FPGA Implementation of the Pixel Purity Index Algorithm

Dominique Lavenier[a], James Theiler[b], John Szymanski[b], Maya Gokhale[a], Janet Frigo[c]

[a]NIS-3: Space Data Systems
[b]NIS-2: Space and Remote Sensing Sciences
[c]NIS-4: Space Engineering

Los Alamos National Laboratory
Los Alamos, NM 87545, USA

ABSTRACT

The Pixel Purity Index (PPI) is an algorithm employed in remote sensing for analyzing hyperspectral images. Particularly for low-resolution imagery, a single pixel usually covers several different materials, and its observed spectrum is (to a good approximation) a linear combination of a few *pure* spectral shapes. The PPI algorithm tries to identify these pure spectra by assigning a pixel purity index to each pixel in the image; the spectra for those pixels with a high index value are candidates for basis elements in the image decomposition.

The PPI algorithm is extremely time consuming but is a good candidate for parallel hardware implementation due to its high volume of independent dot-product calculations. This article presents two parallel architectures we have developed and implemented on the Wildforce board. The first one is based on bit-serial arithmetic operators and the second deals with standard operators. Speed-up factors of up to 80 have been measured for these *hand-coded* architectures.

In addition,the second version has been *synthesized* with the Streams-C compiler. The compiler translates a high level algorithm expressed in a parallel C extension into synthesizable VHDL. This comparison provides an interesting way of estimating the tradeoff between a traditional approach which tailors the design to get optimal performance and a fully automatic approach which aims to generate a correct design in minimal time.

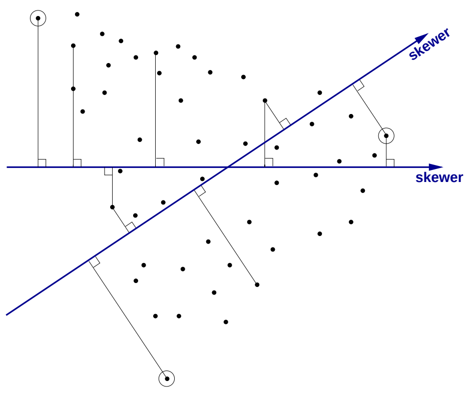
Keywords: Hyperspectral, Dot-Product, Pixel Purity, FPGA, Streams-C, High Level Synthesis

1. INTRODUCTION

Images produced for remote sensing applications are acquired simultaneously in several distinct spectral bands. Modern sensors can capture images with hundreds of spectral bands. These images are called *hyperspectral* and can be viewed as a data cube representing a succession of two-dimensional pixel planes. The challenge is to identify useful features in this huge volume of information.

A hyperspectral image is composed of *hyperpixels* which reflect the nature of the scene it represents. An assumption is that a scene contains relatively few distinct materials, and the hyperpixels (shorten to pixel in the following) are a *mixture* of these *pure* materials. The Pixel Purity Index (PPI) algorithm[1,2] aims to identify these *pure* pixels, so that all the remaining pixels can be expressed as a linear combination of them.

The algorithm proceeds by generating a large number of random D-dimensional vectors, called skewers, through the hyperspectral image (cf right fig.). For each skewer, every data point is projected onto the skewer, and the position along the skewer is noted. The data points which correspond to extrema in the direction of a skewer are identified, and placed on a list. As more skewers are generated, this list grows. The number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the most pure, and a pixel's count provides its pixel purity index.



The complexity of the PPI algorithm is in $O(K \times D \times N)$ with K the number of skewers, D the number of spectral bands and N the number of pixels. This is a very time consuming algorithm for large hyperspectral images. For instance, processing a single 512×614 224-band satellite image takes more than three hours on a conventional 450 MHz microprocessor. In that case, real-time analysis can only be achieved using specific architectures exploiting the potential of parallelism of this algorithm.

The PPI algorithm is a good candidate for hardware acceleration because it is readily parallelizable: the core skewer calculations (simple dot-products) can be done independently for each skewer and pixel of the image. Two architectures exploiting this high degree of parallelism and tailored to FPGA technology are presented. The first one uses bit-serial arithmetic allowing the design to hold a high number of small bit-serial multiplier/accumulator units. The second architecture[3] used optimized operators provided by the CAD FPGA tool.

Both architectures have been manually implemented and tested on the FPGA Wildforce board from Annapolis Microsystems, Inc. Also, automatic synthesis of the second version with the Streams-C compiler have been successfully carried out. Conventional non automated methods requires many steps starting from the VHDL specification to the final tests. This is usually a long and error-prone process, but it allows an experienced designer to get the best performance from the FPGA boards. On the other hand, an automatic approach reduces drastically the design time, but usually at the cost of not-quite-optimal performance. Our purpose in creating both manual and automated designs is to quantify productivity versus circuit performance.

The rest of the paper is organized as follows: the next section shows a possible way of parallelizing the PPI algorithm. Section 3 deals with the skewer data precision. Sections 4 and 5 present respectively the bit-serial and the optimized-operator architectures. Section 6 is devoted to the synthesis using the Streams-C compiler. Section 7 concludes this article.

