# Butterflies in space: High-performance, FPGA-based signal processing on a satellite

Theodore S. Norvell
Department of Electrical and Computer Engineering
Memorial University of Newfoundland
theo@mun.ca

***Abstract*—The Killick-1 satellite is currently being designed to produce delay-Doppler maps of the earth. Because both bandwidth and energy are scarce, the signal processing needs to be done in real-time, on a small space craft, and using low power and little energy. Our solution was to use an FPGA and employ parallelism. This paper discusses our use of parallelism at three levels: At the highest level, task parallelism; in the middle, data-parallel processing; and at the lowest level, careful scheduling. The result is an implementation that meets the goals of the project on modest, low-power, hardware.**

CONTENTS



## I. INTRODUCTION

In the spring of 2021, with about $1\frac{1}{2}$ years to our planned launch date, no student team picked up the payload aspect of the project, leaving the project with a large hole. Much of the design of the digital processing fell to me. This paper is an experience report on the design and implementation of a particular signal processing algorithm under significant performance constraints. I'm going to focus on the algorithmic and parallelization aspects.

## II. DELAY-DOPPLER MAPS

Consider two stationary bats. One is chirping a pseudo-random tune. We can assume the tune has only two notes which we'll represent by $+1$ and $-1$ and that it is repeated as soon as it ends. The other bat is listening to the tune as it echos off of an insect flying toward the second bat. The insect will distort the melody in two ways. First, the melody will be delayed an amount of time that depends on the distance from the chirping bat to the insect and then to the listening bat. Second it will be sped up by an amount depending on the rate at which the length of that path is changing.

Suppose the listening bat multiplies the echo $u$ with a recording of the melody $v$,shifted by an amount $\tau$. If the recording and echo are not too out of phase, the product, $\lambda t \cdot u(t)v(t-\tau)$, should be a sine wave with frequency determined by the Doppler frequency associated with the rate of change of the path length. We can compare this function with a general sine wave of a given frequency $f$ to see if there is a correlation. For a given delay $\tau$, frequency $f$, and time period $(t_0, t_1)$, we can compute the correlation for that time period as[1]

$$\Upsilon(\tau, f, t_0, t_1) \triangleq \left| \int_{t_0}^{t_1} u(t)v(t-\tau)e^{-2\pi i f t}\,\mathrm{d}t \right|^2$$

We can repeat this a number $N$ of times over a period $(t_0, t_0 + N\Delta)$ to reduce the effect of noise:

$$DDM_{t_0,N,\Delta}(\tau, f) = \sum_{p=0}^{N-1} \Upsilon(\tau, f, t_0 + p\Delta, t_0 + (p+1)\Delta)$$

This function is a delay-Doppler map (DDM).

## III. THE KILLICK-1 MISSION PAYLOAD

The Killick-1 is a small satellite being designed at Memorial Unversity and C-Core largely by student teams and work-term students [2]. The mission of the satellite is to collect DDM images of the earth. We use GPS signals transmitted by satellites in medium-earth orbit. While a small amount of these signals hit antennas inside watches, cell phones, and sat nav systems, most of the signal bounces off the surface of the planet and back to space. The Killick-1 in low-earth orbit receives these reflected signals and processes them to make DDMs. We use the coarse-acquisition signals of the GPS sattelites. While all GPS satellites use the same frequency for this signal, each broadcasts a unique pseudo-random sequence of 1023, $+1$s and $-1$s over the course of $\frac{1}{32}$ ms and then repeats. There are 32 such sequences, known as Gold codes, and they have the properties of having low autocorrelation and

