# Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems

Jeffrey P Hammes, Bruce A Draper, and A P Willem Böhm

Colorado State University, Fort Collins CO 80523, USA,
(hammes|draper|bohm)@cs.colostate.edu

**Abstract.** This paper presents Sassy, a single-assignment variant of the C programming language developed in concert with Khoral Inc. and designed to exploit both coarse-grain and fine-grain parallelism in image processing applications. Sassy programs are written in the Khoros software development environment, and can be manipulated inside Cantata (the Khoros GUI). The Sassy language supports image processing with true multidimensional arrays, sophisticated array access and windowing mechanisms, and built-in reduction operators (e.g. histogram). At the same time, Sassy restricts C so as to enable compiler optimizations for parallel execution environments, with the goal of reducing data traffic, code size and execution time.

In particular, the Sassy language and its optimizing compiler target reconfigurable systems, which are fine-grain parallel processors. Reconfigurable systems consist of field-programmable gate arrays (FPGAs), memories and interconnection hardware, and can be used as inexpensive co-processors with conventional workstations or PCs. The compiler optimizations needed to generate highly optimal host, FPGA, and communication code, are discussed. The massive parallelism and high throughput of reconfigurable systems makes them well-suited to image processing tasks, but they have not previously been used in this context because they are typically programmed in hardware description languages such as VHDL. Sassy was developed as part of the Cameron project, with the goal of elevating the programming level for reconfigurable systems from hardware circuits to programming language.

## 1 Introduction

A common programming methodology in image processing and computer vision uses a graphical programming environment where application programs are constructed by graphically interconnecting the outputs of one primitive operator to the inputs of another. One of the most widely used graphical programming environments for image processing and computer vision is Khoros(tm) [22], but other environments exist or are being developed for this purpose as well, including the Image Understanding Environment (IUE) [17] and CVIPtools [23]. These programming environments are advantageous in that they separate application

programming, where domain knowledge may be required, from low-level image processing and/or computer vision programming, and even lower level machine dependent parallel programming. Also, they allow application programs to be distributed across multiple processors, or to be assigned to special-purpose co-processors.

This paper presents Sassy, a programming language based on C that has been developed in concert with Khoral Research Inc. (KRI). This work is part of the Cameron project, which seeks to create a high-level programming environment for image processing and computer vision that is targeted for fine-grain parallel processors. The goal is for low-level image processing algorithms to be written in Sassy inside the Khoros software development environment. Sassy programs can then be manipulated as glyphs inside Cantata (the Khoros GUI) just like any other program. Sassy programs can be executed on parallel architectures, and in particular on reconfigurable (a.k.a. adaptive) computing systems using field programmable gate arrays (FPGAs). Such reconfigurable systems are used in conjunction with a host processor, as shown in figure 1; the reconfigurable hardware has one or more FPGA chips and local memories that hold FPGA configurations and can be used as local memory.

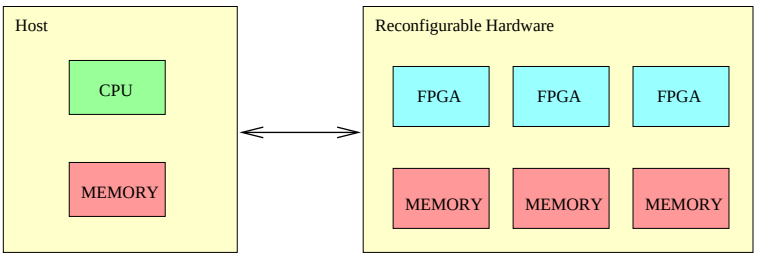



**Fig. 1.** General diagram of host and reconfigurable hardware.

Image processing (IP) applications feature large, regular image data structures and regular access patterns and hence can benefit from parallel implementations. Past attempts to exploit this regular parallelism, for instance with vector or pipelined co-processors, have suffered from communication bottlenecks between the host and co-processor, because of their fixed computation / communication behavior. The programmable nature of FPGA based parallel systems allows greater flexibility, and still promises massive parallelism and high throughput. Reconfigurable systems are therefore interesting candidates for special purpose IP acceleration hardware. Currently, FPGAs are not used in this context, because they are programmed using hardware description languages. The goal of Sassy is to make FPGAs available to IP experts, as opposed to circuit designers.