2. PPI PARALLELIZATION

Most of the execution time of the PPI algorithm is spent in computing dot-products between the pixels and the skewers. These dot-product are highly independent and could be done simultaneously. This leads to many ways to parallelize the algorithm, but our approach targets the limited resources available on real FPGA boards. The sequential version of the Pixel Purity Index algorithm is:

```
PIXELS[N][D];                              // an image of N hyperpixels
SKEWER[K][D];                              // a set of K random vectors (skewers)
PPI[N];                                    // the PPI result

for (n=0; n < N; n++) PPI[n]=0;            // reset pixel purity index
for (k=0; k < K; k++)                      // for the K skewers
 {
   dpmax=MIN_INT; dpmin=MAX_INT;
   for (n=0; n < N; n++)                   // for the N pixels
    {
      dp = 0;
      for (d=0; d < D; d++)                // compute a Dot-Product
         dp = dp + SKEWERS[k][d]*PIXELS[n][d];
      if (dp > dpmax) { imax=n; dpmax=dp; }    // detect extrema
      if (dp < dpmin) { imin=n; dpmin=dp; }
    }
   PPI[imax]++;                            // update PPI
   PPI[imin]++;
 }
```

For each skewer, N dot-products are computed to determine the two pixels which produce the largest and the smallest dot-product. The pixel index (PPI vector) is modified accordingly. A pixel n is a candidate to be a pure pixel if PPI[n] has a high value.

From the above description it can easily be seen that all the dot-products can be computed independently: there are no dependencies between any of them. The parallelization takes advantage of this by computing $KS \times NS$ dot-products simultaneously, where KS and NS represent respectively the number of skewers and pixels which can be processed in parallel. Expressing the parallelism with `forall` statements, the PPI loop can be re-written as follows:

```
for (k=0; k < K; k=k+KS)
 {
   dpmax[] = MIN_INT; dpmin = MAX_INT;
   for (n=0; n < N; n=n+NS)
    {
      dp[][] = 0;
      forall (ks=0, ns=0; ks<KS, ns<NS; ks++, ns++)
       {
         dp[ks][ns] = DOT-PRODUCT (SKEWERS[k+ks], PIXELS[n+ns]);
       }
      forall (ks=0; ks<KS; ks++)
       {
        for (ns=0; ns<NS; ns++)
          {
            if (dp[ks][ns] > dpmax[k+ks])
              { imax[k+ks]=n+ns; dpmax[k+ks]=dp[ks][ns]; }
            if (dp[ks][ns] < dpmin[k+ks])
              { imin[k+ks]=n+ns; dpmin[k+ks]=dp[ks][ns]; }
          }
       }
    }
   for (ks=0; ks < KS; ks++)
    {
      PPI[imax[k+ks]]++;
      PPI[imin[k+ks]]++;
   }
 }
```

The first `forall` loop computes concurrently $KS \times NS$ dot-products. The second identifies the extrema. The two `forall` loops must be executed sequentially since a min/max computation requires the dot-product results already be computed.

In this description, the parallelization is limited to the computation of $KS \times NS$ dot-products simultaneously whereas the potential parallelism is $K \times N$. The reason is that we target real FPGA boards which have limited physical resources. In other words, we suppose that a FPGA board is only capable of executing $KS \times NS$ dot-products in parallel.

An important issue for getting maximum parallelism is to optimize the dot-product operation which is actually a series of multiplication/accumulations between pixels and skewers. Implementing a full multiplier into a FPGA component is not cheap in terms of resources. Hence, it is first worth to wonder if a full multiplication is absolutely needed. The next section discusses how the multiplication can be optimized by using skewer coefficient with limited precision.

3. LIMITED PRECISION SKEWER COEFFICIENTS

Although the PPI algorithm is defined as if the data and the skewers are composed of floating-point values, in practice both are of limited precision. The data is provided in digital format with fixed precision, but the precision of the skewer coefficients is up to the designer of the algorithm. The fewer bits of precision used by the skewer coefficients, the smaller area of the FPGA will be needed for multiplication, and therefore the larger number of operators available on the chip, and the greater degree of parallelism.

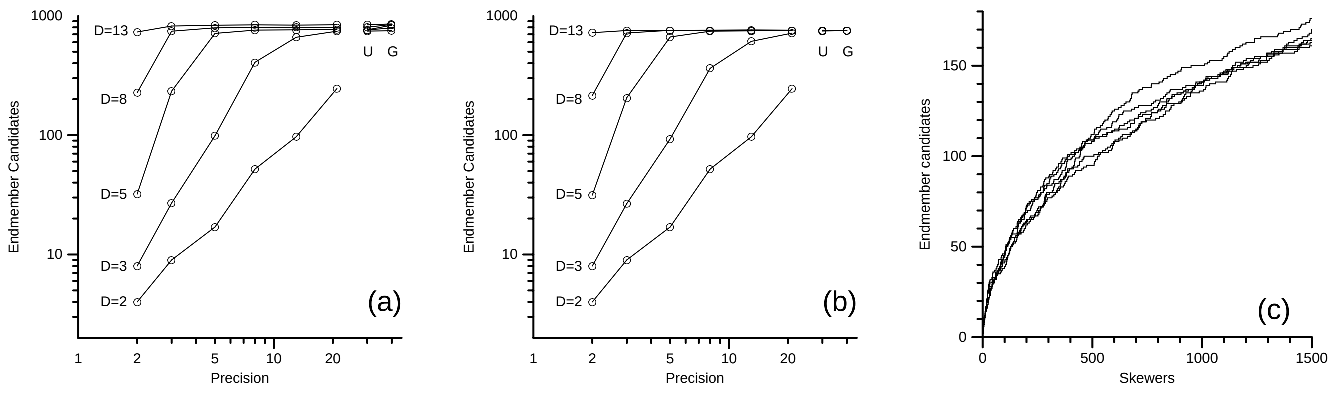


Figure 1. **(a)** The data set consists of $N = 1000$ points placed randomly on the surface of a unit D-dimensional sphere; $K = 1000$ skewers are produced using coefficients chosen randomly from a discrete set of d values. For example, $d = 3$ corresponds to the set {-1,0,1}. The number of candidate endmembers identified by the PPI algorithm is plotted against the precision d. The special values "U" and "G" correspond to continuous-valued random coefficients taken from distributions that are uniform and gaussian, respectively. The plot shows the average of ten runs. **(b)** This experiment is similar to (a), except that the data points are placed on a 2-dimensional sphere (ie, a circle) that is randomly oriented in D-dimensional space. Even though the "actual" dimension of the data is small, the number of identified candidate endmembers depends on the nominal dimension D, corresponding not to the nature of the data but directly to the number of channels in the hyperspectral image data. **(c)** PPI is applied to an actual hyperspectral image, an AVIRIS image with $D = 224$ channels, and $N = 128 \times 128$ pixels. The number of endmember candidates increases with the number of skewers in a way that is for the most part independent of d; oddly, the one curve that is slightly better than the others corresponds to the $d = 2$ case – the one with *least* precision.

