Experience Report: Embedded, Parallel Computer-Vision with a Functional DSL

Ryan R. Newton
MIT CSAIL, Cambridge, MA, USA
newton@csail.mit.edu

Teresa Ko
UCLA Vision Lab, Los Angeles, CA, USA
tko@cs.ucla.edu

Abstract

This paper presents our experience using a domain-specific functional language, WaveScript, to build embedded sensing applications used in scientific research. We focus on a recent computer-vision application for detecting birds in their natural environment. The application was ported from a prototype in C++. In reimplementing the application, we gained a much cleaner factoring of its functionality (through higher-order functions and better interfaces to libraries) and a near-linear parallel speed-up with no additional effort. These benefits are offset by one substantial downside: the lack of familiarity with the language of the original vision researchers, who understandably tried to use the language in the familiar way they use C++ and thus ran into various problems.

Categories and Subject Descriptors:
D.3.2 Concurrent, distributed, and parallel languages; Applicative (functional) languages; Data-flow languages

General Terms: Design, Languages, Performance

Keywords: stream processing languages, computer vision

1. Introduction

A sensor network deployment typically involves a collaboration between domain experts and computer scientists (though the latter would ideally be optional). The domain experts are often programmers themselves, often building prototypes in Matlab or C++. The expertise in short supply is in *embedded software development*. Therefore, tools that make it easier to transition from prototypes to embedded code are of great value.

Functional programming is not known for its use in embedded programming, to say the least. But following the recent trend in two-stage domain-specific languages (DSLs) (3; 7; 5), we find that a stream-processing DSL can retain the software engineering benefits of functional programming (in the *metaprogram*), while generating good embedded code, exploiting parallelism, and partitioning programs transparently across embedded devices and more powerful "servers". In this paper we describe our experience using the *WaveScript* language to implement the latest version of our computer vision application for sensor networks. This paper is not about the WaveScript implementation, but rather the software engineering impact of specific language features on the implementation of our application, namely: (1) *multi-stage programming*; (2) *higher-order functions*; (3) *parametric and ad-hoc polymorphism*; and (4) *shared-nothing message-passing parallelism*. But first we need to describe the application itself.

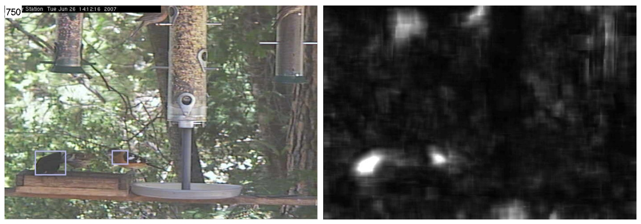

Figure 1. Example background subtraction results.

2. James Reserve Vision Application

A number of pertinent questions about the impact of climate change on our ecosystem are most readily answered by visually monitoring fine-scale interactions between animals, plants, and their environment. For example, species distribution, feeding habits, and timing of plant blooming events are often best observed through visual sensing. Some quantities, such as CO_2 intake of plants, have no "in the wild" sensor and can only be captured through visual sensing.

In this paper, we focus on detecting birds at a feeder station in the wild with a network of cameras. Bird populations are particularly informative about changes in the ecosystem, as species distributions can quickly change due to their mobility. The camera infrastructure used is part of the James Reserve Wildlife Observatory. A feeder station was constructed and equipped with a webcam and server. It captures a frame a second at 704x480 pixels.

There is inherent pressure to increase a camera's coverage at the cost of reducing the size of the objects of interest in the image, thereby creating a more challenging detection and recognition task. Similarly, increasing temporal coverage (battery lifetime) pushes for lower sampling rates, limiting the applicable methods. The resulting image sequence will inevitably have small birds with little features to distinguish them from one another or from the background, and instances of the same bird being in a completely different location in consecutive frames.

Our vision system is able to identify instances of a single bird in spite of these challenges. The system consists of two major components: background subtraction and bird classification. The case study in this paper will focus on the background subtraction component, because it is both computationally intensive and a substantial improvement over the state of the art in this domain.