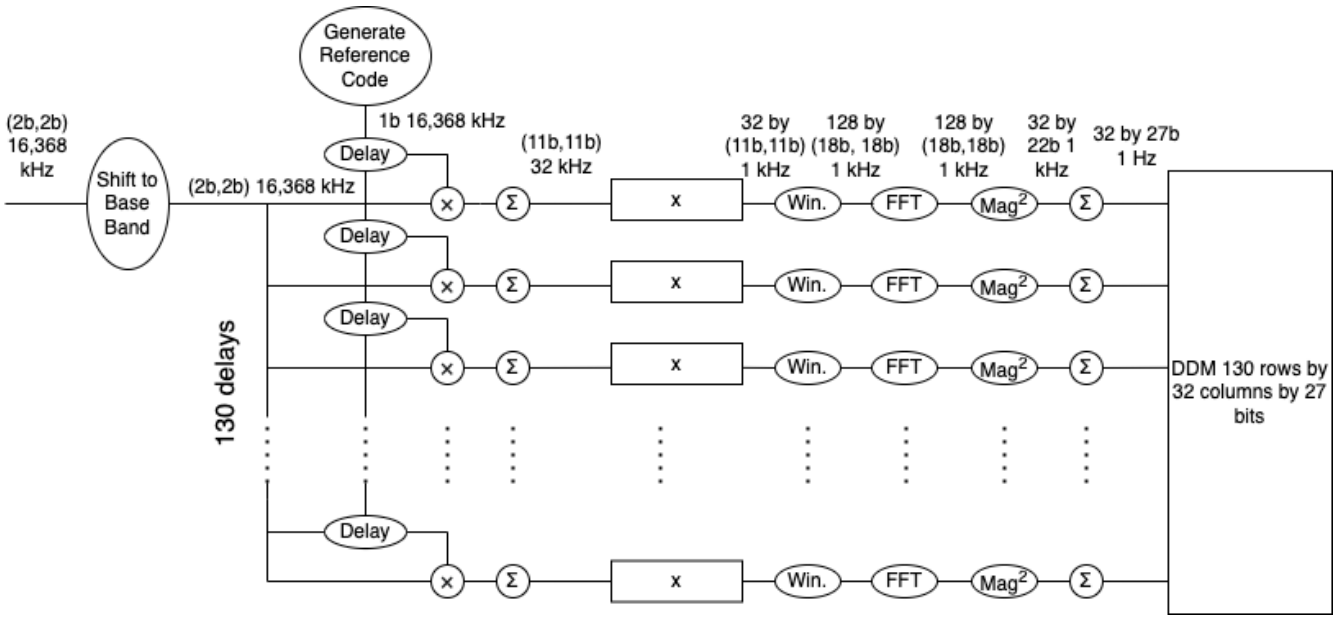


Fig. 1. Data flow of the algorithm

low correlation with each other. Thus we can 'tune' into one satellite by using its Gold code as a reference signal, much as the human auditory system can tune into one voice in a crowd.

We integrate using a numerical approximation

$$\Upsilon(\tau, f, t_0, t_1) \approxeq \left| \frac{1}{N_S} \sum_{j=0}^{N_S} u(t_0 + j\delta) v(t_0 + j\delta - \tau) e^{-2\pi i f (t_0 + j\delta)} \right|^2$$

where $\delta = (t_1 - t_0)/N_S$. We use $N_S = 32$, $\Delta = 1\,\mathrm{ms}$ and hence $\delta = \frac{1}{32}\,\mathrm{ms}$. We are interested in values of $f$ from 0 to 31 kHz with 1 kHz spacing. Let $f_k = k\,\mathrm{kHz}$, after some algebra we have

$$\Upsilon(\tau, f_k, t_0, t_1) \approx \left| \frac{1}{N_S} \mathrm{DFT}(x(t_0, \tau))_k \right|^2$$

where $x(t_0, \tau)$ is a vector of 32 elements with $x(t_0, \tau)_j = u(t_0 + j\delta) v(t_0 + j\delta - \tau)$. Thus we can compute the DDM as

$$DDM_{t_0, N, \Delta}(\tau, f_k) \approx \sum_{p=0}^{N-1} \left| \frac{1}{N_S} \mathrm{DFT}(x(t_0 + p\Delta, \tau))_k \right|^2$$

We need an estimate for $u(t)v(t-\tau)$ 32 times per millisecond for $N$ milliseconds starting at time $t_0$ and for each delay that we are interested in. We estimate each value $u(t)v(t-\tau)$ by summing 510 products in the vicinity of $t$.

We typically have an exposure time of 1s, so $N = 1000$. To make the row for one delay, we need to calculate $u(t)v(t-\tau)$ 510 times, summing the results, in $1/32\,\mathrm{ms}$ – This is repeated 32 times to produce a vector of 32 complex values every millisecond. We expand this vector to length 128 by appending 0s and apply Hamming windowing before doing a 128 point FFT. We select 32 values from this vector and square them to make a real vector of length 32. This process is repeated $N$ (typically 1000) times to produce $N$ vectors which are added together to produce final length real vector of length 32. By doing 128-point FFT, we get a frequency granularity of 250 Hz. The selection of 32 values reduces the size of the DDM while allowing one to zoom in on the frequencies of most interest.

