Short-read Sequence Alignment in S#

Kevin Mitchell, Measurement Analysis Department, MRL

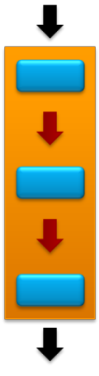
Introduction

S# is a .NET library that allows a developer to implement high-speed stream processing applications within Microsoft's .NET framework. Code written using S# can be executed on the .NET platform, or compiled to native code which can then be automatically embedded within the application. We can also generate code suitable for execution on an FPGA and in this working paper we illustrate this capability by developing a small example. We start with a brief overview of S#, and then describe the example problem, short-read gene sequence matching. The remainder of the paper consists of an S# solution to the problem, and concludes with some preliminary performance figures.

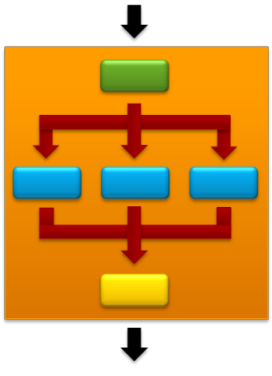
There is nearly always a trade-off between development time and efficiency when building software systems. High-level languages increase productivity, in terms of development and debugging time. But they typically incur some cost at runtime in comparison to carefully written code in a low-level language whose features are more closely matched to the underlying hardware. In most cases this trade-off is worth making unless the application really needs to be at the extreme edge of performance. As the software becomes more complex, or the requirements change rapidly, the performance penalty tends to decrease as it becomes harder to hand-tune the low-level code. For streaming applications we argue that it can be much faster to develop in a framework such as S#, where we have the expressive power of C# and a high-level debugger at our disposal. We show that the S# compiler is able to take such code and compile it for execution on an FPGA, without requiring the user to have a detailed knowledge of firmware design. Initial results suggest the performance of the resulting system, whilst perhaps not exploiting the capabilities of the device to its full potential, are good enough to provide a very useful trade-off between development effort and runtime performance for many applications.

S#

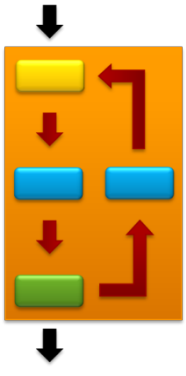
S# is a high-level stream programming library written in .NET for implementing streaming algorithms, e.g. video, DSPs, networking, and encryption. It is being developed within the Measurement Analysis Department, which is part of MRL. It was heavily influenced by MIT's StreamIt language [StreamIt], and is an example of a (mainly) Synchronous Dataflow language. By this we mean that the rate at which data is produced and consumed remains fixed during graph processing. Unlike many such stream languages we impose a rigid hierachical structure on the permissible graphs. All S# graphs are composed from the following components.



Pipeline



Split/Join



Feedback loop

The atomic elements of an S# graph are filters. Most computation takes place within these filters, and each filter has a single input stream and a single output stream. Note that each of the composite graph constructors also has this property. Many of the graph operations performed by the S# compiler rely on the structured nature of the graphs to simplify the analysis and ensure type-correctness.

As S# is a synchronous dataflow language each filter has rate information associated with it. The computation is performed within a work method, and we must define how much data is consumed and produced each time a work method is executed. For example here is a simple up-sampler expressed in S#.

```
public class UpSample : Filter<float, float> {
  private readonly int N;

  public UpSample(int N) : base(pop: 1, push: N) { this.N = N; }

  public override void Work() {
    Push(Pop());
    for (int i = 0; i < N - 1; i++) Push(0);
  }
}
```

Another distinguishing feature of the language is its use of peeking. In addition to consuming data from the input port, and producing data for the output port, a filter can also peek ahead in the input. This can be beneficial as it allows us to express some filters in a stateless fashion, where the filters themselves maintain no internal state. One of the key optimizations available to an S# compiler is the rearrangement of the stream graph to increase the potential for parallel execution. Such rearrangements often rely on the ability to duplicate parts of the stream graph, and it is a lot easier to duplicate a stateless filter than one that maintains state.

In addition to a set of stream-specific classes the S# system also provides a verifier and a compiler. To simplify the analysis and manipulation of filters and stream graphs we limit the expressiveness of code that can be used to define filters. If we allowed our S# filters to use the full power of C# it would be infeasible to do all the analysis we require. Furthermore, it would be easy for a user to accidentally introduce side-effects between graph components that would produce undefined behavior when executing the graph in parallel. The verifier, which runs as a post-processor, ensures that the code inside all classes derived from the `Filter` base class is limited to a safe subset of C#, whilst leaving the remaining classes in an application unaffected.

The job of the compiler is to convert the stream graphs into code that can be efficiently executed on a variety of platforms. Rather then spending more time describing the various language features at an abstract level we use the code for our example to showcase most of the language.

Short-read sequence alignment

Our chosen example for this experiment is short-read sequence alignment. This is a matching problem involving genomic sequences, which are strings over the alphabet $\sum = \{ A, G, C, T \}$. We will refer to the elements of this alphabet as bases. In the short-read sequence alignment problem we have a short-read sequence ***s*** consisting of at most a few hundred bases, and a reference sequence ***r*** that contains potentially billions of bases. For each position *i* in ***r*** we want to find the minimum number of (weighted) steps required to transform ***s*** into a subsequence of ***r*** starting at offset *i*. The mechanisms used to construct these sequences may introduce errors, and so the matching process must take this into account. The possible edits are insertion of new bases, deletion of bases and substitution of bases. We are interested in finding those positions in the reference sequence whose edit cost is below some threshold. In practical applications we may need to match millions of different short-read sequences against the same reference sequence. For simplicity we will assume

that all the short-read sequences have the same length, although it would be relatively simple to extend the example to support variable lengths.