2.1 Background on Background

Natural environments such as the forest canopy present an extreme challenge to background subtraction because the foreground objects, by necessity, blend with the background, and the background itself changes due to the motion of the foliage and the rapid transition between light and shadow. For instance, images of birds at a feeder station exhibit a larger per-pixel variance due to changes in the background than due to the presence of a bird. Rapid background adaptation fails because birds, when present, are often moving less than the background and often end up being incorporated into it.

Our background subtraction approach is based on building a model of the colors seen in the neighborhood around each pixel and then computing the difference between each new pixel value and the historical model for that pixel's neighborhood. Therefore, the algorithm must maintain state for each pixel in the image (its model) and traverse each input image, comparing each new pixel against the model, and updating the model based on the values of surrounding pixels. An example result can be seen in Figure 1.

The background model for the pixel located at the *i*th row and *j*th column is in general a non-parametric density estimate, denoted by $p_{ij}(x)$. The feature vector, $x \in \mathbb{R}^3$, is a colorspace representation of the pixel value. For computational reasons, we consider the simplest estimate, given by the histogram

$$p_{ij}(x) = \frac{1}{|S|} \sum_{s \in S} \delta(s - x), \tag{1}$$

where S, the set of pixel values contributing to the estimate, is defined as

$$S = \{x_t(a,b) \mid |a-i| < C, |b-j| < C, 0 \leq t < T\}, \tag{2}$$

where $x_t(a,b)$ is the colorspace representation of the pixel at the *a*th row and *b*th column of the image taken at time *t*. The feature vector, x, is quantized to better approximate the true density.

To detect foreground at time τ, a distribution, $q_{ij,\tau}(x)$, is similarly computed for the pixel located in the ith row and jth column using only the image at time τ according to

$$q_{ij,\tau}(x) = \frac{1}{|S_\tau|} \sum_{s \in S_\tau} \delta(s - x), \tag{3}$$

where S_τ, the set of pixel values contributing to the estimate is defined as

$$S_\tau = \{x_\tau(a,b) \mid |a-i| < C, |b-j| < C\}, \tag{4}$$

The Bhattacharyya distance between $q_{ij,\tau}(x)$ and the corresponding background model distribution for that location, $p_{ij,\tau-1}(x)$, calculated from the previous frames, is computed to determine the foreground/background labeling. The Bhattacharyya distance between two distributions is given by

$$d = \int_X \sqrt{p_{ij,\tau-1}(x) q_{ij,\tau}(x)} dx, \tag{5}$$

where X is the range of valid x's and d ranges from 0 to 1. Larger values imply greater similarity in the distribution. A threshold on the computed distance, d, is used to distinguish between foreground and background. While subtle, this combination of background model and classifier allows for large articulated movements in the background to be ignored and small foreground objects to be detected.

3. WaveScript, Briefly

A WaveScript program constructs a dataflow graph of stream operators that executes in a non-synchronous (event-driven) manner. Each operator consists of a work function and optional private state.

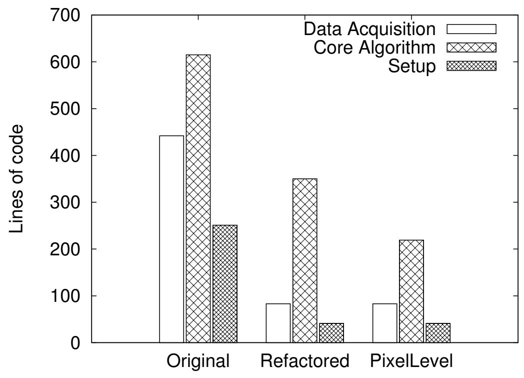

Figure 2. Background Subtraction: Lines of code.

Each work function is an imperative routine that processes a single stream element, updates the private state for that dataflow operator, and produces elements on output streams. The job of the WaveScript front-end is to partially evaluate the source (meta) program to create the dataflow graph, whereas the WaveScript backend performs graph optimizations, profiles and partitions graphs across devices (6), and reduces work functions to an intermediate language that can be fed to a number of backend code generators.