If the skewer coefficients are chosen at random from a gaussian distribution, then the distribution of skewer directions in D dimensional space will be isotropic. If the distribution is nongaussian (eg, uniform), and especially if the skewer coefficients are limited to a small discrete set of values (eg, {-1,0,1}), then the sampling of space in search of the best endmember candidates will be less efficient.

We have performed numerical experiments to quantify this effect, and to determine the precision necessary for the PPI algorithm to efficiently sample D dimensional space. It is not hard to see that there is a theoretical upper bound of d^D distinct skewer directions, where d is the number of discrete values permitted in each coefficient. This bound grows very rapidly with D, and our main result involves the more practical issue of identifying candidate endmembers. We find that when D is even moderately large, a severe truncation of skewer precision can be tolerated with very little effect on the ability to identify endmembers.

Fig. 1(a) shows that the number E of endmember candidates identified from a PPI run increases with the precision of the skewer coefficients. The increase is quite evident for low-dimensional data, but for moderate to higher dimensional data, even a single bit of precision per coefficient (ie, each coefficient has a value of -1 or 1) performs about as well as full floating point precision. Fig. 1(b) shows that this result obtains even when the "actual" dimension of the data is small; even if the data can be projected to a small set of principal components (so its effective dimension is small), PPI can still find a large number of candidate endmembers as long as the nominal dimension (ie, the number of spectral channels) is moderately large. Finally, Fig. 1(c) shows that for an actual hyperspectral image datacube, truncation of skewer precision has negligible effect on the number of candidate endmembers that are identified by PPI.

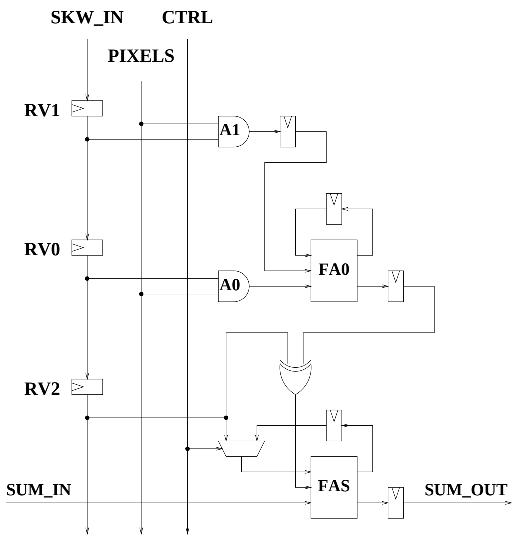
The immediate consequence in terms of hardware is that the cost of a dot-product operator can be significantly reduced. A multiplier can simply be reduced to an adder enhanced with some specific features. The two following architectures take advantage of this.

4. BIT-SERIAL ARCHITECTURE

4.1. Architecture

The bit-serial architecture sets the parameter NS equal to 1 and computes KS dot-products in P cycles, with P the precision (number of bits) of the result. That is, the calculation of a dot-product is pipelined, so that after a latency of $D + P$ (D is the number of spectral bands), KS results are produced every P cycles. Figure 2(A) illustrates the principle of the architecture.

The architecture is composed of an array of $D \times KS$ identical bit-serial multiplier/accumulator operators (MAC). A dot-product operator is thus an horizontal arrangement of D MACs. The pixels are bit-serially entered simultaneously to the KS dot-product operators. At time $t = 1$ the first column of MACs performs the multiplication of the first bit of the first pixel for the spectral band #0. At time $t = 2$ the second column of MACs perform the multiplication of the first bit of the first pixel for the spectral band #1 and add the result produced by the first column during the previous cycle.



Each MAC operator holds a coefficient representing the value of one skewer element. This coefficient must first be downloaded before starting the dot-product computation. Fortunately, the value of the coefficients can be very small (see section 3), leading to a fast initialization and to a very simple MAC operator.

The opposite figure gives the architecture of a bit-serial MAC operator storing a 3-bit coefficient. The coefficient is stored in a shift register (RV0, RV1, RV2) linked to its left and right neighbors, providing an inexpensive initialization mechanism. The AND gates `A0` and `A1`, and the 1-bit full adder FA0 constitute the multiplier. The 1-bit full adder `FAS` adds or subtracts the result of the multiplication to the `SUM_IN` input according to the value stored in the register `RV2`. This operator is thus able to perform a bit-serial multiplication ranging from -3 to +3. The control signal (`CTRL`) reset the adder before starting a new MAC operation.

The output of the dot-product operators are directly linked to the min/max units. They first de-serialize the data coming from the dot-product operators and perform the min and max pixel index calculation using standard comparison operators. When a complete image has been processed, the KS min/max units hold two pixel indexes representing the extrema.

4.2. FPGA Implementation

This architecture has been implemented on the Wildforce board from Annapolis Microsystems, Inc.[4] The board available at LANL is composed of five Xilinx XC4036EX processing elements[5] interconnected in a 36-bit width systolic ring. Each processing element, except PE0, is connected to a 512 Kbytes memory (128 K 32-bit words). The processing elements PE0, PE1 and PE4 are connected to a PCI interface bus by a bidirectional 512 32-bit word fifos. The memories are dual-port memories accessible both from the host (through the PCI bus) and the processing element. A programmable crossbar connects all the processing elements.