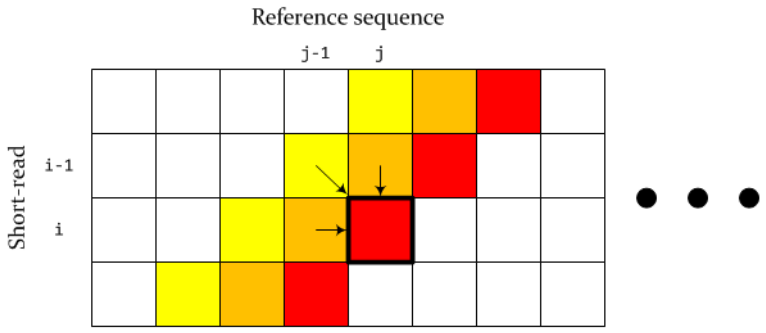
The short-read alignment problem has been exhaustively studied in the literature, and our solution will not involve developing any new techniques. However, we hope to show that by expressing the solution in the S# framework we can quickly develop code suitable for execution on an FPGA with very little firmware expertise. Furthermore, the changes we need to make to tailor the code for FPGA execution are small, allowing us to develop and debug the program on a PC.

Using dynamic programming for short-read sequence alignment

We can view the problem as a two-dimensional grid, with the rows indexed by the short-read sequence and the columns by the reference sequence. Each cell (i,j) can be viewed as the problem

> Compute the minimum cost of matching the short-read sequence $s_0..s_i$ against the reference sequence $r_0..r_j$ where we are allowed to drop initial elements of r in the matching process without incurring a penalty. That is, we are interested in computing local alignments of s against r.

Consider the cell highlighted in the following picture:



We want to compute the cost of matching the short-read sequence at position `i` against the reference sequence at position `j`. There are three ways we can perform this match.

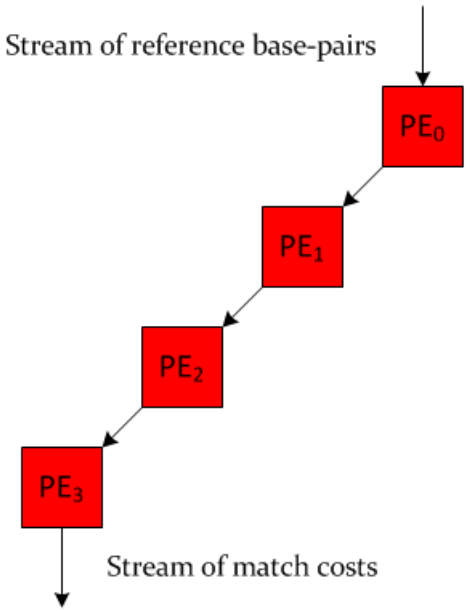
1. We can match the base at short-read position `i` with that at position `j` in the reference sequence, performing a substitution if the two bases differ. The total cost of matching the short-read sequence up to this point will therefore be the minimum cost of matching at short-read position `i-1` against reference position `j-1` plus the cost of the substitution.
2. We can perform an insertion of the base at short-read position `i`. The total cost will be the cost of this insertion plus the minimum cost of matching at position `i` against position `j-1`.
3. We can perform a deletion of the base at short-read position `i`. The total cost will be the cost of this deletion plus the minimum cost of matching at position `i-1` against position `j`.

To compute the best (smallest) cost for matching the two sequences at these positions we take the smallest of these costs. By computing the results from left-to-right along the anti-diagonal we end up with the Smith-Waterman algorithm.

Note that each of these computations requires computing a cost from an adjacent cell, as shown by the arrows in the diagram. We can compute the costs using a dynamic programming approach. The computations

performed by the red cells in the diagram are independent of each other. So we can perform these computations in parallel by building a set of processing elements whose length is determined by the length of the short-read sequence. Once the processing elements have computed the minimum costs at the current positions we can move the diagonal one step to the right and repeat the process, continuing until we have examined all the reference sequence.

The arrows in the diagram represent data dependencies. The horizontal arrow implies that each processing element must store the cost from the last iteration. But the diagonal and vertical arrows imply the need to access information that has been computed by a different processing element. If we connect all the processing elements together in a pipeline, with the first element in the pipeline corresponding to the processing element at the top of the picture, then the data dependencies correspond to upstream nodes passing results to their downstream neighbors. Furthermore, the diagonal dependency implies we must maintain not just the cost of matching the last base encountered, but the pair before that as well.



As with many such algorithms, the steady-state processing behavior is relatively easy to define, but initializing the local state takes a bit of care to ensure we generate the expected values. Furthermore, it should be clear from the diagram that the first few values generated by the pipeline will be meaningless, and so any downstream processing elements must ignore such values.

The short-read sequence alignment problem in S#

Given this description of the problem, and its solution using dynamic programming, we now turn it into code for S#. Our first task is to choose a representation for bases and costs. For a base we choose a C# `enum`:

```
public enum Base { A, C, G, T }
```

When running in C# each value of type `Base` will typically occupy 32-bits. However when we convert the code to run on an FPGA the compiler will automatically deduce that we only need two bits to represent each base.

There are some tradeoffs involved when choosing the representation of matching costs. The simplest approach might be to use something like an `int`. However, we want to minimize the size of the datatypes we use so that they occupy fewer resources on the FPGA, allowing us to deploy more matchers in parallel. For this experiment we follow the approach of [Stevens] and represent each cost by a 6-bit quantity. This gives us a good tradeoff between accuracy and size. However, there is a danger that the matching costs can exceed 2^6-1 in some examples, depending on the size of the short-reads and the matching penalties. If the overflow behavior was to wrap the result then we could potentially confuse a very bad match with a very good one. To avoid this we use saturation arithmetic for the cost type; if the result of an operation is greater than the maximum, it is set ("clamped") to the maximum.

S# comes with some tools that allow us to build arbitrary precision integer and fixed-point types, and these tools allow us to specify the desired overflow behavior. Furthermore, such types are handled efficiently when they are converted to run on the FPGA. Optionally we can also specify the C# type name to be used for the new type. The command

```
apg --name=MatchCost --sat -u 6
```

builds a .NET assembly called `MatchCost.dll` that defines a type called `MatchCost` representing an arbitrary-precision unsigned type six bits in length, with a saturated overflow behavior. We simply have to reference this assembly in our C# project and can then start using the type. When running the code in C# we can replace this reference by the phrase `using MatchCost = System.Int32` if we wish to trade bit-wise accuracy for speed.