The final intermediate language for work functions is a monomorphic first-order language that is easily retargetable to any platform that has a C-compiler (and many that don't). WaveScript currently supports many embedded platforms including TinyOS "motes", smartphones running JavaME, iPhones, and embedded Linux devices such as routers. WaveScript is used for embedded sensing applications that involve digital signal processing together with more irregular event processing. For example it has been used for acoustic localization of wild animals (2) and detection of potholes with sensor-equipped taxicabs (4).

WaveScript itself is essentially an ML-dialect with a C-like syntax, a special form for accessing first-class streams, and miscellaneous extensions (e.g., extensible records). A top-level source program returns a stream value. Timers and drivers for hardware sensors provide stream sources, and a pair of primitives are the sole means of processing streams: *merge* combines streams in real-time order of arrival, and *iterate* is a "for-each" style construct whose evaluation creates a new dataflow operator and provides its work function, and whose return value is a new stream. The user manual contains details (1).

4. Implementation in WaveScript

The application was ported to WaveScript from a prototype in C++. Figure 2 shows a breakdown of how lines of code were spent in both the C++ and WaveScript versions of the application. The porting process consisted of four steps:

1. Port code verbatim to WaveScript.
2. Factor duplicated code using higher-order functions.
3. Remove unnecessary floating point.
4. Parameterize design; expose parallelism.

The most interesting step is exposing parallelism. The algorithm is clearly data parallel. In fact, a separate process can compute each pixel's Bhattacharyya distance (and update each pixel's model) independently. But the data-access pattern is non-trivial. To update each pixel's model, each process must read an entire patch of pixels from the image around it. Thus, *tiling* the matrix and assigning tiles to worker threads is complicated by the fact that such tiles must

```
for r = 0 to rows-1 {
  // create the left most pixel's histogram from scratch
  c :: Int = 0;
  roEnd = r - offset + SizePatch;  // end of patch
  coEnd = c - offset + SizePatch;  // end of patch
  for ro = r-offset to roEnd-1 { // cover the row
    roi = if ro < 0 then -ro-1 else
          if ro >= rows then 2 * rows-1-ro else ro;
    for co = c-offset to coEnd-1 { // cover the col
      coi = if co < 0 then -co-1 else
            if co >= cols then 2 * cols-1-co else co;
      // get the pixel location
      i = (roi * cols + coi) * 3;
      // figure out which histogram bin:
      binB = (Int) ((Float)image[i  ] * inv_sizeBins1);
      binG = (Int) ((Float)image[i+1] * inv_sizeBins2);
      binR = (Int) ((Float)image[i+2] * inv_sizeBins3);
      // add to temporary histogram
      tempHist[binB][binG][binR] += sampleWeight;
    }
  };
  // copy temp histogram to left most patch
  for cb = 0 to NumBins1-1 {
   for cg = 0 to NumBins2-1 {
     for cr = 0 to NumBins3-1 {
       bgHist[k][cb][cg][cr] += tempHist[cb][cg][cr];
  }}};
  // increment pixel index
  k += 1;
  // compute the top row of histograms
  for c = 1 to cols-1 {
    ...
    // Here: two more ro/co loops like above.
    // These add and subtract new data from the
    // histogram to update it incrementally.
    ...
  }}
```

Figure 3. An excerpt from the verbatim port.

overlap so each pixel may reach its neighbors. For these reasons, it is not straightforward to implement this algorithm in most stream processing languages. For example, stream processing languages tend to require that the input to each stream operator be a linear sequence of data. Exposing parallelism and exploiting locality in the background subtraction stage then requires an appropriate serialization of the matrix (for example, using Morton-order matrices), but this in turn creates complicated indexing. It is reasonable to say that the stream-processing paradigm is not a natural fit for parallel matrix computations. Yet it can be made to work using a high-level streaming language with a full datatypes (algebraic datatypes, dynamic allocation) and the additional power of meta-programming.

4.1 Porting Verbatim

Because WaveScript has imperative constructs and a C-like concrete syntax, it is straightforward to do a verbatim translation of C or C++ code. This does not in any way extract parallelism (it results in a dataflow graph with one operator). But it is the best way to establish correspondence with the output of the original program and then proceed by correctness preserving refactorings.