An elementary access memory can read or write a 32-bit word, leading to a maximum storage of 32 spectral bands per PE, or a 128-bands hyperspectral image considering all the available memory. As shown Figure 2(B), each PE implements an array of 12×32 bit-serial MAC (or 12 dot-product operators). Pipelining horizontally 4 of these operators gives a dot-product operator capable of processing a 128-band image. Finally, the Wildforce board fits 12 dot-product operators delivering 12 results every 16 cycles.

The hyperspectral image is directly loaded from the PCI bus and the skewers are initialized through the processing PE0 which acquires and writes data from two FIFOs.

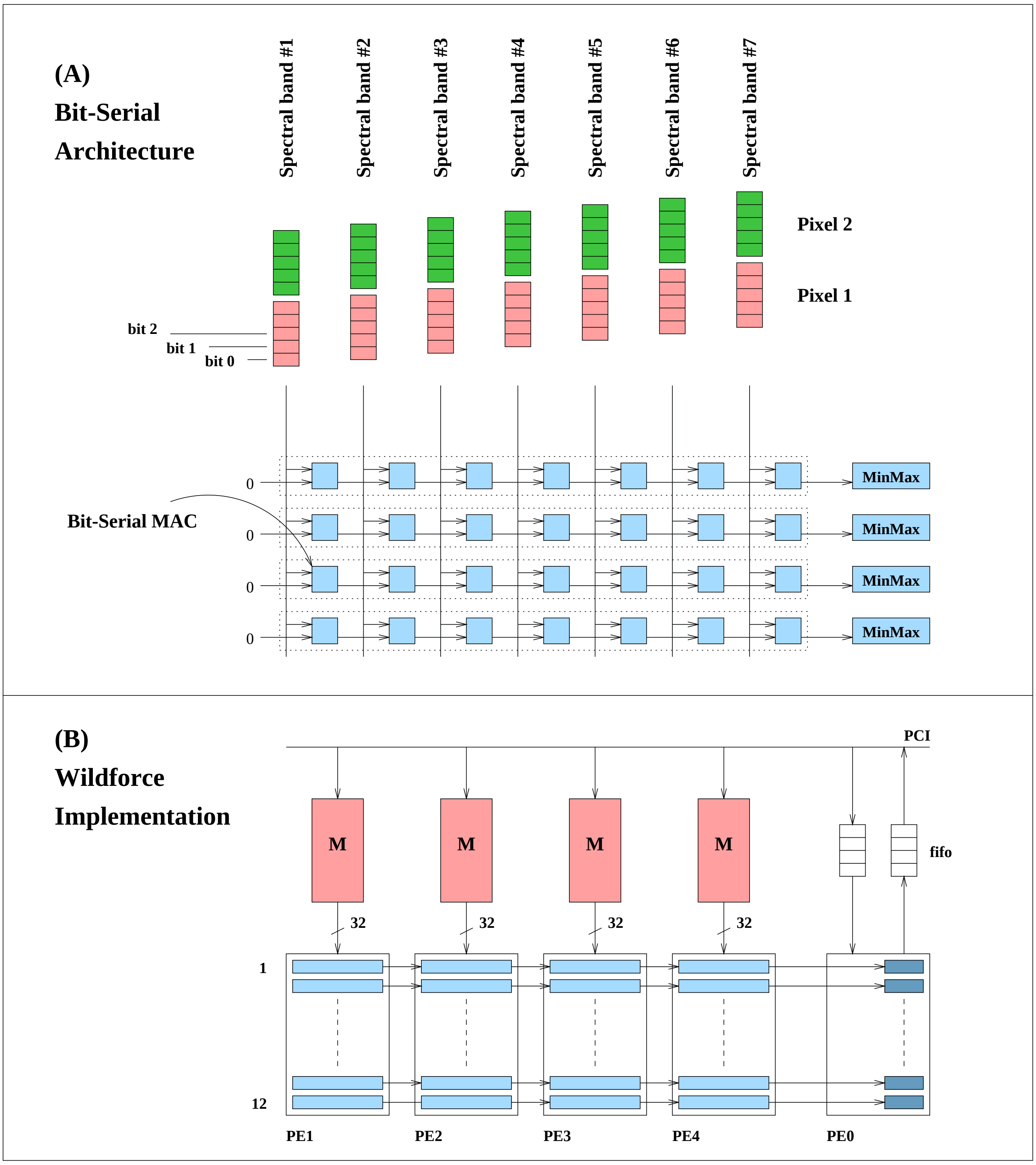


Figure 2. (A) Bit-serial architecture: A dot-product operator is made of D horizontal bit-serial multiplier/accumulator operators (MAC). KS dot-products are computed simultaneously, and after a latency of $D + P$ cycles, KS dot-products are output every P cycles – P represents the precision (number of bits) of the calculation. Each MAC operator stores internally a multiplicative coefficient representing the skewer values. **(B) Wildforce implementation:** A dot-product operator is made of 128 bit-serial MAC operators split into the processors PE1 to PE4. 8 dot-products fit into the board. The MinMax units are implemented on PE0.

4.3. Speed-up

We measure the speed-up as the ratio between the execution time on a standard 450 MHz PC running an optimized C-code, and the execution time on the same PC runing the parallel execution on the Wildforce board. As an example, running sequentially PPI on a 128-band 512×614 hyperspectral image (with 8K skewers) takes 215 minutes.