Before defining the code for each processing element we must decide on the type of value passed between each element. Each processing element needs to know the last base processed by its upstream neighbor, the cost of matching that element, and the cost of matching the base before that one. We therefore define a C# `struct` that encapsulates this information. Note that input and output types in S# are constrained to be value types, and so we must use a `struct`, not a `class`, here.

```
public struct State {
  public readonly Base currentBase;
  public readonly MatchCost cost;
  public readonly MatchCost previousCost;

  public State(Base b, MatchCost cost, MatchCost previousCost) {
    this.currentBase = b;
    this.cost = cost;
    this.previousCost = previousCost;
  }
}
```

Now we have the basic building blocks we can define a processing element. This will take a stream of `State` elements as input and produce a stream of `State` elements as output. Furthermore, each time the processing element executes one step it will consume one state element from the input and produce one element on the output. So we say the filter has a pop rate of 1 and a push rate of 1.

If we look back at our earlier diagrams we see that each processing element will always be matching the reference sequence against the same value in the short-read sequence. If we only needed to match against a fixed short-read sequence then we could just hard-wire the value of the base into the definition of each processing element. However, in reality we want to match against potentially many short-read sequences and so the matching value in each processing element will change every time we start processing a new short-read sequence. To allow the processing elements to be reconfigured our input sequence will therefore consist of the short-read sequence, followed by the reference sequence, followed by the next short-read sequence, and so on. The code for the processing engine must be aware of whether it is currently receiving the reference sequence or the next short-read sequence. Furthermore, in the second case it must pick out the base in the short-read sequence corresponding to the position of the processing element in the pipeline, and store this ready for matching against the reference sequence. The code for the filter is shown below. Note that we are using a very simplistic set of penalties in this example. Real-life applications of the Smith-Waterman algorithm will typically use more sophisticated affine gap scores which penalize gap extension less than gap opening.

```
public class PE : Filter<State,State> {
  private readonly uint shortReadIndex;
  private readonly uint shortReadLength;
  private readonly uint referenceLength;

  private uint position; // Current position in input sequence (mod sequence length)
  private Base currentBase; // Base we are currently matching against
  private Base lastBase;
  private MatchCost lastCost;
  private MatchCost lastLastCost;

  // Penalty for deletion in short-read sequence
  public readonly static MatchCost Alpha = 1;

  // Penalty for insertion in short-read sequence
  public readonly static MatchCost Beta = 1;

  // Penalty for a base mismatch (substitution)
  [Inline]
  public static MatchCost Gamma(Base s, Base r) {
    if (s == r) return 0; else return 1;
  }

  public PE(uint index, uint shortReadLength, uint referenceLength)
    : base(pop: 1, push: 1) {

    this.shortReadIndex = index;
    this.shortReadLength = shortReadLength;
    this.referenceLength = referenceLength;
    this.lastCost = Alpha;
    this.lastLastCost = lastCost + Alpha;
    this.position = index == 0 ? 0 : shortReadLength + referenceLength - index;
    this.lastBase = 0;
  }

  public override void Work() {…}
}
```

The processing of the input stream is performed in the `Work` method.

```
public override void Work() {
  Push(new State(lastBase, lastCost, lastLastCost));
  State upstream = Pop();

  if (position >= shortReadLength) {
    MatchCost gamma = Gamma(currentBase, upstream.currentBase);
    MatchCost diag = upstream.previousCost + gamma;
    MatchCost left = lastCost + Alpha;
    MatchCost up = upstream.cost + Beta;

    MatchCost cost =
      up < left ? (up < diag ? up : diag) : (left < diag ? left : diag);

    lastLastCost = lastCost;
    lastCost = cost;

  } else {

    // We are receiving the next short sequence
    if (position == shortReadIndex) {
      // This is the next base we should match against
      currentBase = upstream.currentBase;
      lastCost = (shortReadIndex + 1) * Alpha;
      lastLastCost = lastCost + Alpha;
    }
  }

  lastBase = upstream.currentBase;
  position = (position + 1) % (shortReadLength + referenceLength);
}
```

To build our matcher we must assemble a pipeline of PE filters whose length is determined by the length of the short-read sequences. We can express this as follows:

```
class Matcher : Pipeline<State, State> {
  public Matcher(uint shortReadLength, uint referenceLength) {
    for (uint i = 0; i < shortReadLength; ++i) {
      Add(new PE(i, shortReadLength, referenceLength));
    }
  }
}
```

The output from the final processing element in the pipeline will give us the (implicit) positions and costs we want. However, most locations in the reference sequence will not match against a given short-read sequence without a lot of transformation and so we are only interested in match positions whose match costs are below some threshold. This implies we need to define a threshold filter to prune the output stream. We are only interested in the cost of each match below this threshold, and the offset in the reference sequence where this match occurred. So our first step is to define a new type to hold this information.

```
public struct Match {
  public readonly uint offset;
  public readonly MatchCost cost;

  public Match(uint offset, MatchCost cost) {
    this.offset = offset;
    this.cost = cost;
  }

  public static readonly Match Terminator = new Match(uint.MaxValue, 0);
}
```

The threshold filter takes as input a stream of `State` values and produce a stream of `Match` values. We need to know when we transition from processing one short read sequence to the next one. We identify this point by outputting a terminator match; this is simply a distinguished match value where the offset is set to `uint.MaxValue`. The threshold filter is not synchronous as the output rate varies. We indicate this by specifying the push rate as `upto(1)`. We also use a `valid` flag so we can ignore the initial output from the matcher. Putting this all together we end up with the following definition for the filter.

```
public class Threshold : Filter<State, Match> {
  private readonly MatchCost threshold;
  private readonly uint shortReadLength;
  private readonly uint totalLength;
  private uint position;
  private bool valid = false;

  public Threshold(MatchCost threshold, uint shortReadLength, uint referenceLength)
    : base(pop: 1, push: upto(1)) {

    this.threshold = threshold;
    this.shortReadLength = shortReadLength;
    this.totalLength = shortReadLength + referenceLength;
    this.position = referenceLength;
  }

  public override void Work() {
    State b = Pop();

    if (valid) {
      if (position == 0) {
        Push(Match.Terminator);
      } else if (position > shortReadLength) {
        var referenceCharPosition = position - shortReadLength;
        if (b.cost <= threshold && referenceCharPosition >= (shortReadLength - 1)) {
          var estimatedStartCharPosition = referenceCharPosition-(shortReadLength-1);
          Push(new Match(estimatedStartCharPosition, b.cost));
        }
      }
    }

    if (position == totalLength - 1) {
      position = (position + 1) % totalLength;
      valid = true;
    } else
      position++;
  }
}
```