The original source code possessed substantial code duplication (Figure 3), having mainly to do with repeated processing of nested arrays, including index calculations. The code in Figure 3 is part of the `populateBg` function, which builds the initial background model for each pixel and takes as input storage space for the models and an input image. It has the following signature:

```
populateBg :: (Array4D Float, Image) -> ();
type Image = (RawImage * Int * Int); // With wid,height
type RawImage = Array Color;
type Array4D t = Array (Array (Array (Array t)));
```

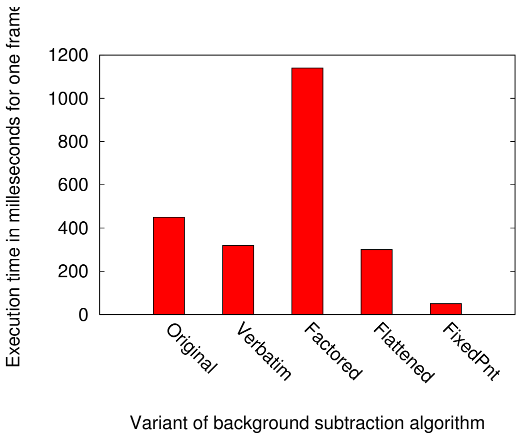

Figure 4. Single threaded performance of ported versions vs. original C++ version. Shows average time for processing each frame on a 3.2 gHz Xeon machine.

`populateBg` is called repeatedly with a series of `Images` to ready the background model before processing (classifying) new frames. In this version of the code, no significant type abstraction has yet been applied. The "model" for each pixel is a three-dimensional histogram in color space (the argument to `populateBg` is four dimensional to include a 3D histogram for each pixel in the image).

The background subtraction algorithm as a whole consists of 1300 lines of code containing three functions very much like `populateBg`. A second function updates the model for each new frame, and a third compares each new image with the existing model to compute Bhattacharyya distances for each pixel. These functions traverse both the input image and stored histograms. The performance of the original version, initial port, and subsequent refactorings is illustrated in Figure 4. Generally speaking, excluding automatic memory management, the object-language generated by WaveScript shares most of the characteristics of C code.

4.2 Refactoring Code

The next step was to simply clean up the code. Some of this consisted of factoring out simple first-order functions to capture repeated index calculations (a refactoring applied just as easily to the original C++). Other refactorings involved using higher order functions, for example, to encapsulate traversals over the image matrices and thereby remove `for`-loops and indexing expressions. After refactoring, the code was reduced to 400 lines. The clearer structure of the `populateBg` function can be seen in Figure 5. Both Figures 3 and 5 contain an optimization: the difference in histograms for neighboring pixels is small, and one can incrementally be computed from the other: adding some samples, removing others, and thereby "sliding" the patch. But this optimization has become much clearer in the structure of Figure 5.

In this version we have begun to abstract the types used. Rather than a 4D nested array to represent a 5D space (a matrix of histograms), we use the preexisting WaveScript 2D and 3D matrix libraries. These provide ADTs with multiple implementations, including a WaveScript native one and a Gnu Scientific Library wrapper (which uses BLAS). Not needing linear algebra, we use the former in this paper. Swapping in a flattened row-major representation results in fewer small objects and fewer pointer dereferences, creating the performance improvement shown in Figure 4. (Note that the initial "Factoring" damaged performance, possibly by obscuring backend compiler optimizations.) Also, it's a banal point, but parametric polymorphism for data types is critical. Most likely, the inconvenience of emulating this in C++ (templates) and the lack of

```
type PixelHist = Matrix3D Float;
populateBg :: (Matrix PixelHist, Image) -> ();
fun populateBg(bgHist, (image,cols,rows)) {
  // bgHist  : background histograms
  // image   : frame of video stream
  assert_eq("Image size:", length(image), rows*cols*3);
  tempHist = PixelHist:make(rows,cols);
  // Strong assumption about order of matrix traversal:
  Matrix:foreachi(bgHist, rows,cols,
    fun(r,c, bgHist_rc) {
      if c==0 then
           initPatch(r,c, rows,cols, tempHist, image)
      else shiftPatch(r,c, rows,cols, tempHist, image);
      // copy temp histogram to left most patch:
      Matrix3D:map_inplace2(bgHist_rc, tempHist, (+));
    })
}
```