In many respects, Sassy is similar to other parallel, C-based (or C++-based) languages for image processing such as C\\ [6] or $C_T++$ [1], in that it parallelizes loops that operate over large arrays (e.g. images). Sassy offers a powerful set of language facilities for supporting image operations, including windowing facilities (with or without padding at the image boundaries), the ability to select array "sections" (rows, columns, windows, etc.), and reduction operators such as histogram and accumulation primitives. Sassy also supports true multidimensional arrays.

Unlike these other languages, however, Sassy supports fine-grain instruction-level parallelism. Sassy is a single assignment language, meaning that each variable can be assigned only once. Sassy also forbids recursion and pointer manipulation. These restrictions allow data dependencies in Sassy programs to be analyzed, enabling compiler optimizations, such as partial evaluation, strip-mining and loop reordering. The Sassy compiler can combine code from two or more loops, scheduling as many operations as possible on a selected window of pixels before moving on to the next window. This minimizes the number of times image data must be transferred between the host and the parallel co-processor, easing the primary bottleneck in many parallel image processing applications.

The Sassy program parts that are to be executed on the FPGA are converted into dataflow graphs that map directly onto FPGA configurations (circuits) to exploit the flexibility of reconfigurable hardware.

## 2 Reconfigurable Systems

The Sassy language is able to exploit coarse-grain, loop-level parallelism that should be useful for a variety of parallel architectures. It is also able to use the fine-grain, instruction-level parallelism that can be exploited on reconfigurable computing systems based on integrated circuits called field-programmable gate arrays (FPGAs). These chips, made by manufacturers such as Xilinx and Altera, are used along with optional on-board memory and/or co-processors in boards such as Wildfire(tm) by Annapolis Microsystems.

Field programmable devices, including FPGAs, already enjoy an established market and have been used extensively in digital devices of many kinds. They offer the manufacturer or user quick turnaround, low start-up costs and ease of design changes. Their speed of programming has improved, and it now is feasible to supplement a conventional CPU with one or more FPGAs and not only to configure them with custom functionality for each program that is run, but even to change the configuration during a program's execution.

Figure 2 shows the structure of a typical FPGA [2]. It consists of a grid of logic cells interspersed with wires to connect them. The perimeter has I/O cells that interface with the external pins of the chip. One example of a currently available FPGA is the Xilinx XC4085XL [29]. It is made up of 3,136 Configurable Logic Blocks (CLBs) and 448 I/O Blocks (IOBs). Each CLB has two 16-bit static random access memories (SRAMs), two D flip-flops, and a small number of miscellaneous multiplexors and gates. A CLB can function as RAM

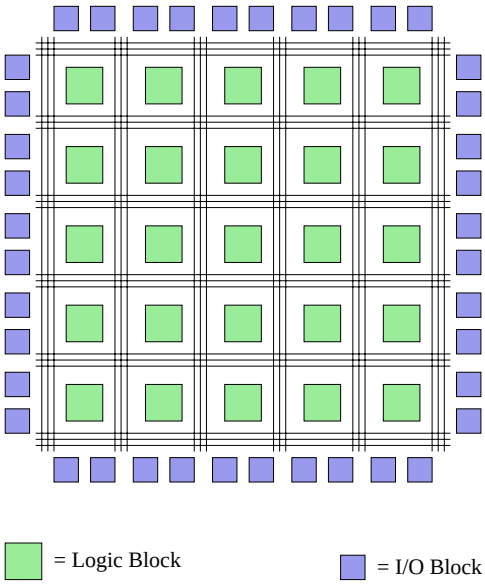



**Fig. 2.** Structure of an FPGA

or as combinational logic. Each IOB controls one external pin of the chip, and contains two D flip-flops and associated buffer-driver circuits. The chip supports system speeds up to 80 MHz. Reconfigurable systems are programmed through configuration codes, which specify the functions of the cells and their interconnections within the FPGA.

Although the clock speeds of current FPGAs are lower than RISC processor clocks, the potential massive parallelism in an FPGA makes them good candidates for many real-time image processing tasks. Dehon calculates that, even with their lower clock speeds, FPGAs may have an order of magnitude better *computational density* (the number of bit operations a device can perform per unit of area-time) as compared with RISC CPUs [5]. Petersen and Hutchings specifically consider digital signal processing tasks, and also calculate a ten-fold speed-up [21]. More importantly, Hartenstein et al. demonstrate a ten-fold speed-up for *jpeg* image compression, a common image processing task [8]. Also, new investments in reconfigurable systems seem destined to increase their clock speeds and make inexpensive fine-grain parallel processors available for a wider range of applications.