The output from the threshold filter will tend to occur in short bursts, contiguous sequences of possible matches. We want to find the best match for each of these bursts and so we define the `LocalMinimum` filter. In addition to selecting the best match in each burst it will also limit the maximum number of matches that will be reported for each short-read sequence. This is not essential when considering a single matcher. However, when we consider running multiple matchers in parallel it will simplify the buffering requirements, as we shall see later.

```
public class LocalMinimum : Filter<Match, Match> {
  private readonly uint gap;
  private readonly uint maxMatches;

  private uint lastOffset;
  private MatchCost localMinimum;
  private uint localMinimumOffset;
  private uint matches = 0;

  public LocalMinimum(uint gap, uint maxMatches = uint.MaxValue)
    : base(pop: 1, push: upto(2)) {
    this.gap = gap;
    this.maxMatches = maxMatches;
    this.lastOffset = uint.MaxValue;
  }

  public override void Work() {
    Match m = Pop();
    var currentOffset = m.offset;
    var currentCost = m.cost;

    if (currentOffset == Match.Terminator.offset) {
      if (matches > 0) // Pending minimum
        Push(new Match(localMinimumOffset, localMinimum));
      Push(Match.Terminator);
      matches = 0;

    } else if (lastOffset == Match.Terminator.offset) { // Starting first match region
        localMinimum = currentCost;
        localMinimumOffset = currentOffset;

    } else if ((currentOffset - lastOffset) > gap) { // Starting new match region
      if (matches < maxMatches) {
        Push(new Match(localMinimumOffset, localMinimum));
        matches++;
        localMinimum = currentCost;
        localMinimumOffset = currentOffset;
      }

    } else if (matches <= maxMatches && m.cost < localMinimum) { // Better minimum
      localMinimum = currentCost;
      localMinimumOffset = currentOffset;
    }

    lastOffset = currentOffset;
  }
}
```

We can put all these elements together to produce a `ShortSequenceMatcher`. Note the first filter that just converts a base into a `State` value.

```
public class ShortReadMatcher : Pipeline<Base, Match> {
  public ShortReadMatcher(
      uint shortReadLength, uint referenceLength,
      MatchCost threshold, uint gap, uint maxMatches) {
    Add(StreamGraph.Filter((Base b) => new State(b, 0, 0)));
    Add(new Matcher(shortReadLength, referenceLength));
    Add(new Threshold(threshold, shortReadLength, referenceLength));
    Add(new LocalMinimum(gap, maxMatches));
  }
}
```

Before proceeding further, let's test our example. To do this we need to define a source filter that generates a stream of short-read sequences and then pass this through a filter that interleaves the reference sequence after each short-read sequence. A suitable Interleave filter is provided by the S# library. The details of the source filter are unimportant.

```
public class Source : Filter<VOID, Base> {
  public Source(…) : base(push: 1) { … }
  public override void Work() { … }
}
```

We also need to examine the results so we need a sink filter that consumes the sequence of `Match` values and displays them in some fashion.

```
public class Sink : Filter<Match, VOID> {
  public Sink(…) : base(pop: 1) { … }
  public override void Work() { … }
}
```

We need to put the source, interleave, matcher and sink in a pipeline:

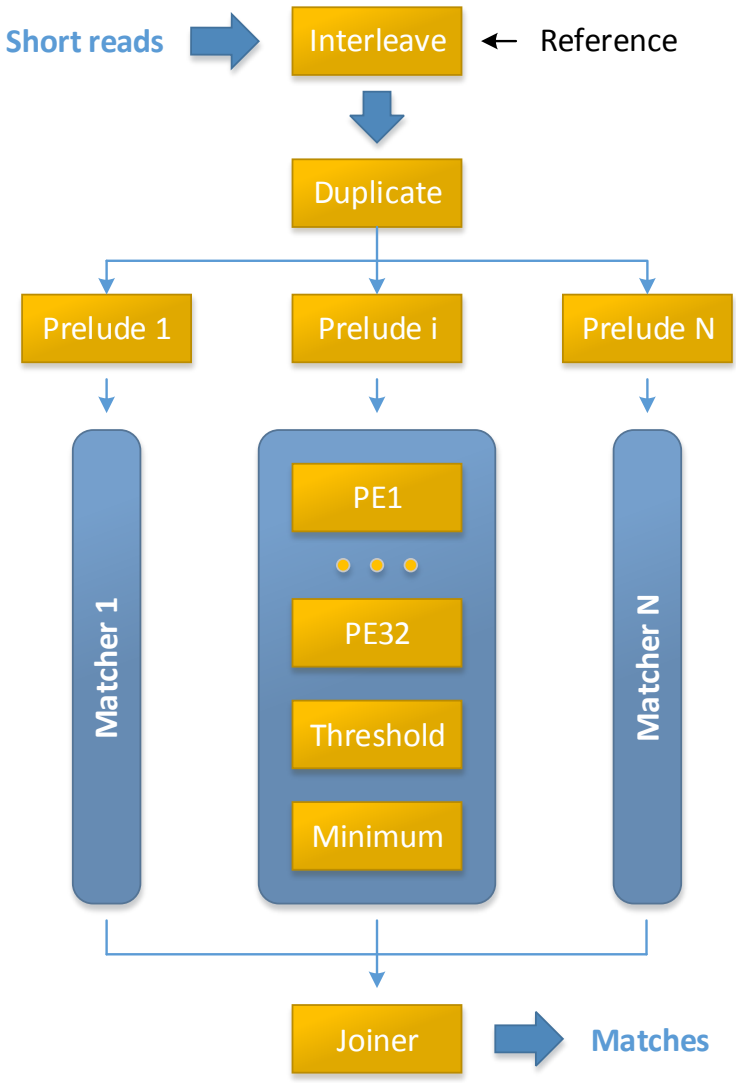
```
public class ShortReadTest : Pipeline<VOID, VOID> {
  public ShortReadTest() {
    Add(new Source(…));
    Add(new Interleave<Base>(<short-read length>, <reference sequence>));
    Add(new ShortReadMatcher(32, 15279296, 8, 2, 10));
    Add(new Sink(…));
  }
}
```

To execute this code in C# we build a stream processor from an instance of this graph and then execute it.

```
class Program {
  static void Main(string[] args) {
    var graph = new ShortReadTest();
    StreamProcessor sp = new StreamProcessor(graph);
    sp.Execute();
    Thread.Sleep(200000);
  }
}
```

The result should be a sequence of matches that continues until either the source filter stops feeding data to the graph, or the top-level thread stops sleeping and terminates the program. We could also wait until the graph itself terminated the execution, avoiding the need to sleep.