Figure 5. The `populateBg` function builds background models (histograms) for the "Patch" centered around each pixel. First it creates a histogram for the leftmost pixel in a row. Then the next pixel's histogram is calculated incrementally by `shiftPatch`: (1) removing pixels in the left most col of the previous patch from the histogram and (2) adding pixels in the right most col of the current pixel's patch to the histogram. The `foreachi` function (as opposed to `foreach`) also passes indices for the data being accessed.

built-in matrix libraries resulted in the use of monomorphic, nested arrays in the original source.

4.3 Reducing Floating Point

One of our goals in porting this application was to run it on a wide range of embedded hardware as well as on multicore desktops (and partitioned between the two). In particular, we used Nokia smartphones (N95) with ARM processors lacking floating-point units. Thus, the penultimate step was to reduce the use of floating point calculations (for example, in calculating indices into the color histograms), replacing them with integer or fixed-point calculations. This results in a significant speedup even on desktop machines (Figure 4). WaveScript offered no special support for this refactoring. While it has what amounts to a built-in `Num` type class, which helps write reusable code, ideally there would be some tool support for this common problem: porting to fixed point, monitoring overflows and quantifying the loss of accuracy.

4.4 Exposing Parallelism, Design Parameterization

Finally, the most interesting part of this case study was using WaveScript to parallelize the application. Fortunately, refactoring for clarity (abstracting data types, matrix transforms) had gotten us most of the way. The essential change was to move away from code handling monolithic matrices (arrays) to expressing the transform locally on image tiles—we use *tile* to refer to a fixed submatrix of the image—and then finally at the level of the individual pixel (with the stipulation that a pixel transform must also access its local neighborhood). The end result was a reusable library for parallel matrix computations (see `parmatrix.ws`).

The first step is to implement transforms on tiles. From the perspective of the client code, this is the same as doing the transform on the original matrix (only smaller). The library code handles splitting matrices into (overlapping) tiles and disseminating those tiles to workers. The resulting dataflow graph structure is seen in Figure 6. The results of each independent worker are joined, combined into a single matrix, and passed downstream to the remaining computation. In Figure 7 we see the interface for building *tile kernels* via the function `tagged_tile_kernel`. Calling `tagged_tile_kernel`$(x, y, w, transform, init)$ will construct a stream transformer that splits matrices on its input stream into

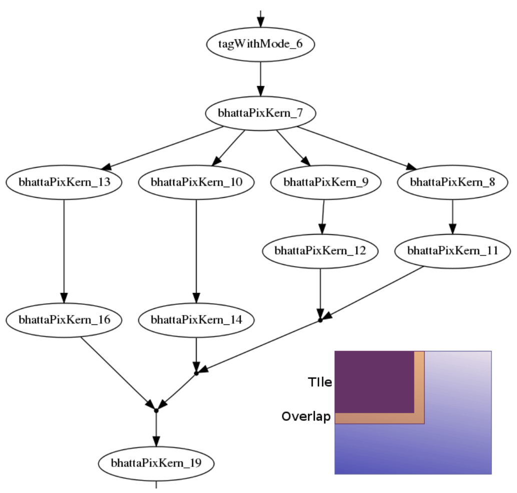

Figure 6. An example dataflow graph resulting from using a tile/pixel transform with $rows = cols = 2$ (generated by the compiler using AT&T GraphViz). Even though the pixels are partitioned into four disjoint tiles (dark purple), an extra margin must be included around each tile (orange) to ensure that each pixel within the tile may access its local neighborhood.

$x \times y$ tiles, with an overlap of w pixels in both dimensions. First, the $init$ function is called (at metaprogram evaluation) to initializes the mutable state for each worker. Then, at runtime, each tile is processed by the $transform$ function, producing a new tile.

The type signature for `tagged_tile_kernel` is listed in Figure 7. The "tagged" part is an additional complication introduced by the control-structure of the application. Because there is no shared state between kernels, all data must be passed through streams. Typical of stream-processing applications, there is a tension between dividing an application into finer-grained stream kernels and avoiding complicated data movement between kernels (for example, packing many configuration parameters into the stream)[1].