The time for executing the PPI algorithm on the board is given by:

$$Ti + \frac{K}{KS} \times (Ts + \frac{N \times P}{F})$$

- Ti is the time for loading the image in the memory
- Ts is the time for initializing the skewers
- K, N are respectively the number of classes and the number of pixels
- KS is the number of classes processed in parallel
- P is the precision of the results (16 bits)
- F is the frequency (25 MHz)

Actually, the time for downloading a hyperspectral image and initializing the skewers is very small compared to the time for computing the dot-products. In that case, I/O transfers between the host and the board are not a limitation. Thus, neglecting the host/board transfer, we obtain an execution time of:

$$\frac{K}{KS} \times (\frac{N \times P}{F}) = 134 \ sec$$

The maximum speed-up is equal to $(215 \times 60)/85 = 96$. Real tests for different sizes of images have shown an average speed-up of 80. The elapsed time has been measured with the unix `time` command for both the sequential version and the parallel FPGA version.

Despite the good performances we get, this architecture suffers two main drawbacks. First, the number of spectral bands is dictated by the board architecture. In our case, a read memory access provides 128 bits per cycle, and prevents the processing of hyperspectral images with higher numbers of spectral bands. Second, the image must be pre-processed for bit-serial operation: the pixels must be split over the different memory banks and carefully shifted to meet the bit-serial time scheduling requirement. In addition, pixel values have to be encoded in P bit values (P is the precision of the calculation) resulting in a waste of memory space.

5. OPTIMIZED OPERATOR ARCHITECTURE

This section proposes an alternative architecture overcoming the disadvantages of the bit-serial one. It's a more scalable architecture allowing the processing of larger hyperspectral images.

5.1. Architecture

The principle of the architecture is represented figure 3(A). It is composed of a matrix of $NS \times KS$ dot-product operators. Each dot-product is fed serially with the pixels and the skewers. The results of the dot-product are stored into registers and shifted to KS MinMax units. These units compute the minimum and the maximum of the dot-product and kept track of the pixels which produce extreme values.

In order to get faster performance, the minimum and maximum operations are performed in parallel with the dot-product computation: as soon as a dot-product phase is accomplished, the results are stored into a shift register, and another phase begins immediately. During the next dot-product phase, the previous dot-product values are bit-serially shifted to the MinMax units providing a complete overlap between the dot-product and the minimum/maximum computations.

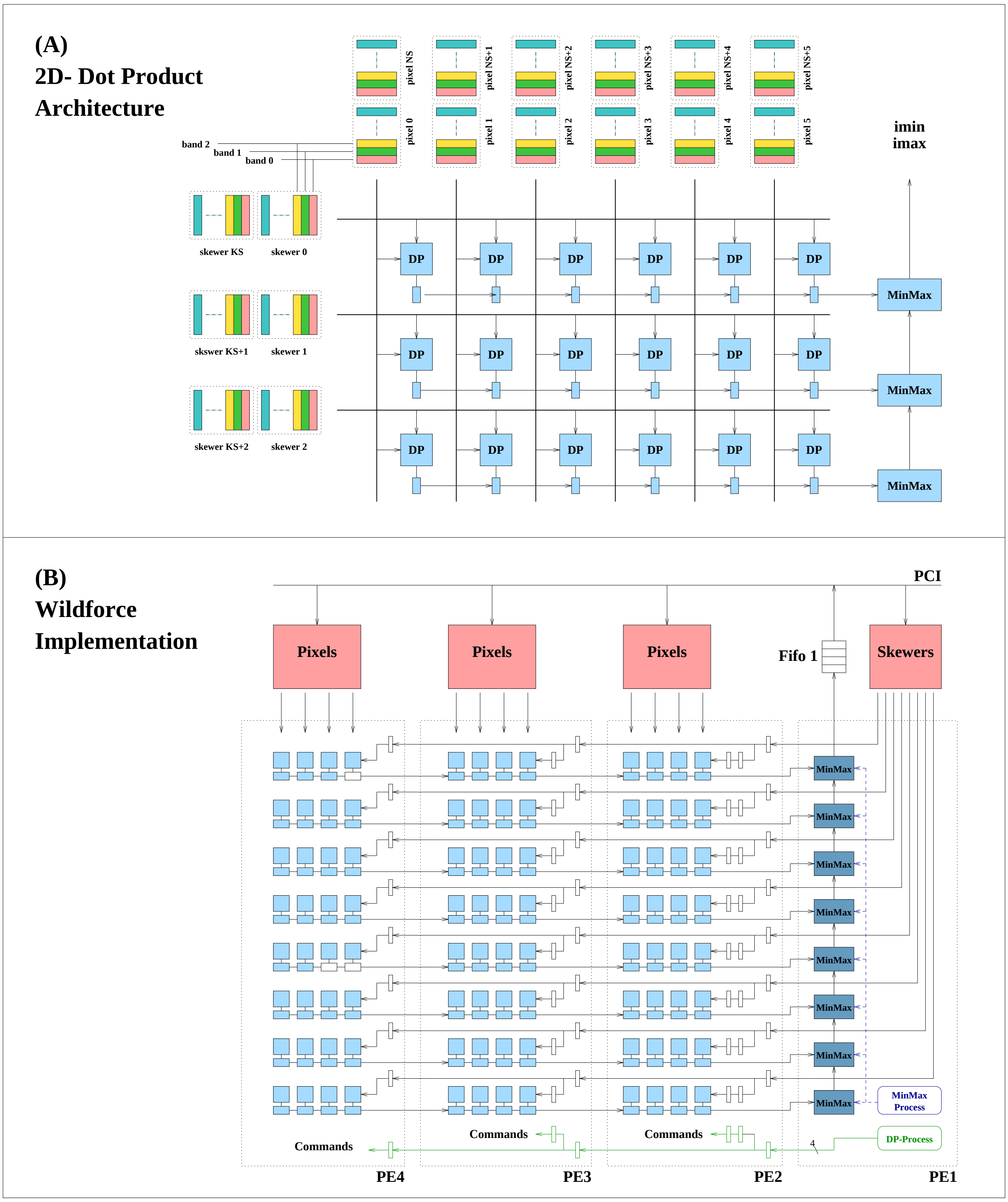
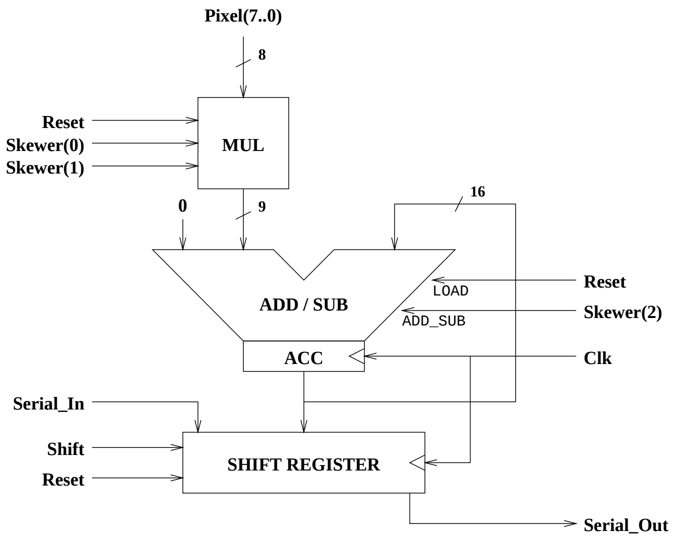