As mentioned, the Gold code of 1023 values is replayed every millisecond, each value is called a chip and so there are 1.023 million chips per second. Each chip is represented by 16 bits, thus each bit period is about $61\,\mathrm{ns}$, coresponding to $16.368\mathrm{MHz}$. Our DDMs have 130 delays with a spacing of 2, 4, 8, or 16 bit periods. Thus the process of the paragraph above is repeated 130 times. Here is the algorithm. $F$ is a 130 by 32 real array in which the DDM is accumulated; the value of $\epsilon$ is one bit period; $W$ is a vector containing the first 32 values of a 128-point Hamming window; $g$ is a gain factor used to scale the values so that their sum will not overflow; and sel is a linear function that maps $\{0, ..32\}$ to $\{0, ..128\}$.

$F := \mathbf{0}$
for $p \in \{0, ..1000\}$
  for $r \in \{0, ..130\}$
    var $X$ : array $\{0, ..128\}$ of $\mathbb{C} := \overrightarrow{0}$
    for $j \in \{0, ..32\}$
      var $a : \mathbb{C} := 0$
      for $q \in \{0, ..510\}$
        $a := a + x(t_0 + p\Delta + j\delta + q\epsilon, \tau(r))$
      $X(j) := a \times W(j)$
    FFT( $X$ )
    for $k \in \{0, ..32\}$
      $F(r, k) := F(r, k) + g \times |X(\mathrm{sel}(k))|^2$

Figure 1 shows the data flow implied by this algorithm.

## IV. The hardware context

The Killick-1 bus is a roughly $10\,\mathrm{cm} \times 10\,\mathrm{cm} \times 20\,\mathrm{cm}$ box. Within that we have a stack of aproximatley $10\,\mathrm{cm} \times 10\,\mathrm{cm}$ printed circuit cards comprising a power system, an attitude determination and control system, a GPS receiver to obtain positions, a GPS receiver for making DDMs, a radio for communications, and a computer for controlling all functions of the satellite. The computer board chosen is an Abacus 2019 made by Gauss. This board contains an MSP 432 micro controller, some flash memory and, most importantly for this paper, a Xilinx Spartan 3E XC3S500E FPGA. The FPGA resources include

- 4,656 slices each containing
  - 2 flip-flops
  - 2 4-input LUTs
  - A few other gates and MUXs
- 20 dual-port memories, each holding 1024 18-bit words.
- 20 multipliers with 18 bit inputs and an 18 bit output

It runs with a $100\mathrm{MHz}$ clock for a $10\,\mathrm{ns}$ cycle time.

Each DDM requires over 64 megabits of input, while this could be stored and processed at a later time, the number of DDMs that could be produced in one pass over our target imaging area would be limited by storage considerations. Instead the decision was made to produce DDMs in real time.

Besides the FPGA, other important components of the paylod system include a patch antenna, and a custom board based on the MAX2769 GPS receiver from Maxim Integrated. This is configured to produce a four bit output at $16.368\,\mathrm{MHz}$. The output values represent values in the set

$$\{a + bi \mid a, b \in \{-3, -1, +1 + 3\}\}$$

After shifting this input signal to base band, we have our signal $u$.

## V. Algorithm design

The Abacus board with its small, dated FPGA were picked before the implementation of the DDM algorithm was given much consideration and so the issue of whether and how all calculations could be done in real-time became an important unanswered question.

In the spring of 2021, with about $1\frac{1}{2}$ years to our planned launch date this question loomed large. However no student team picked up the payload aspect of the project, leaving the project with a large hole. We decided to complete the payload using co-op students. Des Power of C-Core designed the high level algorithm presented above and I agreed to design the digital hardware to implement it with some guidance from Phillip Jales's thesis [3].

As can be seen, there is a lot of potential for parallelism. However limited hardware resources mean we need to be careful about how parallelism is used. The design exploits parallelism at three levels.