In the case of the background subtraction algorithm it is desirable to pass an extra piece of information (tag) from the original stream of matrices down to each individual tile- or pixel-level worker, which is exactly what the interface `tagged_tile_kernel` allows. For the background subtraction application, an earlier phase of processing determines what mode the computation is in (populating initial background model, or estimating foreground) and attaches a mode flag (boolean) on the stream of images.

From tiles to pixels: The next step in building the `parmatrix.ws` library was to wrap the tile-level operator to expose only a pixel-level transform. The modified interface is shown in Figure 8. Note that the result—a stream transformer on matrices—has the same type as the tile-level version. The core of the background subtraction algorithm, using the pixel-level interface, is shown in Figure 9. There is, however, one problem. When processing image data at the tile level, it is still possible to incrementally update histograms within a tile, albeit with decreased benefit as tiles get smaller. The pixel-level version based on the interface in Figure 8, however, cannot leverage this optimization. We will return to this issue in a moment.

In Figure 8, the `Nbrhood` type is used to represent the method for accessing the pixels in a local vicinity. It is a function mapping

[1] This is analogous to the sometimes awkward growth in the number of function arguments in purely functional programs, where function arguments are the sole means of communication between disparate program fragments. Of course this problem can be addressed by structuring techniques, for example, using a Reader monad.

```
tagged_tile_kernel ::
   // First, provide # X/Y workers and depth of
   // neighborhood access ("overlap") required:
   (Int, Int, Int,
     // Work function at each tile:
    ((tag, st, Tile px) -> Tile px2),
     // Per-tile state initializer:
    (Tile px) -> st)
  ->
    Stream (tag * Matrix px) -> Stream (Matrix px2);

type Tile t = (Matrix t * (Int * Int) * (Int * Int));
```

Figure 7. Signature for tile-level transform. This function creates X×Y workers, each of which handles a region of the input image. The transform is applied to each tile, and may maintain state between invocation, so long as that state is encapsulated in a mutable value passed as an argument to the transform. This assumes the `overlap` between tiles is the same in both dimensions. A tile is a piece of the original matrix together with metadata to tell where it came from; its fields are: (1) a matrix, (2) tile origin on original matrix, and (3) original image dimensions.

```
tagged_pixel_kernel_with_nbrhood ::
   (Int, Int, Int, // X,Y,overlap
    (tag, st, Nbrhood px) -> px2,
    // Per-pixel state initializer, takes indices:
    (Int, Int) -> st)
  ->
   Stream (tag * Matrix px) -> Stream (Matrix px2);

type Nbrhood a = (Int,Int) -> a;
```

Figure 8. Signature for pixel-level transform.

```
tagged_pixel_kernel_with_nbrhood (
  workersX, workersY, overlap,
  // This is the work function at each pixel.
  fun(bgEstimateMode, bghist, nbrhood) {
    if bgEstimateMode
    then populatePixHist(bghist, nbrhood);
    else estimateFgPix(bghist, nbrhood);
  },
  // Initialize per-pixel state; create a histogram:
  fun(i,j) Matrix3D:make(NumBins1, NumBins2, NumBins3, 0))
```

Figure 9. Background subtraction using a pixel-level transform.

(x, y) locations onto pixel values. At $(0, 0)$ the function gives the value of the center pixel (the one being transformed and updated). With this we are able to express the background subtraction application as a simple pixel-level transformation. The core of the implementation is shown in Figure 9. Given a boolean tag on the input stream (`bgEstimateMode`), and a state argument that contains the histogram for just that pixel (`bghist`), the kernel decides whether to populate the background (`populatePixHist`) or to estimate the foreground (`estimateFgPix`).