Reconfigurable computing presents new challenges to language designers and compiler writers. The task of programming and compiling applications will consist of partitioning the algorithm between a host processor and reconfigurable modules, and devising ways of producing efficient FPGA configurations for each piece of code. Presently, FPGAs are programmed in hardware description languages, such as VHDL [20]. While such languages are suitable for programming

chips that are used as "glue" logic in digital circuits, they are poorly suited for the kind of algorithmic expression that takes place in applications programming.

The Cameron project provides Khoros with a programming language for expressing IP applications, an optimizing and parallelizing Sassy compiler, and a run-time system for FPGA based, reconfigurable systems. This will enable IP programmers to exploit this desirable new hardware, by using a well known programming interface, and writing high level language code. This paper presents the first component of this trio, the Sassy programming language. A prototype Sassy implementation, generating C code, is available; work on the optimizing compiler for FPGAs continues.

## 3 Related Work

### 3.1 State of the Art: DataCube and Transputers

Most current real-time image processing applications run on either DataCube pipeline processors or Transputers. DataCube [4] produces a number of special-purpose image acquisition and processing boards that communicate with a host processor over a VME or PCI bus. Each DataCube board contains interconnected special-purpose processors designed for specific image processing operations, such as image capture, image arithmetic (addition/subtraction), or taking a histogram. DataCube programmers use a graphical interface to route data through sequences of these special-purpose processors, essentially drawing a small data-flow graph. Unfortunately, programmers are limited to the set of image operations (and interconnections) provided, and must learn a new (albeit graphical) programming language in order to use them. DataCube boards are also clocked at lower speeds than current general-purpose processors (for example, the MaxVideo 250 is clocked at 40MHz).

Transputers, on the other hand, represent an approach to parallelism that is more fine-grain than DataCube boards, but less fine-grain than reconfigurable systems. Each Transputer is a small general-purpose processor on a chip, with four high-speed input/output channels. Transputers can be connected together easily, and while the clock speeds of individual Transputers are typically low (50MHz or less), the speed-up comes via parallelism. Researchers in Munich, for example, have built a real-time image processing system out of three fast (Power-PC-based) Transputers and six slower (T4 or T8) Transputers for high-speed mobile robotics [13]. Unfortunately, it is difficult to program large networks of Transputers in MIMD style, so most Transputer systems for image processing use fewer than a dozen processors.

### 3.2 Parallel Languages for Image Processing

Compared to DataCubes or Transputers, reconfigurable systems offer two orders of magnitude more parallelism: thousands of functional units vs. tens. High-level parallel languages will be needed, however, if this advantage is to be exploited.

Most parallel languages for image processing are based on C or C++, and most are designed to parallelize loops over images or large arrays. C\\ [6], for example, is a parallel language for image processing based on C++ and targeted for the GFLOPS multiprocessor [9]. Variables are declared to be either scalar or parallel; scalar variables are kept on the host, while parallel variables are distributed across processors. A `forall` loop is introduced that allows distributed variables to be processed in parallel. Synchronization commands (*barrier*, *wait*, and *sync*) are also available to allow a programmer to control program threads explicitly.

Other parallel languages for image processing are designed for SIMD, rather than MIMD, programming environments. $C_T++$, for example, defines an array class that allows operations in parallel over the elements of the array [1]. The Image Understanding Architecture (IUA) programming environment used a similar mechanism for programming the IUA [27].

Two other parallel C-related languages, not targeted specifically to image processing, are SAC (Single Assignment C) [24] and Handel-C [19]. SAC emphasizes powerful array operations, for use in scientific numerical codes. Handel-C is designed to target synchronous, mostly FPGA-based hardware, and the language includes bit-precision specifications in its data types.