Where we have parallel resources available we can potentially match multiple short sequences against the reference sequence at the same time. On an FPGA we can choose the number of matchers so we fill up the available space on the FPGA. Let's assume we have *N* parallel matchers. We now assume the input will contain *N* short-read sequences followed by the reference sequence, with this pattern repeating. We can use a split/join to process this stream in parallel. The splitter will just duplicate the input to each branch. We can't just use a short sequence matcher in the branch as it will expect to receive a single short sequence, but the branch will be fed *N* short sequences. We therefore define a prelude filter to prune out the short sequences we don't need.



The prelude filter is simple to define:

```
public class Prelude : Filter<Base, Base> {
  private readonly uint shortReadPreludeLength;
  private readonly uint shortReadStart;
  private readonly uint shortReadEnd;
  private readonly uint totalLength;

  private uint position = 0;

  public Prelude(
    uint branchIndex, uint numBranches, uint shortReadLength, uint referenceLength)
      : base(pop: 1, push: upto(1)) {

    shortReadStart = shortReadLength * branchIndex;
    shortReadEnd = shortReadStart + shortReadLength;
    shortReadPreludeLength = shortReadLength * numBranches;
    totalLength = shortReadPreludeLength + referenceLength;
  }

  public override void Work() {
    var popped = Pop();

    if ( position >= shortReadPreludeLength
    ||   (position >= shortReadStart && position < shortReadEnd)) {
      Push(popped);
    }

    position = (position + 1) % totalLength;
  }
}
```

To build the parallel matcher we need to use a split-join:

```
class ParallelMatcher : SplitJoin<Base, Match> {
  public ParallelMatcher(
      uint branches, uint maxShortReadLength, uint referenceLength,
      MatchCost threshold, uint gap, uint maxMatches) {

    Splitter = new DuplicateSplitter<Base>();

    for (uint i = 0; i < branches; ++i) {
      Add(
        new Pipeline<Base, Match>(
          new Prelude(i, branches, maxShortReadLength, referenceLength),
          new ShortReadMatcher(maxShortReadLength, referenceLength,
                               threshold, gap, maxMatches)));
    }

    Joiner = new MatchJoiner();
  }
}
```

The splitter is easy to understand. It simply takes each input and duplicates it to all the branches. Another common kind of splitter uses a round-robin strategy, passing the first *N* values to the first branch, then the next *M* values to the second, and so on. S# supports round-robin splitters and joiners. The joiner in this case is a bit more complicated. We want to receive matches from the first branch *until* we see the terminator, then start

receiving matches from the second branch, and continue this process in a round-robin fashion. Fortunately S# also supports such joiners. We simply have to specify the condition that should become true when we wish to move onto the next branch. The required behavior is encapsulated inside the `MatchJoiner` class:

```
public class MatchJoiner : JoinFromBranchUntil<Match> {
  public MatchJoiner()
    : base((m) => // Predicate for moving to next branch
          m.offset == Match.Terminator.offset) {
  }
}
```

We can test our program as before, by replacing the instance of the short-read matcher by the parallel matcher in our test harness. However, we will also need to modify the arguments to our interleave filter so it generates *N* short-read sequences every cycle.

If you run the code on a multicore machine you should see that multiple cores are used to perform the matching. Where a subgraph uses fixed push and pop rates the S# compiler will typically execute all the nodes in the subgraph within a single thread, as this yields the best performance in many cases. However, where there are variable rate links, as there are on the outputs of the threshold filters, this will result in the graph being decomposed into a number of subgraphs, each of which will be executed asynchronously. This is why we end up with the branches of our split join all executing in parallel.

Buffering

Ideally an S# developer should not have to worry about implementation details, focusing instead on the definitions of the filters, and the desired graph structure. In many cases we can get close to this ideal. For example, if our graphs have fixed rates for all the filters then we can compute a static schedule for executing such graphs. These schedules allow us to compute accurate buffer sizes for all the links within the graph. However, in some situations we cannot build a single execution schedule. This may be because we want to run multiple schedules in parallel, for performance reasons. Or we may be forced to partition the graph because some filters use variable rates. In such cases we must construct buffers to connect the components that will run asynchronously with respect to each other. The approach used is to construct a set of fixed-size blocks. The upstream node, the producer of the data, requests a block and then starts writing items to this block. When it is full passes the block downstream and requests a new block, waiting if no block is currently available. Similarly the downstream node, the consumer, will wait until a block is available, then process the items in the block, before freeing it when finished. Freed blocks can then be reused by the producer.

For each asynchronous link we therefore have two parameters we can tune, the size of each block, expressed as the number of items it can contain, and the number of blocks to allocate. In many cases the system can automatically choose values for these parameters without requiring human intervention because the choice only affects the performance of the algorithm, not its correctness. For example, consider a pair of statically-scheduled subgraphs connected by an asynchronous link. If we make the block size very small then the system will spend proportionately a lot of time synchronizing threads to request and release blocks. At the other extreme, if we make the blocks very large then we will reduce this overhead at the expense of increased latency.

In a small number of cases the sizes of these blocks must be chosen carefully to avoid deadlock. And, unfortunately, our example is one such case. Suppose we have two parallel matchers. The rate at which data is generated by the Threshold and LocalMinimum stages will vary depending on the reference sequence and short-read value. So in a pathological case we might have the first matcher generating no matches, and the second

matcher generating a lot. Now imagine that the buffers connecting these stages used large block sizes. The link between the first LocalMinimum filter and the joiner would need to fill a complete block before the joiner received any of the data. This may require processing a lot of short-read sequences. But the second matcher will need to be consuming short-reads at the same rate, and so the buffers connecting the LocalMinimum filter to the joiner in the second matcher will start to fill up. When that happens the link from the Threshold to LocalMinimum will also start to fill up in that matcher. This process will continue until all the links in that matcher are full. At that point this "back-pressure" will stop the splitter from accepting any more data and the system will deadlock.