Restoring incremental histograms: The final step in porting our background subtraction algorithm was to reenable incremental computation of histograms in spite of the pixel-level interface to images. This is not difficult, but it does make the interface more complex (seen in Figure 10). Rather than direct access to neighboring pixel values, now the pixel kernel sees a sliding patch across the image (currently square, but could be generalized to a rectangle). The carried over result from the last position is used, together with the pixels sliding into view and the pixels sliding out, to compute the new result. The underlying implementation is still based on `tagged_tile_kernel`, and can only leverage incremental results within a tile.

```
tagged_pixel_kernel_sliding_nbrhood ::
   (Int, Int, Int,
     // Work function takes pixels in and pixels out.
     // 'carry' will represent the last computed result:
    (tag, st, carry, PixSet px, PixSet px) -> (px2, carry),
     // Per-pixel state initializer, takes indices:
    (Int, Int) -> st,
     // Function to compute the first carry:
    Nbrhood px -> carry)
  ->
   Stream (tag * Matrix px) -> Stream (Matrix px2);

type PixSet t = ((Int,Int,t) -> ()) -> ()
```

Figure 10. Signature for pixel-level transform supporting incremental computation of result. With `PixSets` we avoid constructing new sets in memory, rather we let the client "visit" the relevant pixels (and indices). Whole-program function inlining removes any performance penalty with this pattern.

5. WaveScript Learning Curve

The core background subtraction algorithm was ported by the first author, who is also the primary implementor of WaveScript, and naturally finds it easy to use. However, as other members of the UCLA group got involved and used WaveScript for other parts of the application, the retraining challenges became clear. There were two main lessons learned, having to do with **C-like syntax** and **quotation-free metaprogramming**.

First, WaveScript's syntactic similarity to C-family languages does reduce initial trepidation (even if perhaps it shouldn't). Certainly, several domain specific languages (e.g., Bluespec (7)) have ended up mimicking C or Verilog syntax for this reason. But we found that it also encouraged attempts to directly reuse inappropriate programming idioms. Setting aside basic misunderstandings of WaveScript constructs (for example, one programmer, unfamiliar with type inference, thought `type` declarations were necessary for assigning types to variables rather than defining new types), the major problem we found was with the use of mutable state. Of course, many C programmers use mutation by habit. WaveScript's support for mutable variables and arrays can encourage this. For example, programmers would often declare state globally and attempt to modify it within the work functions for stream operators:

```
myvar = 0;
S2 = iterate x in S1 { myvar++; ... }
```

Dangerously, this example actually *works*; the WaveScript evaluation model involves reifying a stream value back into code, and whatever state is found in the environment of the closure attached to a dataflow operator (work function) becomes the private state for that operator. However, sharing state between operators (the same mutable object reachable by two closures) is disallowed and will result in a compile-time error. The WaveScript design is predicated on the idea that understanding these meta-program evaluation failures is easier than understanding the error messages generated by a sufficiently sophisticated type system to rule out the errors. Nevertheless, it still helps to teach a "beginner" mode, where a special `state` keyword forces each operator's state to be declared in a restricted lexical scope:

```
S2 = iterate x in S1 {
       state { myvar = 0 }
       myvar++; ... }
```

The second major hurdle was understanding meta-programming in WaveScript. Again, WaveScript assumes that a simple model with some exceptions and corner cases is easier to learn than a more sophisticated one. Specifically, unlike more general meta-

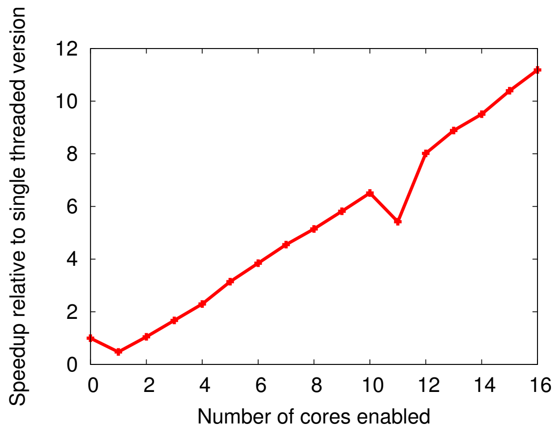

Figure 11. Parallel speedup: 16 data-parallel workers with variable number of enabled processor cores. Test platform was an AMD Barcelona with four quad-core processors. Cores were disabled using Linux's `/sys/devices/system/cpu` interface. The data point at 0 shows the single threaded speed—i.e. the program configured with only a single worker. The drop in performance at data-point 1 is due to the overhead of splitting and reassembling the matrix.

programming languages like MetaML (8), WaveScript does not use explicit quotation and anti-quotation constructs for creating "code values". Rather, the programmer is told that everything outside of an `iterate` will evaluate at compile-time. Nonetheless we saw frequent confusion about whether a data structure should be initialized at meta-program evaluation time, or at the beginning of runtime.