### 3.3 Functional Languages

Functional languages date back to the early 1960s when John McCarthy designed the Lisp language [15]. A variety of functional languages have been created since then, all built around a central idea: a computation is specified in terms of *pure*, i.e. non-side effecting, expressions [7]. The result of a function's evaluation is based only on its arguments, and the evaluation cannot change the state of the computation outside of the function. This guarantees that a function call's return value on a given set of arguments must be the same regardless of *when* the call takes place. This leads to inherent concurrency. For example, in the expression $f1(a,b)+f2(a,c)$, the calls of $f1$ and $f2$ can be evaluated in parallel since neither can alter a global state or cause side effects to its arguments. Examples of pure functional languages are Sisal [16] and Haskell [12]. Some languages are built around a functional core, but include side effecting extensions that make the language *impure*. An example of such a language is ML [14].

Early functional languages tended to omit loop constructs, relying instead on recursion to express iterative execution, and they emphasized recursive data structures such as lists and trees instead of arrays. However, some modern functional languages include loops and arrays for reasons of efficiency. A loop in a pure functional language is an expression, meaning that it returns one or more values. Arrays in pure functional languages are typically *monolithic*, meaning that the entire array is defined at once. The storage space for the array is dynamically allocated from the heap when the array is created, and the array is automatically garbage collected when it is no longer needed. An array carries its extents (or bounds) with it. Since a pure language is free of side effects, it is not possible to overwrite an array element with a new value. Semantically, updating an array requires copying the current array and replacing the desired value

with a new one. However, compilers often can optimize this with *update-in-place* analysis, giving performance on a par with imperative languages [3].

The clean semantics, automatic garbage collection and inherent parallelism of functional languages make them appealing vehicles for high performance parallel computing. Their side effect-free nature makes compiler analysis much easier than that which is required for imperative languages. In spite of this, performance of functional languages often has been disappointing [10, 11]. Features of some very high level general purpose functional languages, such as higher order functions and lazy evaluation, have created new challenges for compiler writers and have prevented these languages from attaining the performance that some imperative languages achieve.

Sassy is designed to exploit the useful and efficiently implementable aspects of functional languages, while avoiding those aspects that have caused performance problems. This is done by integrating it in C, and restricting the functional aspects of the language to those features that are appropriate for image processing.

## 4 Sassy

Sassy (short for single-assignment C) attempts to take the best features of existing imperative and functional languages and combine them into a language that is amenable to compiler analysis and optimization, and that is well suited for image processing. The language is intended to exploit both coarse-grain (loop-level) and fine-grain (instruction-level) parallelism, as appropriate to the target architecture. The language should be suitable for conventional Symmetric Multiprocessors (SMPs), networks of workstations (NOWs), and vector computers, but the target of interest to the Cameron group is reconfigurable (a.k.a. adaptive) computing systems using field programmable gate arrays (FPGAs). Sassy is based on C in order to be as intuitive as possible to image processing experts, most of whom program in C or C++, but it differs from C in some important ways:

- It is an *expression-oriented*, pure functional language, not imperative.
- Its scalar types include signed and unsigned integers with specified bit widths. For example, the type **uint12** represents a twelve-bit unsigned integer.
- It has no explicit pointers.
- It is non-recursive.
- It has true multi-dimensional arrays, including array sections similar to those in Fortran 90. For instance, "**int10** V[:,:]" declares a two dimensional array of ten-bit integers.
- It has powerful loop generators and return operators similar to those in the Sisal language.
- It has multiple-value returns and assignments.

The elimination of pointers and recursion, and the single-assignment restriction, enable important compiler code optimizations. In compensation for these

restrictions, Sassy programmers are given powerful high-level constructs to create and access arrays in concise ways. As an example, figure 3 shows Sassy code that smoothes an image with a median filter and then convolves it with an edge detection mask. In the discussion that follows, Sassy code examples will be shown with their C or Fortran 90 equivalents.

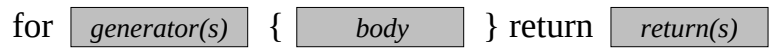
```
P1[:,:] = for window W[7,7] in Image {
            uint12 m = array_median (W);
          } return (array (m));

P2[:,:] = for window W[24,24] in P1 {
            uint12 ip = array_sum (W, M);
          } return (array (ip));
```

**Fig. 3.** Sassy code showing image smoothing with a median filter, followed by convolution with an edge detection mask.