### A. Macrolevel parallelism: Task parallelism

To avoid storing the input, the calculation of $a$ needs to happen at the same rate as the bits are arriving from the receiver. This suggests it should be its own process. As a first step we split the computation into three phases: Phase 0, clears the F array. Phase 1 computes an array we call the $T$ array, which is 1000 pages by 130 rows by 32 columns of complex numbers one column at a time. Phase 2 does the windowing, FFTs, selection, magnitude calaculation, and adds to $F$ one row at a time. Once phase one has calculated a page, phase 2 can start processing it. This suggests that we represent 2 pages of the T array at a time, one being filled by the phase 1 process and the other being processed by the phase 2 process. Thus at a high level we have

$F := \mathbf{0}$
(
  for $p \in \{0, ..1000\}$ (Phase1($T(p, \_, \_)$)
‖
  for $p \in \{0, ..1000\}$ (Phase2($T(p, \_, \_)$))
)

Not shown is coordination so that, the Phase2 process does not begin work on a page until Phase1 has finished producing it. This scheme means that only two pages of $T$ need to be represented at once, provided Phase 2 can keep up. Unfortunately double buffering would use too much memory; we had room for 52 columns in TRep, our representation of $T$ rather than the 64 required. We ended up using what might be called 'fractional buffering'. In general column $j$ of page $p$ of $T$ is represented by column $(32 \times p + j) \bmod 52$ of TRep. Since we have only 20 excess columns, phase 2 must be almost complete before phase 1 starts to fill the 21st column. Each page takes $1\,\mathrm{ms}$ to fill, so phase 2 must process 130 rows in $0.625\,\mathrm{ms}$.

### B. Mesolevel parallelism: Data parallelism

Within phase 1, each delay is treated in parallel. We have 130 summator units to which the $u$ signal is distributed. The $v$ signal is passed through a chain of 130 delay units, each of which feeds a separate summator. Each summator is responsible for multiplying the $u$ and $v$ values —since the values on $v$ are either $+1$ or $-1$, this requires simple hardware— and accumulating the sum. This process is data-driven in that the summators and the delay chain react whenever a new value appears on the $u$ input. After 510 values have been accumulated, the sum is transferred to a holding register. The holding registers are then transferred to the appropriate column of the *TRep* array.

The TRep array is divided horizontally into 10 segments each holding 13 rows. Each segment is held in a different memory. Data is transferred from the holding registers to the *TRep* array sequentially for each segment of 13 summators.

In terms of memory space the each segment of the *TRep* array uses 3 bytes per entry. That makes 2028 of the 2048 bytes in one memory. So 10 of our 20 memories are devoted