The solution in this example is simple. We just have to make the Threshold and LocalMinimum filters pass on their results as soon as they become available, which we can achieve by making the block size on these links contain a single entry. Furthermore, as the rate that data is produced on these links is far less than the rate at which the main matching pipeline is operating, the inefficiency of using such small blocks sizes is unimportant in this case.

To alter the characteristics of an asynchronous buffer we can either add a property to the upstream filter or the downstream filter, whichever is more convenient. In our definition of ShortReadMatcher the resulting code would look like this:

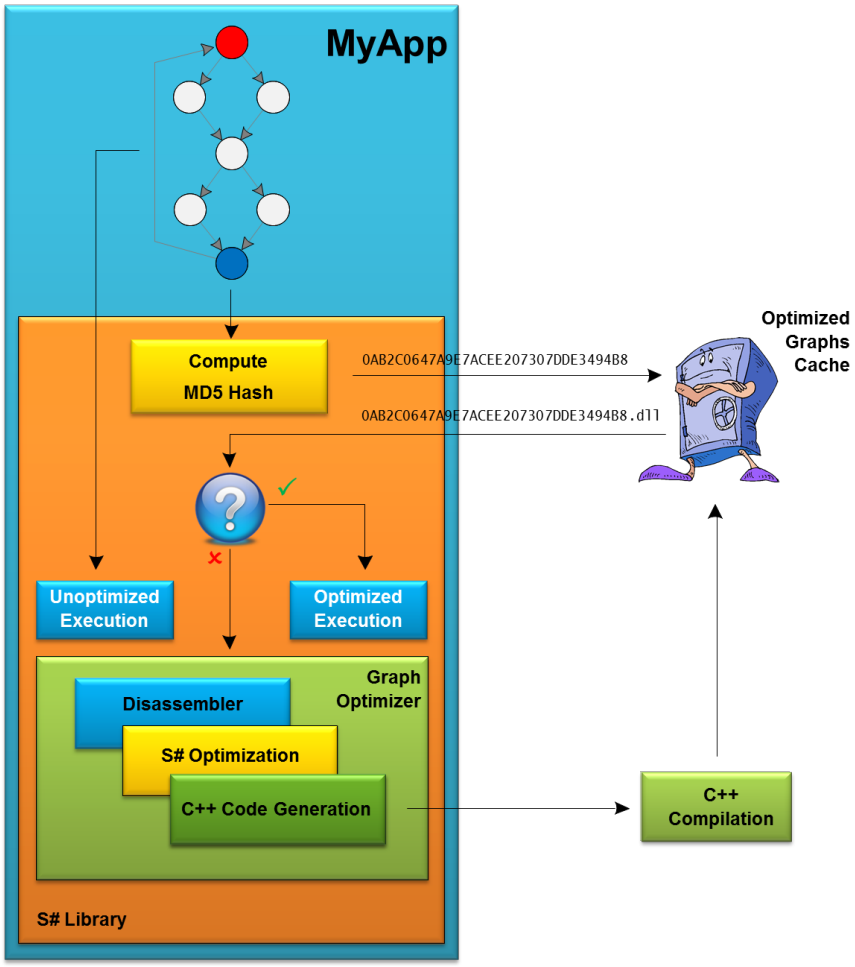
```
public class ShortReadMatcher : Pipeline<Base, Match> {
  public ShortReadMatcher(
      uint shortReadLength, uint referenceLength,
      MatchCost threshold, uint gap, uint maxMatches) {
    Add(StreamGraph.Filter((Base b) => new State(b, 0, 0)));
    Add(new Matcher(shortReadLength, referenceLength));
    Add(new Threshold(threshold, shortReadLength, referenceLength)
        { DownstreamProperties = {
            CompilationOptions.AsynchronousBuffer(numEntries: 1, maxBuffers: 100) }});
    Add(new LocalMinimum(gap, maxMatches)
        { DownstreamProperties = {
            CompilationOptions.AsynchronousBuffer(numEntries: 1, maxBuffers: 100) }});
  }
}
```

As you can see, specifying such properties is rather verbose at present. Fortunately it is rare to have to provide such hints, and the user can always define more concise helper functions if desired. Hopefully over time, as the system becomes smarter, there will be less need for such manual tuning.

Native Compilation

Whilst we can exploit multicore parallelism when executing a graph in C# the performance won't be that great for a variety of reasons. Although the C# compiler and runtime do an excellent job of executing C# programs, there is still some overhead compared to executing C/C++ code. Furthermore, when a subgraph contains components that communicate at fixed rates the S# compiler can compute an execution schedule. This defines the optimum order for executing each filter. However, there is some overhead in interpreting this schedule potentially millions of times per second, and it would be faster to compile such schedules directly into code. The S# framework allows us to take a graph and compile it to native code. Furthermore, we can then take this compiled code and use it as part of a larger graph. The runtime takes care of streaming the data from the C# graph to the native subgraph, and streaming the results back again.

If we needed to generate a native DLL every time we created a graph component from a stream graph then the performance would be very slow, even if we could guarantee the availability of a C++ compiler. Fortunately the graph component uses a caching mechanism to avoid this overhead. When a graph is made into a graph component we first compute a hash for the graph. A graph cache is consulted to see if this graph has been seen before. If it has then the DLL that was previously generated for this graph is simply loaded into the application, a fast operation. Of course a cache miss still requires a relatively expensive optimization and compilation step, but in many cases we can avoid this overhead completely at runtime by prepopulating the cache.



We would normally compile our compute-intensive graphs to C++, for execution on the same machine as our C# program. However, we can also compile our graphs to code that can run on an FPGA, either as a standalone device or as a compute accelerator for the PC. It is this capability we will explore in the rest of this paper.

To produce code for an FPGA we need to convert our parallel matcher into a graph component, specifying we want to target AutoESL. This is a high-level synthesis tool provided by Xilinx, the product now called VivadoHLS.

```
var pm = new ParallelMatcher(
               branches: 24, maxShortReadLength: 32, referenceLength: 15279296,
               threshold: 8, gap: 2, maxMatches: 10);
var component = new GraphComponent<Base, Match>(pm,
  new ProcessingOptions { CompilationOptions = new AutoESLCompilationOptions() } );
```

The process for compiling all graphs is similar. We build a graph component from the graph, and then create instances of the graph component within larger graphs. The construction of the graph component tells the system that this subgraph may be worth compiling, and tries to compile it if a compiled version does not already exist. We can specify different compilation options to choose different target platforms. In this example we've chosen the AutoESL backend.