Loops and arrays are at the heart of the language, and the two are closely interrelated. Loops have special forms designed to work with arrays, and arrays are easily created as return values of loops. The Sassy parallel `for` loop is the source of coarse-grain parallelism, and has three parts: one or more *generators*, a loop *body*, and one or more *return* values. The structure is:

**for** *generator(s)* **{** *body* **}** **return** *returns.*




### 4.1 Loop Generators

There are three kinds of loop generators: *scalar*, *array-component* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's `do` loop. The array-component and window generators extract components of arrays in various ways.

Two simple examples illustrate array component extraction:

```
for val in M                    for (i=0; i<n; i++)
    ...                             for (j=0; j<m; j++)
                                        val = M[i][j];
                                        ...

for V(~,:) in M                 do i = 1, n
    ...                             V = M(i,:)
                                        ...
```

The first extracts scalar values from $M$; the second extracts row vectors. Note that these loops automatically access the extents of $M$, making it unnecessary for the programmer to reference them explicitly. The '$\sim$' indicates a dimension over which iterations are indexed, whereas the ':' indicates an array section. The second example shows a Fortran 90 equivalent, since array section capability does not exist in C.

Window generators allow a rectangular window to traverse the source array, as in:

```
for window W[3,3] in M {
    ...
```

```
do i = 1, n-2
    do j = 1, m-2
        W = M(i:i+2,j:j+2)
        ...
```

Each iteration produces a 3x3 sub-array from the source matrix $M$. In general, windows always extract arrays with a rank equal to the rank of the source, but with smaller extents. The language includes a built-in function that pads values around an array's perimeter. This is useful with window generators so that the number of generated windows will be the same as the number of elements in the source array.

When a loop needs to extract values from more than one array, the Sassy programmer can use `dot` and `cross` products to combine generators. The `dot` product runs the generators in lock step, whereas cross products produce all combinations of components from the generators, producing the effect of nested loops. The following two examples demonstrate these two operators:

```
for a in A dot b in B {
    uint8 p = a * b;
    ...
```

```
for (i=0; i<n; i++) {
    p = A[i] * B[i]
    ...
```

```
for a in A cross b in B {
    uint8 p = a * b;
    ...
```

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        p = A[i] * B[j]
        ...
```

Loop generators are important for two reasons. First, they give the programmer a simple and concise way of processing arrays in regular patterns, often making it unnecessary to create loop nests to handle multi-dimensional arrays or to refer explicitly to the array's extents or the loop's index variables. Second, they make compiler analysis of array access patterns significantly easier. In C or Fortran, the compiler must look at index variables generated by the loop nest and relate these indices to their uses in array references. In Sassy the index generators and the array references have been unified; the compiler can reliably infer the patterns of array access.

### 4.2 Loop Returns

Since Sassy is a functional language, every loop returns one or more values. In addition to returning scalar values, a Sassy loop can return arrays and reductions built from values that are produced in the loop iterations. In its simplest form, an array returned from a loop is built out of scalar values that are created in the loop generator or loop body. For example, the following loop creates an array $A$ that has the same shape as matrix $M$, but with each value doubled:

```
uint8 A[:,:] = for val in M
    return (array (2 * val));
```

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        A[i][j) = 2 * M[i][j];
```

This example illustrates that the shape of the return array is determined by the shape of the generator. A generator that extracts scalars from a source array has the same shape as that array, so the returned array $A$ has the same shape as $M$. Also, a Sassy loop body may be empty; this example shows a loop that has only a generator and a return. Arrays may also be built out of other array components, as well as by concatenating array components.

A variety of built-in operators are available to reduce scalar values that occur in loops. Many of these operators reduce to a scalar value, including `sum`, `product`, `min`, and `max`. For example, a loop to compute the dot product of two vectors can expressed as:

```
for a in A dot b in B
    return (sum (a * b));
```

Other interesting reductions exist: `min_indices` and `max_indices` return an array of index locations of the min/max values, while `histogram` returns a histogram of the reduced values as a one-dimensional array. Any of the reduction operators can be used in an `accum` operation where a label value is used to partition the scalars into regions. For example,

```
for a in A dot lab in L
    return (accum (min (a), lab, 4),
        accum (max (a), lab, 4))
```

will find the min and max value in each region of $A$, where the regions are defined by array $L$, as shown in Figure 4.