Also, there is the issue of distinguishing the capabilities of the meta- and object-languages. Like many other two-stage DSLs, WaveScript is an *asymmetric* meta-programming system, where the meta-language differs from the object-language. (For example, the object-language lacks closures.) Attempting to use meta-language features in the object-language results in a compile-time error (with code location). For example, one programmer tried to read configuration files at runtime, which is not possible if the code is running on an embedded platform such as a phone. One related problem had to do with the foreign-function interface (FFI), which can only be accessed at runtime. Programmers ran into difficulties trying to use the FFI before they had a firm grasp of the language. Some of these uses of the FFI were spurious, while others were an unfortunate consequence of the need to interface with hardware to get off the ground in sensing applications. Possible fixes would include banning the FFI in the aforementioned beginner mode, or restricting its use to strict idioms for data acquisition.

6. Results and Discussion

The end result of this project was a cleaned up implementation that also exposes parallelism. (And a reusable parallel matrix library!) Parallel speedups for the final version of the background subtraction algorithm are shown in Figure 5. These results are generated given a single, fixed 4×4 tiling of the matrix (16 tiles) that results in 16 stream operators ("workers") running on 16 threads. Another approach is to have the metaprogram set the tiling parameters based on the number of CPUs on the target machine. But this is complicated by the need to factor the target number of threads into separate x and y components such that $xy = numthreads$, which results in tiles of varying aspect ratios. Somewhat suprisingly, the operating system does a great job of juggling these 16 threads among a variable number of processor cores, with the exception of the unfortunate case at 11 cores. If the OS were doing poorly, we would expect to see an outlier at 16 cores where the mapping is one-to-one (and perhaps 8, 4, and 2).

Allowing the metaprogram to read the number of CPUs and determine a tile size is an example of the utility of the metaprogram as a separate phase that precedes the WaveScript compiler's own dataflow-graph scheduling and optimization. Still, it would be ideal to expose more of this tuning problem to the compiler itself. In the future, perhaps a method will be discovered to expose the compiler's own profile-driven optimization and auto-tuning process so that the programmer may delegate the tile-size decision.

This application also turned out to be a good candidate for distributed (inter-device) partitioning. After performing background subtraction most of the image is simply blacked out; with simple run-length encoding, these frames require much less network bandwidth when sent back to the server over a wireless network. Moreover, once the `parmatrix.ws` library is used, the space of choices becomes more continuous. Each tile-level worker can be placed on the embedded or server-side. Hosting half the workers results in half the data-reduction at half the cost, and, importantly, half the memory usage for expensive 3D histograms. For a detailed discussion of WaveScript's partitioning methodology, see (6).

Ultimately, while this application was greatly improved during its reimplementation, some of the interfaces we used represent a more imperative formulation than we would like—for example, kernels accepting a mutable state argument rather than producing a fresh state. A pure formulation would be ideal, but we are not currently able to achieve it with the near zero performance penalty that we require. Impure or not, abstracting control-flow was nevertheless valuable in this application.

References

[1] Wavescript users manual, http://regiment.us/wsman/.

[2] Michael Allen, Lewis Girod, Ryan Newton, Samuel Madden, Daniel T. Blumstein, and Deborah Estrin. Voxnet: An interactive, rapidly-deployable acoustic monitoring platform. In *IPSN '08: Information processing in sensor networks*, 2008.

[3] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage dsl embedded in haskell. *ICFP Experience Report*, pages 225–228, 2008.

[4] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 29–39, New York, NY, USA, 2008. ACM.

[5] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. *SIGPLAN Not.*, 43(9):335–346, 2008.

[6] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI'09: Networked Systems Design and Implementation*, 2009.

[7] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, June 2004.

[8] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.