When we execute this code the result is a collection of C++ files suitable for processing by the AutoESL HLS tool. This produces RTL code functionally equivalent to the C/C++ code, suitable for FPGA implementation. Just as with the C# and C++ versions, we need to provide some hints to the compiler to help determine the best buffer strategy. In this case we modify the Joiner to look like this:

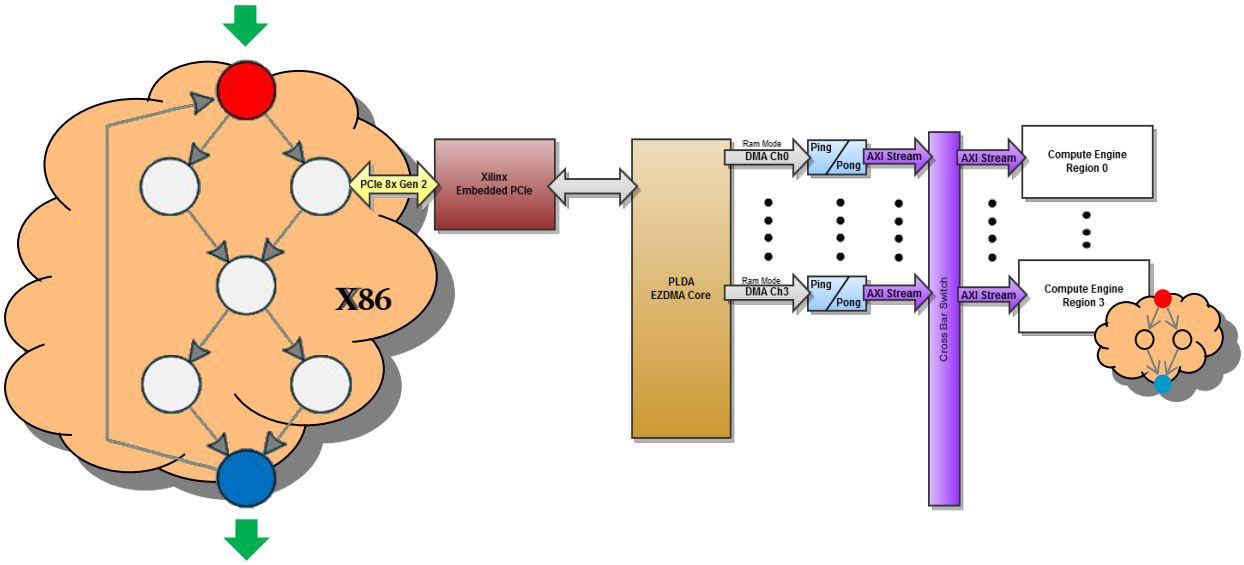
```
class ParallelMatcher : SplitJoin<Base, Match> {
  public ParallelMatcher(…) {
    …
    Joiner = new MatchJoiner()
             { Properties = { AutoESLCompilationOptions.InputFIFO((maxMatches+2) } };
  }
}
```

Note that the sizes of the short-read and reference sequences influence the performance of our design in different ways. The short-read sequences dictate the size of the matching pipeline. The longer the sequence the more hardware resources are needed to implement it and the fewer number of parallel matchers we can fit in the FPGA. The size of the reference sequence influences the performance in a different way. We need to stream the whole reference sequence through the graph each matching cycle. So the size of the reference sequence dictates the number of clock cycles required to perform a match.

Gran Turismo

At present the code generated from the AutoESL backend cannot be directly accessed from an S# application running on a PC. We need some code that can interface to the PCIe bus on the PC, marshaling data onto the bus and unmarshaling the results back again. Similarly on the FPGA side we need the opposite operations, reading data from the bus and feeding them to our design. The AutoESL compiler does not provide any assistance in this area, assuming that the generated code will be embedded within some larger design that handles interfacing with the outside world. In principle the S# compiler could generate the missing components. However, there is potentially a better solution.

The Gran Turismo system currently under development in MRL is building a framework for using FPGAs as compute accelerators. Partial reconfiguration effectively breaks one large FPGA into a number of smaller virtual devices, or sandboxes. If one of the sandboxes gets reconfigured, the rest of the chip "keeps on running". Each of these sandboxes exposes a standard interface built on AXI. This system will eventually not only allow for DMA transfers between the host and the sandboxes but also streaming data in between the sandboxes.