### 4.3 Sequential Loops

Sassy has sequential (non-parallel) `for` and `while` loops for use when loop-carried dependencies exist. A loop-carried dependency occurs when the value given to a variable in one iteration of a loop uses (directly or indirectly) that variable's value from the previous iteration. The assignment of a new value to a previously

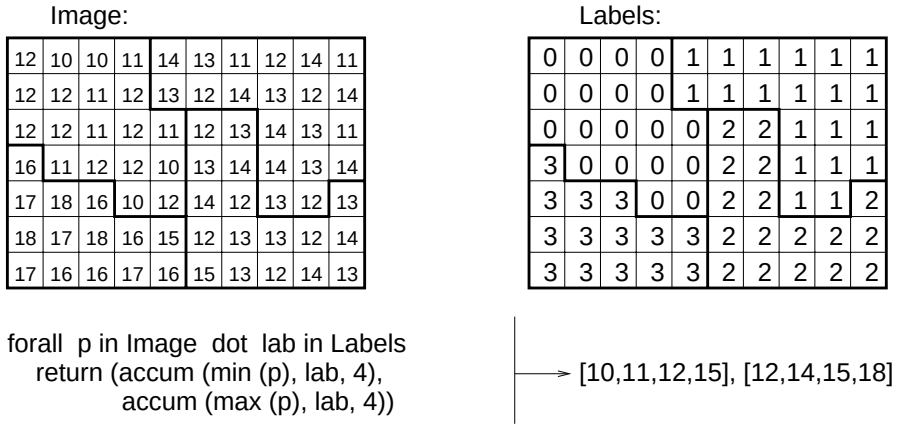



**Fig. 4.** Example of accum operator, finding the min and max values in each region.

defined variable is counter to the idea of single-assignment, so Sassy treats these cases in a special way, as have other functional languages such as Sisal [16] and Id [18]. The keyword `next` allows a current loop variable be to computed based on its previous value. Conway's Life demonstrates an array variable $A$ with a loop-carried dependence. It also shows how a window generator, combined with a mask, can implement general stencil-type operations.

```
uint1[:,:] main (uint1 A[:,:], uint16 n) {
    bool M[:,:] = array_def (bool, [3,3], {
                              {true, true, true},
                              {true, false, true},
                              {true, true, true}});
    uint1 res[:,:] =
        for _ in [n] {
            next A =
                for window W[3,3] in array_conperim (A, 1, 0) {
                    uint3 c = array_sum (W, M);
                    uint1 v = (c==3 || c==2 && W[1,1]==1) ? 1 : 0;
                    } return (array (v));
            print (true, A);
            } return (final (A));
} return (res);
```

Another example of a sequential loop is the following square root function, designed to use only addition and bit-manipulation operators. It uses four variables with loop carried dependencies:

```
uint6 sq_root (uint12 vsqn) {
    bits12 vsq, bits12 asq, bits6 a, bits12 tvsq = vsqn, 0, 0, 0;
    uint6 v = for uint4 i in [6] {
        bits12 nasq = ((bits12)((uint12)asq+(uint6)a)<<2) | 0b1;
        bits6 sa = a<<1;
        next tvsq = (tvsq<<2) | ((vsq>>10) & 0b11);
        next vsq = vsq<<2;
        next a, next asq =
            if (nasq <= tvsq) return (sa|0b1, nasq)
            else return ( sa, asq<<2);
        } return (final (a));
    } return (v);
```

## 5 Parallelism and Performance

Currently, high performance and throughput in FPGAs are achieved by manually optimizing circuit descriptions, using hardware design tools. To avoid this, Sassy is designed so that it can be automatically parallelized and optimized. Sassy exposes both coarse- and fine-grain parallelism. Its parallel `for` loop is a source of coarse-grain parallelism that should be useful across a wide variety of parallel architectures, while single assignment exposes expression parallelism. This allows Sassy procedures to be compiled into dataflow graphs and mapped directly into circuit diagram specifications.

The Sassy compiler will use an intermediate form called "Data Dependence and Control Flow" (DDCF) graphs, similar to the Sisal compiler's IF1 form [25]. This exposes data dependencies and opens up a wide range of loop-related optimization opportunities. The `for` loop generators are attractive because their "outputs" can be viewed as streams, making the transition from a memory-reading execution model to a data-driven stream model that is much closer to the execution that will take place on FPGAs. Similarly, the `array` loop-return operator produces array elements in storage order, making it easy to stream the results back into memory in a straightforward way.