Figure 3. (A) Optimized Operator Architecture: KS skewers and NS hyperpixels are sent simultaneously to a two-dimensional array of optimized dot-product operators. After D steps (D is the number of spectral bands) $KS \times NS$ dot-products are available. They are shifted out to KS min/max units which calculate the index having produced the maximum and minimum dot-product. **(B) Wildforce Implementation:** PE2, PE3 and PE4 house each 32 dot-product operators. PE0 manages the control of the min/max units and feed the dot-product array with the skewers.

The dot-product operator is an optimized 16-bit multiplier/accumulator. Pixels are 8-bit encoded and skewers are 3-bit encoded. The skewer values range from -2 to +2. The multiplication by 0, 1 or 2 is straightforward: it's a null value, the pixel itself, or the pixel shifted to the left.



The figure to the right details the architecture of such an operator. The heart is an adder/subtracter-accumulator provided by the Xilinx logiblox library from the Alliance Series FPGA CAD tools.[6] The inputs are connected to a multiplier unit which is actually reduced to a multiplexer operator for providing either a null value, a direct pixel value or a 1-bit left-shifted value. The `Reset` signal resets the accumulator to zero and loads the shift register with the last computed dot-product.

The overall architecture runs two processes concurrently. The first one controls the dot-product array, and the second one controls the MinMax units. Both are implemented with a state machine. The dot-product process is the main process. It handles the outer loop which performs K/KS iterations corresponding to the number of skewers (K) divided by the number of skewers concurrently computed (KS). Each iteration computes N dot-products (N = number of pixels). Knowing that NS pixels are processed in parallel, N/NS steps are required. The minmax process is started after a dot-product has been completed. It performs NS iterations. Each iteration gets serially KS different dot-products (16 cycles), then computes concurrently KS minimum and maximum according to these new dot-product values. To work properly, the time for computing a dot-product and the time for processing the results must respect the following constraint: $D >= (16 * NS)$ which states that the time for computing a dot-product (D cycles for a hyper-spectral image of D bands) must be greater (or at least equal) to the time for processing the min and max (precision of the results multiplied by the number of pixels processed concurrently).

5.2. FPGA Implementation

The architecture is split into the four processing elements PE1, PE2, PE3 and PE4 of the Wildforce board as shown figure 3. PE0 is not used. PE2, PE3 and PE4 house each 32 dot-product operators, leading to a total of 96 operators. PE1 contains 8 MinMax units and the two state machines. The memories of the processing elements PE2 to PE4 contain the pixels. The memory of PE1 contains the skewers.

The control located into the processing element PE1 is pipelined through the processing elements PE2 to PE4. The skewers, located into the memory of PE1, are propagated in the same way. The hyperspectral image is stored into the memory of the processing elements PE2 to PE4.

Since the size of a PE memory is 512 Kbytes, it can contain a maximum of 2340 224-band hyperpixels. In other words, the board memory dedicated for storing a hyperspectral image can hold a maximum of 7020 pixels. The current implementation stores 6144 pixels (512 × 12): each processor stores 2048 pixels and can simultaneously access 4 of them. Similarly, the PE1 memory stores 4096 skewers (512 × 8) and a single memory access provides 8 values.

The design has been described in VHDL, and synthesized with Synplify from Synplicity Inc..[7] Manual placement has been performed to get a *regular* layout array in order to optimize the place-and-route step and obtain better performance. The design works at 25 MHz.

5.3. Speed-up

Again, the speed-up is measured as the ratio between the execution time (given by the unix command `time`) of the PPI sequential algorithm and the Wildforce implementation. The time for executing the PPI algorithm on the Wildforce board is:

$$Ti + Ts + \frac{K \times D \times N}{KS \times NS} \times \frac{1}{F} \quad \text{which can be approximated to} \quad \frac{K \times D \times N}{KS \times NS} \times \frac{1}{F}$$

for the same reasons as discussed in the bit-serial architecture: the downloading time of both the skewers and the image represents a very small fraction of the time compared to the dot-product computation time. Tests have been made on real data provided by satellite remote sensors. As an example, it takes 190 minutes on a 450 MHz PC to process the PPI algorithm on a 512×640 pixels 224-band image with 4K skewers, and 140 seconds using the Wildforce board, providing a speed-up of 80.

