming Algorithms*, M.S. Thesis, University of Washington, Dept. of EE, 2011.

[5] E. R. Mardis, “The impact of next generation sequencing technology on genetics”, *Trends in Genetics*, vol. 24, no. 3, pp. 133‑141, 2008.

[6] J.D. McPherson, “Next generation gap”, *Nature*, vol. 6, no. 11s, 2009.

[7] Corey Olson, *An FPGA Acceleration of Short Read Human Genome Mapping*, M.S. Thesis, University of Washington, Dept. of EE, 2011.

[8] Corey B. Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, Walter L. Ruzzo, “Hardware Acceleration of Short Read Mapping”, to appear in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2012.

[9] M. S. Waterman and T. F. Smith, “Rapid dynamic programming algorithms for RNA secondary structure”, *Advances in Applied Mathematics*, vol. 7, no. 4, pp. 455 – 464, 1986.

[10] http://www.xilinx.com/products/design-tools/autoesl/

[11] http://www.picocomputing.com/

[12] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, March 2009.

[13] Heng Li and Richard Durbin, "Fast and accurate short read alingment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754-1760, July 2009.

[14] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Technical report 124, 1994.

[15] Paolo Ferragina and Giovanni Manzini, "Opportunistic Data Structures with Applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, Washington, DC, 2000, p. 390.

[16] O. Knodel, T. B. Preusser, and R. G. Spallek, "Next- generation massively parallel short-read mapping on FPGAs," in *2011 IEEE International Conference on Application- Specific Systems, Architectures and Processors*, 2011, pp. 195-201.

[17] Edward Fernandez, Walid Najjar, Elena Harris, and Stefano Lonardi, "Exploration of Short Reads Genome Mapping in Hardware," *2010 International Conference on Field Programmable Logic and Applications*, 2010, pp. 360-363.

[18] A. D. Smith, Z. Xuan, and M. Q. Zhang, "Using quality scores and longer reads improves accuracy of Solexa Read Mapping," *BMC Bioinformatics*, vol. 9, no. 128, pp. 1471- 2105, February 2008.

[19] O. Cret, Z. Mathe, P. Ciobanu, S. Marginean, and A. Darabant, "A hardware algorithm for the exact subsequence matching problem in DNA strings," *Romanian Journal of Information Scient and Technology*, vol. 12, no. 1, pp. 51-67, 2009.

[20] J. Cardoso, P. C. Diniz, and M. Weinhardt, “Compiling for Reconfigurable Computing: A Survey,” ACM Computing Surveys, vol. 42, no. 4, pp. 1–65, 2010.

[21] M. Saldana, A. Patel, C. Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow, “MPI as an abstraction for software-hardware interaction for HPRCs,” in Second Interna- tional Workshop on High-Performance Reconfigurable Computing Technology and Applications. HPRCTA 2008., Nov. 2008, pp. 1– 10.

[22] J.Auerbach,D.F.Bacon,P.Cheng,andR.Rabbah,“Lime:ajava-compatible and synthesizable language for heterogeneous architectures,” in Proceedings of the ACM international conference on Object oriented programming systems languages and applications,

[23] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, “Achieving Programming Model Abstrac- tions for Reconfigurable Computing,” IEEE Transactions on Very Large Scale Integration Systems, vol. 16, no. 1, pp. 34 –44, jan. 2008.

[24] B. Chamberlain, D. Callahan, and H. Zima, “Parallel Programmability and the Chapel Language,” Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, 2007.

[25] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An Object- oriented Approach to Non-uniform Cluster Computing,” in OOP- SLA ’05: Proceedings of the 20th Annual ACM SIGPLAN Confer- ence on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM, 2005, pp. 519–538.