Important DDCF-graph optimizations include

- Loop unrolling, which replicates a loop body one or more times, resulting in pipeline parallelism.
- Loop strip-mining, which splits a parallel loop into a pair of nested loops. The outer loop produces chunks of work; the inner loop performs the work in each chunk.
- Loop fusion, which takes two adjacent loops and fuses them into one loop. This can produce better coupling of array producers and consumers, sometimes completely eliminating intermediate arrays.
- Partial evaluation, which involves statically evaluating parts of an expression where some array references are constant. For example, the inner loop of the

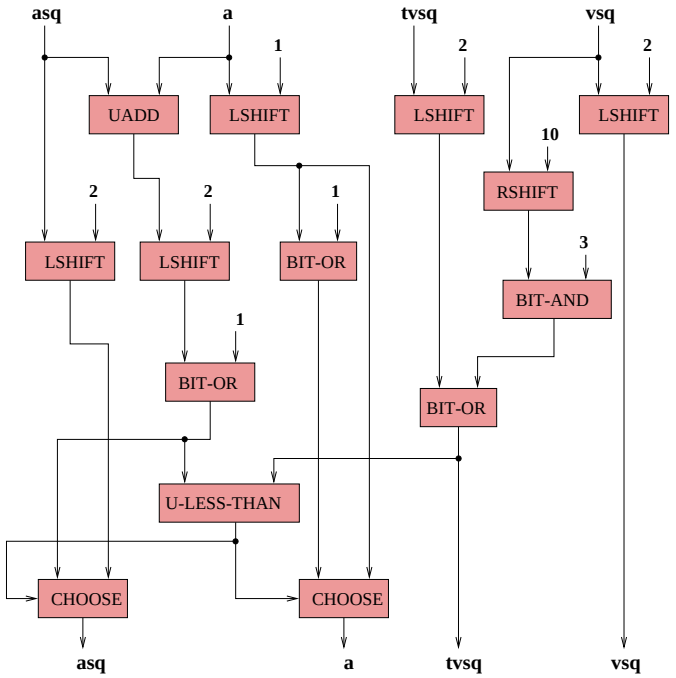



**Fig. 5.** Dataflow graph of square root loop body.

example in figure 3 uses a mask which may be known at compile time. If so, the loop can be fully unrolled and the constant mask values can be substituted in place, with subsequent expression simplification through constant folding.

– Flow control reordering, which is designed to tightly couple producers and consumers, promote data reuse and minimize host-FPGA data movement. In figure 3, array $P1$ can be eliminated; image data can be streamed to the FPGAs in appropriate-size chunks and the median values pipelined directly into the convolution loop.

These optimizations, as well as others, will be applied with the overall goal of streaming arrays into and out of FPGAs and reducing data movement between host and FPGAs. Evaluation order is crucial here, since the FPGA boards typically have limited ability to store temporary data close by; data will be streamed in chunks that are sized such that all necessary data can fit in the FPGA cache memories.

Sassy's single-assignment, functional semantics makes fine-grain, instruction-level parallelism easy to exploit on architectures such as FPGA-based reconfigurable systems. Since each Sassy variable is assigned exactly once, it is possible to create a static dataflow graph of a loop body in which each variable corresponds

to an edge in the dataflow graph. For example, the dataflow graph corresponding to the loop body of the square root function, seen earlier, is shown in figure 5. Reconfigurable computing systems, based on FPGAs, are ideally suited to this approach since a static dataflow graph can be mapped onto FPGA circuits in a straightforward way.

## 6 Conclusion and Future Work

The paper has presented Sassy, a single-assignment variant of C for exploiting both coarse-grain and fine-grain parallelism. Sassy targets image processing applications with true multi-dimensional arrays, powerful generators for accessing elements and sections of arrays, built-in reduction operators (including histogram and accumulate), and bit-precision variables. Although many types of parallel processors can take advantage of Sassy, the single-assignment semantics of Sassy is intended to allow massive parallelism on reconfigurable computing systems.

Sassy is fully defined, and a Sassy to C compiler is implemented. In addition, a Sassy subset to data flow graph compiler is also implemented. As part of the Cameron project, a data flow graph to VHDL compiler is currently being implemented. (Since VHDL compilers are available for most brands of reconfigurable systems, this will allow the compilation of Sassy code for FPGAs.) Also as part of the Cameron project, the VSIP image processing library [26] is being programmed in Sassy.