6. HIGH LEVEL SYNTHESIS

The above algorithms were designed and implemented in synthesizable Register Transfer Level (RTL) VHDL. Optimized operators from the Xilinx Hardware Library were explicitly instantiated as components of the basic dot product units. In addition, the optimized architecture was manually floorplanned to optimize placement for subsequent routing, gaining a higher clock frequency than an automatically placed design. In this section, we discuss an automated design approach: we use the Streams-C C-to-VHDL compiler to describe the optimized architecture, and compare design time and circuit performance between the two approaches.

6.1. Overview of Streams-C

The concept of stream-based computation is a fundamental formalism for high performance embedded systems, which are characterized by (multiple) streams of data produced at a high rate, with complex operations performed on the incoming data. The Streams-C[8,9] language supports this computational model with a minimal number of language extensions and library functions callable from a C program. The compiler generates RTL VHDL for a target FPGA board containing multiple FPGAs, external memories, and interconnect. The language extensions, such as declarations for a process or stream, allocate resources on the board for these objects. These extensions allow the programmer to allocate registers on an FPGA and define register bit lengths; assign variables to memories; define concurrent processes; define stream connections between processes; and read/write streams to communicate data between processes. The processes operate asynchronously, and synchronize through stream operations, which may occur anywhere within the body of the process. A distributed memory model is followed, with local state belonging to each process and inter-process communication via streams. The extensions include mapping directives to give the applications developer control over the mapping of processes to hardware components and of streams to communication media on the target application board.

A hardware streams library has been built for the Annapolis Microsystems Wildforce accelerator board. The compiler, based on the Napa C compiler and Malleable Architecture Generator (MARGE), synthesizes hardware circuits from a C-language program. Although the target is a synchronous set of circuits on multiple communicating FPGAs, the C programmer does not have to be concerned with synchronizing state machines, or other hardware timing events. The compiler-generated state machines control sequence and loops. The hardware streams library encapsulates the data flow synchronization between stream reader and writer. The combination of compiler-generated computation nodes with the hardware streams library allows applications developers to target FPGA boards from a high level concurrent language. A software library using POSIX threads is used to provide concurrent processes and stream support in software and several image processing applications have been prototyped with the software library. The figure 4 shows the simulation path and the synthesis path for the Streams-C compiler.

6.2. Streams-C PPI Implementation

Our Streams-C development approach uses the same parallelization and mapping that is implemented by hand and compares the area utilization and speed on the same hardware. In addition, we compare the development time between hand-coded VHDL and the Streams-C version. The optimized-operator architecture was written in Streams-C. A software simulation (cf figure 4) provides verification that the algorithm written in Streams-C yielded correct results as compared to a conventional C program. Second, the Streams-C compiler generates the hardware design in VHDL for the Wildforce board. This hardware design is modeled and simulated for verification before synthesis and place-and-route on the hardware.

The optimized-operator architecture has been written in Streams-C. In the Streams-C version, there are several concurrent processes, each written as a C subroutine. The processes communicate by reading and writing streams. Some of the processes are resident on the conventional processor, and others are resident on FPGAs on the Wildforce board. In contrast, in the hand coded versions, the designer must write a separate program on the host computer to control the FPGA board and supply it with data, and coordinate it with the VHDL designs on the FPGAs. In