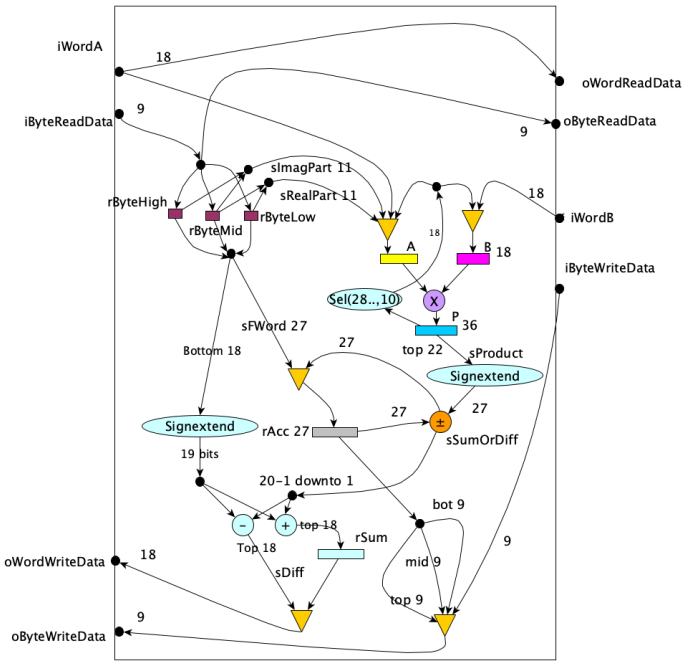


Fig. 2. The phase 2 data path

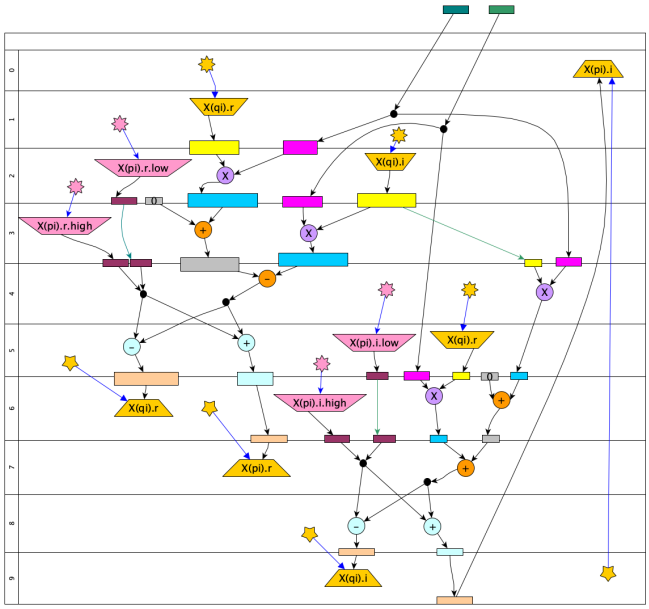


Fig. 3. Scheduling of the butterfly basic block

to the *TRep* array. The remaining 10 memories are used in phases 0 and 2. Phase 1 now looks something like this

par for $s \in \{0,..10\}$ , $r \in \{0,..13\}$
  for $p \in \{0,..1000\}$
    for $j \in \{0,..32\}$
      $a(s,r) := 0$
      for $q \in \{0,..510\}$
        $a(s,r) := a(s,r) +$
          $x(t_0 + p\Delta + j\delta + q\epsilon, \tau(r))$
      $h(s,r) := a(s,r)$
      signal *colDone*

||

par for $s \in \{0,..10\}$
  for $p \in \{0,..1000\}$
    for $j \in \{0,..32\}$
      wait for *colDone*
      for $r \in \{0,..13\}$
        $TRep(s,r,j) := h(s,r)$
    signal *pageDone*

Unlike phase 1, phase 2 requires a lot of memory operations; this greatly limits parallelism. We decided to use a sequential algorithm with data parallelism at the granularity of segments. The $F$ array is broken into 10 segments and each segment gets its own $X$ vector to use for input, output, and working memory of the FFT.

par for $s \in \{0,..10\}$
  for $p \in \{0,..1000\}$
    wait for *pageDone*
    for $r \in \{0,..13\}$
      for $j \in \{0,..32\}$
        $X(s,j) := T(p,r,j) \times W(j)$
      for $j \in \{32,..128\}$
        $X(s,j) := 0$
      FFT( $X(s,\_)$ )
      for $k \in \{0,..32\}$
        $FRep(r,k) := FRep(r,k) +$
          $g \times |X(s, \text{sel}(k))|^2$

For each segment the $X$ vector requires 256 18-bit words and the segment's portion of the FRep array requires 416 entries,each 3 9-bit bytes. The total makes 1,760 9-bits bytes per segment, taking up most of the remaining 10 memories. The remaining space is used to hold the 32 windowing constants and 33 constants needed by the FFT algorithm.

### C. *Microlevel parallelism: Scheduling basic blocks.*

To fit all the logic required for phase 2 in the space available, I designed one data path capable of doing the copy-and-window, the FFT, and the add-to-$F$ subphases. Since this data path is replicated 10 times, it needed to be small. The resulting data path is shown in Figure 2. It consists of one muliplier a few adders and a few registers.

Each inner loop body was carefully scheduled to execute in as few cycles as we could manage. This design was done concurrently with data path design. Here I'll only discuss the FFT inner loop body, known as a butterfly operation. In terms of complex operations the butterfly operation is

val $p := X(pi)$ ; val $r := X(qi) \times \omega(ks)$
$X(pi) := p + r$ ; $X(qi) := p - r$

Normally a 128-point FFT requires 7 rounds of 64 butterflies. However, as we know that the last 96 inputs are 0,