## References


1. F. Bodin, H. Essafi and M. Pic. A Specific Compilation Scheme for Image Processing Architecture. *Computer Architectures for Machine Perception,* Cambridge, MA, 1997, pp. 56-60.
2. S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A Tutorial. *IEEE Design and Test of Computers*, Vol 12, number 2, pages 42-57, Summer 1996.
3. D. C. Cann. *Retire Fortran? A Debate Rekindled*. Communications of the ACM, Vol 35(8), 1992.
4. DataCube: http://www.datacube/com (or http://robocop.anu.edu.au/docs/MaxVideo250)
5. A. DeHon. Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density. *Proc of Fourth Canadian Workshop of Field-Programmable Devices*, Toronto, Canada, May 1996.
6. A. Fatni, D. Houzet and J. Basille. The C\\ Data Parallel Language on a Shared Memory Multiprocessor. *Computer Architectures for Machine Perception,* Cambridge, MA, 1997, pp. 51-55.
7. A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
8. R. Hartenstein, J. Becker, R. Kress, H. Reinig and K. Schmidt. A Reconfigurable Machine for Applications in Image and Video Compression. *Conf. on Compression Technologies and Standard for Image and Video Compression,* Amsterdam, 1995.
9. D. Houzet and A. Fatni. A 1-D Linearly Expandable Interconnection Network Performance Analysis. *IEEE Int. Conf. On Application Specific Array Processors,* Venice, 1993, pp. 572-582.

10. J. Hammes, O. Lubeck, and A. P. W. Böhm. Comparing Id and Haskell in a Monte Carlo photon transport code. *Journal of Functional Programming*, Vol. 5, Part 3, pp 283-316, July 1995.
11. J. Hammes, S. Sur, and A. P. W. Böhm. On the effectiveness of functional language features: NAS benchmark FT. *Journal of Functional Programming*, Vol. 7, Part 1, pp 103-123, January 1997.
12. P. Hudak, S. Peyton Jones and P. Wadler eds. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, vol 27, number 5, 1992.
13. M. Maurer, R. Behringer, S. Fürst, F. Thomanek, E.D. Dickmanns. A Compact Vision System for Road Vehicle Guidance, *International Conference on Pattern Recognition,* Vienna, 1996. Vol. C, pp. 313–317.
14. D. MacQueen, R. Harper, R. Milner, et al. *Functional Programming in ML.* Lfcs education, University of Edinburgh, 1987.
15. J. McCarthy, et al. *LISP 1.5 programmers manual.* MIT Press, 1962.
16. J. McGraw et.al., SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2, Lawrence Livermore National Laboratory, Memo M-146, Rev. 1, 1985.
17. J. Mundy. The Image Understanding Environment Progam. *IEEE Expert,* 10(6):64-73, 1995.
18. R. S. Nikhil. Id Version 90.0 Reference Manual. Computational Structures Group Memo 284-1, Massachusetts Institute of Technology, 1990.
19. Oxford Hardware Compilation Group. The Handel Language. Technical report, Oxford University, 1997.
20. D. Perry. VHDL. McGraw-Hill, 1993.
21. R. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. *5th Int. Workshop on Field-Programmable Logic and Applications,* Oxford, 1995.
22. J. Rasure and S. Kubica. The KHOROS Application Development Environment. In H. I. Christenses and J. L. Crowley, editors, *Experimental Environments for Computer Vision and Image Processing*. World Scientific, New Jersey, 1994.
23. S. Umbaugh. *Computer Vision and Image Processing: A Practical Approach using CVIPtools.* Prentice Hall, New Jersey, 1998.
24. S. B. Scholz. Single Assignment C – Functional Programming Using Imperative Style. In *Proc. of the 6th International Workshop on th Implementation of Functional Languages*. University of East Anglia, 1994.
25. S. K. Skedzielewski, J. R. W. Glauert. IF1, an Intermediate Form for Applicative Languages. Refernce Manual, M-170, Lawrence Livermore National Laboratory, July 1985.
26. http://www.vsip.org/
27. C. Weems and J. Burrill. "The Image Understanding Architecture and its Software Development Tools," *Applied Imagery and Pattern Recognition Workshop,* McLean, VA, 1991.
28. M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.
29. Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc., San Jose, California, 1998.
30. H. Zima. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, 1990.