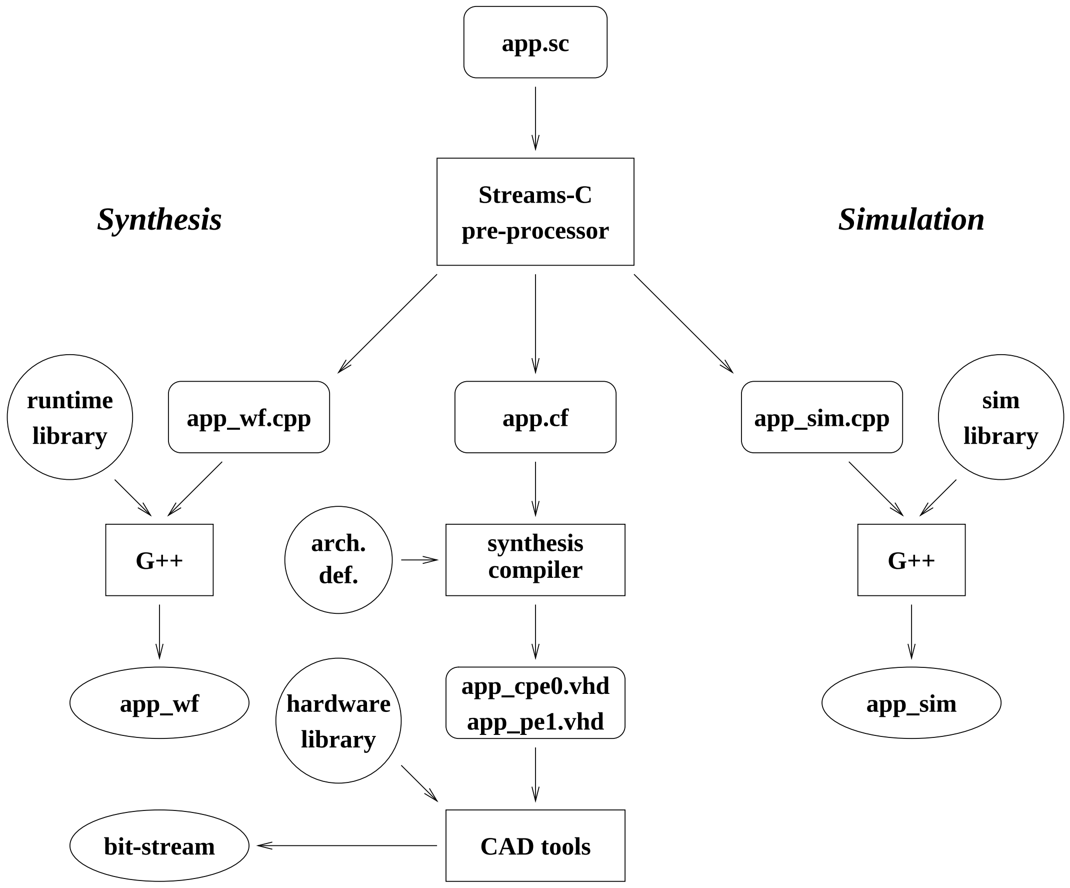


Figure 4. Organization of the Streams-C compiler: The application written in streams-C (app_sc) goes through the streams-C preprocessor which produces three descriptions: app_sim.cpp for simulation purpose, app.cf as an entry point for the synthesis compiler, and app_wf.cpp for running the host process. The synthesis compiler translates the app.cf file into a VHDL description for each processing element of the Wildforce board. Thus the VHDL files are applied to the Xilinx CAD tools which generate the bit-stream for the FPGA components.

the Streams-C program, the dot-product computation and min/max computation are implemented as concurrent processes.

In the table below, we compare the hardware generated for the computationally intensive process, the parallel dot product operations, between automatic synthesis and hand-coded and optimized designs. For each of them, we give the percentage of CLBs used and the clock frequency obtained after place and route.

Architecture	Area	# DP Processors	Clock Frequency
Streams-C	100%	2	15 MHz
Hand-coded	90%	8	25 MHz

The automatically generated version could fit 2 dot product processors into a Xilinx 4036-EX at a frequency of 15 MHz. The design routed at 20 MHz with a single unrouted signal. A three processor design ran at 1 MHz.

If the hand-coded versions provide better performance both in terms of speed and resources used, one may not forget the time spent for designing these different versions. A rough estimation approximates a design time from 4 to 6 weeks for each hand-coded versions while the design time using the Streams-C compiler is decreased to 3-4 days, again for each versions.

7. CONCLUSION

We have presented two architectures tailored to FPGA components for speeding up the Pixel Purity Index algorithm. Both architectures deliver similar performance on the Wildforce board and take advantage of the limited precision the skewer coefficients can afford. The two designs work at an equivalent speed whereas the bit-serial implementation should theoretically provide a higher clock frequency. The reason is that the processing elements need to address their memory through counters which actually limit the frequency. A stand alone bit-serial dot-product runs at more than 40 MHz, but requires 128 new bits of data to be provided on each cycle.

The last experiments done concerning the precision of the skewer coefficients indicate that for hyperspectral images good results are obtained for a small discret set of values such as {-1,0,1}. In that case, the multiplication of the dot-product disappears and this operation is reduced to an accumulation of positive or negative values. From a hardware point of view, this means that a higher number of dot-product operators can fit into a single chip, providing a higher parallelism.

Finally, the automated approach experimented through the Streams-C compiler needs to be discussed more deeply. In a research environment, the use of reconfigurable platforms to reach high performance may be not the ultimate goal. In such an environment, the algorithmic research activity for developing new hyperspectral processing treatments is very important, and the need for rapidly testing different variants of time-consuming algorithms becomes obvious. In that case, spending a few weeks programming an FPGA board is wasteful: even if it takes days to get some results from a sequential version, it may be better than waiting weeks to have the hardware ready to get results in a handful a minutes! Furthermore, an optimized implementation is always difficult to modify when significant changes occur (cf[10] and may require a complete redesign that is a painful task when following a traditional design approach.

A tool like Streams-C is an alternative: the algorithmic designer can rapidly get a hardware version with a significant speed-up compared to a standard workstation. Modifications can easily be handled from the high level description and reported automatically in the hardware. Hence, a few variants can easily be tested and the best one serve as a base for further optimized implementation.

REFERENCES

1. J. W. Boardman, "Automating spectral unmixing of AVIRIS data using convex geometry concepts," in *Summaries of the Fourth Annual JPL Airborne Geoscience Workshop*, R. O. Green, ed., pp. 11–14, 1994.
2. J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in *Summaries of Fifth Annual JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 23–26, 1995.
3. D. Lavenier and J. Theiler, "FPGA implementation of the pixel purity index algorithm for hyperspectral images," Tech. Rep. LA-UR 00-2466, Los Alamos National Laboratory, 2000.
4. "Wildforce reference manual," Tech. Rep. revision 3.4, Annapolis Micro Systems, Inc., 1999.
5. "Xilinx xc4000e and Xilinx xc4000x series field programmable gate array, product specification," Tech. Rep. version 1.6, Xilinx, Inc., 1999.
6. "Alliance series," Tech. Rep. 2.1i, Xilinx, Inc., 1999.
7. "Synplify user guide," Tech. Rep. release 5.2.2, Synplicity Inc., 1999.
8. M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE international Symposium on FPGAs for Custom Computing Machines*, 2000.
9. M. B. Gokhale and J. M. Stone, "Co-synthesis to a hybrid RISC/FPGA architecture," *Journal of VLSI Signal Processing Systems* **24**, March 2000.
10. J. Theiler, D. Lavenier, N. R. Harvey, S. J. Perkins, and J. J. Szymanski, "Using blocks of skewers for faster computation of pixel purity index," in *Proceedings of the SPIE'e 45 Annual Meeting, The international Conference on Optical Science and Technology*, vol. 4132, 2000.