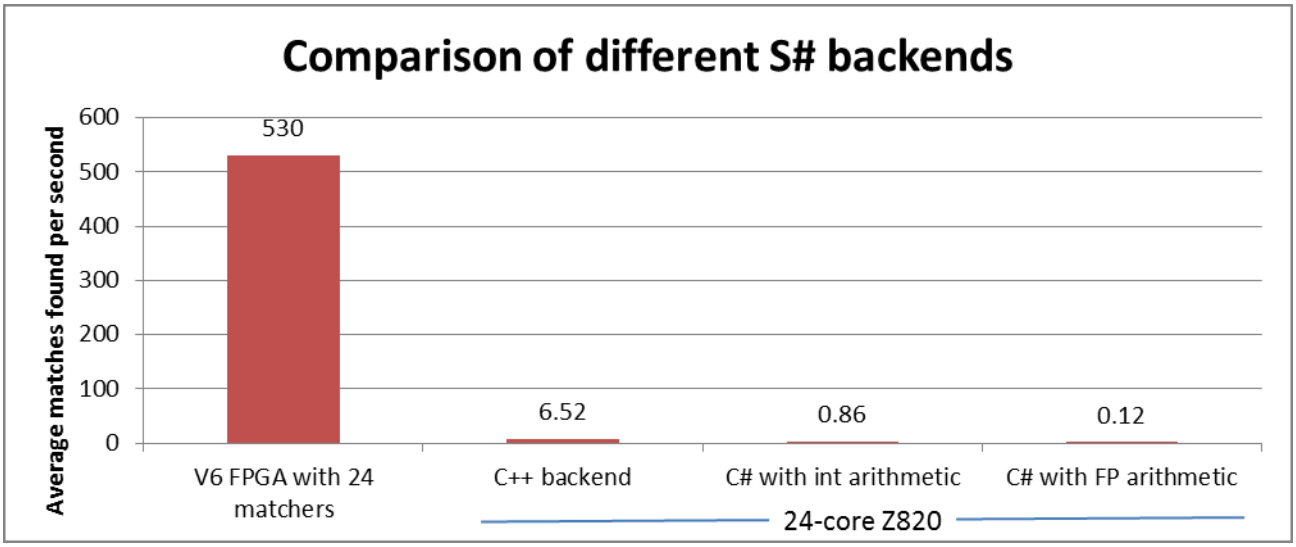


The S# framework automatically generates the scripts required to construct the FPGA bit files for each sandbox. The same caching mechanism described earlier is used to detect when an FPGA connected to the PC can accelerate a subgraph using such a bit file. The main difference from a users' perspective when using the FPGA backend is that generating a bit file is a lot more time-consuming than generating a DLL. When using the C++ backend in many cases it will be acceptable to pause the computation to build the DLL corresponding to a new graph. In the case of the FPGA backend this will never make sense as the build may take hours. We therefore start an asynchronous build and then continue the computation using the C# version of the graph, or potentially a C++ version. When we rerun the application after the bit files have been constructed the program will then find the file in the graph cache and will use this version of the graph to accelerate the computation.

At present the GT framework does not allow us to access any DRAM on an FPGA board, and so we need to stream the reference sequence from the PC to the FPGA board repeatedly, once for every *N* short read sequences. Although in principle PCIe can stream this data as fast as it could be accessed from DRAM, this solution is not ideal in general, particularly if we wanted to use multiple FPGA boards to accelerate matching even more. Hopefully Gran Turismo will eventually allow applications to access the DRAM. In that scenario the PC would just have to feed the short read sequences to the FPGA during the steady-state processing phase, drastically reducing the load on the bus between the PC and FPGA. However, for this particular example it turns out that the transfer rates between the PC and the FPGA are relatively low, and are not the bottleneck. This implies the lack of DRAM support is less of an issue for this example, although it still feels slightly inelegant to have to repeatedly transfer constant data.

Performance

To give the reader some indication of the performance of the system we used some test data obtained by sequencing C. elegans. The reference sequence consisted of 15279296 bases and the short reads were each 32 bases in length. Each short read typically matched the reference sequence in a small number of locations, on average two, given our choices for cost threshold and gap width. Some short reads matched at no locations, and some matched in more locations, although we limited the maximum number of matches that could be reported for each short read. The following graph shows how the performance varies with each backend.



The third column shows that the C# implementation generates less than one match a second when using ints as the match cost. We can run the example in C# using the fixed-point representation of the costs, but this runs substantially slower as the matching pipelines perform a lot of arithmetic using these costs. In both cases the number of parallel matchers was chosen to keep all 24 cores occupied on the test machine. The second column shows how using the C++ backend we can increase the matching rate by a useful amount. In this case we also choose the number of parallel matchers to keep all the cores busy. All these figures underrepresent what is achievable using these backends as we have spent relatively little time optimizing the analysis and code-generators in these components.

The interesting column is clearly the first one. It demonstrates that it is possible to generate a substantial increase in performance over the alternatives, and yet requires minimal knowledge of FPGA tools; most of the messy details are hidden within the S# framework. The figures were obtained using a Virtex 6 running in a HiTech Global board. We predict that using a more modern platform, such as a VC707 containing a Virtex 7, the matching rate would more than double.

Comparing implementations of systolic algorithms such as Smith-Waterman is often tricky. One measure that is frequently used is the number of cell updates per second (CUPS). For our example we have 32 cells in each matcher, the PE elements. And there are 24 matchers, resulting in 768 cells. We can stream in roughly 10 reference sequences per second, or 15.279 x 10^7 bases per second. Multiplying by 768 we end up with a processing rate of 117 GCUPS. These figures appear to be very competitive with other approaches. For example, a hand-crafted implementation of the algorithm on a GPU, albeit using a slightly more sophisticated matching function, produced a rate of 29.5 GCUPS, and required a lot more development effort.

Conclusions

The algorithm developed in this white paper is not novel, and is probably too simplistic for real use. Furthermore, you could certainly get better performance by writing the algorithm by hand using a hardware description language such as Verilog. Systems such as MatLab can also be used for generating code that targets FPGAs. However, hopefully the example illustrates that for some problems, where you can trade off some performance for implementation time, starting with a high-level approach such as S# may be an advantage.

The example provides a good example of where Gran Turismo's partial reconfiguration mechanism can be useful. At present we assume all the short-read sequences have exactly the same length, and we build our matching pipeline to exploit this knowledge. Suppose we now have the situation where the lengths can vary within some bounds. One approach would be to extend our algorithm so the matching pipeline had the length of the longest match. We would then add code to "short-circuit" those processing elements not required for the current short-read.

However, there is another alternative that might be more attractive. Suppose we build a family of matchers, one for each short-read length. Generating such a family in S# would be trivial. If each matcher ran in its own sandbox we could then reconfigure each sandbox, on the fly, as we encountered each new short-read sequence. At present such an approach would not be efficient as the number of sandboxes currently supported by Gran Turismo is far fewer than the number of parallel matchers we are using, 24 in the case of the Virtex 6. However, even in this case it might be possible to build a hybrid, for example three sandboxes with each one running a set of parallel matchers, where the short-read length supported by each sandbox could vary. We'd simply have to preprocess the input sequence to "cluster" the short-reads based on their lengths.

The author would like to thank Mauro Berdondini for all his help in demystifying AutoESL, and debugging the code generated by the S# compiler. The Gran Turismo team, Tom Vandeplas and Zeger Hendrix, were also invaluable in helping to understand the Gran Turismo framework, and suggesting improvements to the code generated by the FPGA backend.

Refererences

[StreamIt] StreamIt: A Language for Streaming Applications. William Thies, Michal Karczmarek, Saman Amarasinghe. International Conference on Compiler Construction. Grenoble, France. Apr, 2002

[Stevens] P. McMahon, K. Stevens, H. Chen, T. Filiba, V. Nagpal and Y. Song. Parallel alignment of multiple short-read sequences against a reference genome on a reconfigurable computer cluster. Submitted. (2008).