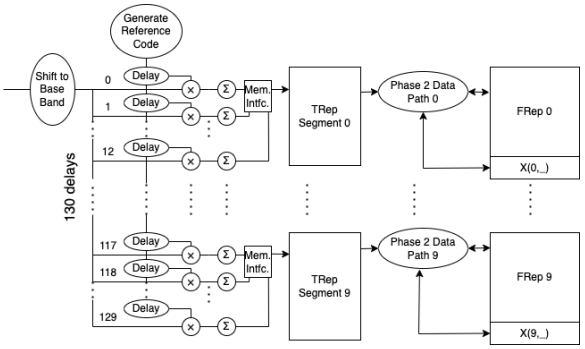


Fig. 4. Final HW design

the first 2 rounds consist only of copying numbers and can be combined with the copy and window subphase. Thus each FFT has 320 butterfly operations. In the end a 10 cycle butterfly proved fast enough. An entire FFT can be completed in 3,220 cycles. Adding in the other tasks, each iteration requires about 4,000 cycles, and 13 iterations about 52,000 cycles, which is well less than the 62,500 cycles that we have. The 10 cycle schedule for the butterfly operation is shown in Figure 3. Stars represent memory requests, trapezoids represent memory operations, rectangles represent registers.

Figure 4 shows an outline of the hardware design to implement our algorithm.

## VI. Roads not taken

We tend to present designs in their final form as if they were the first idea that came to us. This is rarely the case; design is an iterative process that interacts with implementation. With severe implementation constraints, design can not fully preceed implementation. For much of the project, our design assumed 20 segments each covering 6 or 7 rows, and each having one multiplier and one memory. The motivation was clear: since there are a lot of multiplications, use all the multipliers.

As it developed there were both space and time problems. Twenty data paths take up a lot of slices and this took more space than we had. The decision was made to cut the number of segments to 16 reducing the rows number of rows per DDM to 112. However there is still a time problem. With an $X$ vector, 7 rows of the *TRep* array, and 7 rows of the *FRep* array all in one memory, the *TRep* array could only have 40 columns. $40 - 32$ is $8$ and $8$ is one quarter of 32 meaning that phase 2 must complete in $0.25\,\mathrm{ms}$. An 8-cycle butterfly schedule met the time limit. But, on implementation, it turned out that 2 additions could not complete in 1 cycle as required by the schedule. Carry look-ahead logic might have solved that problem, but at the cost of a larger data path.

The solution came, as solutions so often do, in the shower. By having 10 segments instead of 20, we save $5,120$ bytes of space by only needing 10 $X$ vectors rather than 20. This space can be used to make the T array 12 columns wider, increasing the phase 2 time limit by a factor of $\frac{20}{8} = 2.5$. The number of rows per segment (minus 1) and hence the amount of work to be done in that time increases by slightly less than $\frac{12}{6} = 2.0$. The amount of work per unit of time changes by a factor of slightly less than $2.0/2.5 = 0.8$, which is enough to allow a change from 8 cycles to 10 for each butterfly. Less parallelism means less space, obviously, and, surprisingly, a significantly lower work/time requirement for the sequential algorithm. That decrease in required speed more than makes up for the decrease in parallelism.

## VII. Conclusion and thanks

Some lesson that can be learned include.

- Design is iterative and entwined with implementation.
- Keep the implementation flexible enough to accomodate late changes to the design. We had done that and changing the memory layout, giving each segment two memories, and cutting the number of segments was remarkably easy.
- Parallelism has hidden as well as obvious costs. Understanding these can be crucial.
- Careful consideration of parallelism at the algorithmic level is important.

This paper has focused on algorithmic design and components in the data flow. I haven't even touched on numerical considerations, on the address path —for creating memory addresses— and the control path —which for phase 2 consists of a state machine of about 70 states. All of this needs to be prototyped in software, and then expressed in synthesizable VHDL and thoroughly tested at the unit and system levels, which is a tremendous implementation effort especially for engineering students (and professors) learning both VHDL and the various simulation and synthesis tools.

I would like to thank Weimin Huang and Des Power for initiating the Killick-1 project, co-supervising the payload teams and much more. Co-op students Andrew O'Brien, Charles Smith, and Dustin Smith contributed greatly to the VHDL implementation.

## References

[1] D. Yang and F. Wang, *Multifunctional Operation and Application of GPS*. IntechOpen, 2018, ch. GNSS Application in Retrieving Sea Wind Speed, pp. 89–116.

[2] A. Quadri and G. Deveaux, "Preliminary design of the Killick-1 earth observation cubesat," in *Proceedings of NECEC*, 2019.

[3] P. Jales, "Spaceborne receiver design for scatterometric GNSS reflectometry," Ph.D. dissertation, University of Surrey, Jul. 2012.