

# MAPPING HIGH LEVEL ALGORITHMS ONTO MASSIVELY PARALLEL RECONFIGURABLE HARDWARE

**Issam Damaj**

Centre for Applied Formal Methods.  
London South Bank University  
103 Borough Road  
London, SE1 0AA. U.K.  
damajiw@sbu.ac.uk

**John Hawkins**

Centre for Applied Formal Methods  
London South Bank University  
103 Borough Road  
London, SE1 0AA. U.K.  
john.hawkins@sbu.ac.uk

**Ali Abdallah**

Centre for Applied Formal Methods  
London South Bank University  
103 Borough Road  
London, SE1 0AA. U.K.  
A.Abdallah@sbu.ac.uk

## Abstract

*The main focus of this paper is on implementing high level functional algorithms in reconfigurable hardware. The approach adopts the transformational programming paradigm for deriving massively parallel algorithms from functional specifications. It extends previous work by systematically generating efficient circuits and mapping them into reconfigurable hardware. The massive parallelisation of the algorithm works by carefully composing “off the shelf” highly parallel implementations of each of the basic building blocks involved in the algorithm. These basic building blocks are a small collection of well-known higher order functions such as map, fold, and zipwith. By using function decomposition and data refinement techniques, these powerful functions are refined into highly parallel implementations described in Hoare’s CSP. The CSP descriptions are very closely associated with Handle-C program fragments. Handle-C is a programming language based on C and extended by parallelism and communication primitives taken from CSP. In the final stage the circuit description is generated by compiling Handle-C programs and mapping them onto the targeted reconfigurable hardware such as the Celoxica RC-1000 FPGA system. This approach is illustrated by a case study involving the generation of several versions of the matrix multiplication algorithm.*

## Keywords

Reconfigurable Architectures, FPGAs, Rapid Development, Handle-C, Functional Specification, CSP.

## INTRODUCTION

At one extreme of the computing spectrum, computing systems based on the traditional von Neumann model provide a single and generic computational medium for applications with diverse characteristics. These systems are known as general purpose processors (GPPs). At the other end of the spectrum there are systems with architectures customized for particular applications. These systems are built around one or more Application Specific Integrated Circuits or ASICs. In certain application areas, software executed on a single sequential processor no longer meets our ever increasing efficiency requirements. Besides, the direct archi-

ture algorithm mapping restricts the range of applicability of ASIC-based systems. Consequently, this led to the introduction of reconfigurable computing (RC) combining the flexibility of general-purpose processors and the high performance of ASICs [1-9]. Field Programmable Gate Arrays (FPGAs), an instance of RCs, has recently enabled RC chips with Millions of gates (Xilinx) affording more scalability and cost effectiveness due to hardware reuse. FPGAs offer much flexibility for the design of integrated circuits (ICs) chips for parallelism.

Generally, parallelism and implementation in hardware provide us with two alternatives that can often deliver very dramatic improvements in efficiency. With the emergence of such reconfigurable hardware chips, the presence of a rapid development environment for these scalable hardware circuits is very useful. Moreover, it would constitute the cornerstone solution for the ever-increasing need for more: efficiency, scalability and flexibility in realizing massively parallel algorithms for a wide area of applications [2].

The proposed rapid development model (RDM) adopts the transformational programming approach for deriving massively parallel algorithms from functional specifications (See Figure 1) [1-4]. The functional notation is used for specifying algorithms and for the reasoning about them. This is usually done by carefully combining small number of high order functions (like map, zip and fold) to serve as the basic building blocks for writing high-level programs. The systematic methods for massive parallelisation of algorithms work by carefully composing “off the shelf” massively parallel implementation of each of the building blocks involved in the algorithm. The emphasis in this method is on correctness, scalability and reusability.

To describe parallelism we use Hoare’s CSP that allows issues of immense practical importance (such as data distribution, network topology, and locality of communications) to be carefully reasoned about [12]. Relating the Functional Programming and CSP fields gives the ability to exploit a well-established functional programming paradigms and transformation techniques in order to develop efficient CSP processes.

The reconfigurable hardware realization step is done using Handel-C an automated compilation development model [8, 11]. Handel-C uses much of the syntax of conventional C with the addition of explicit parallelism. Handel-C relies on the parallel constructs in CSP to model concurrent hardware resources. Accordingly, algorithms described with CSP could be implemented with Handel-C. For the desired hardware realization, Handel-C enables the integration with VHDL and EDIF along with various synthesis and place-and-route tools. Our targeted compilation is EDIF using the DK1. The EDIF output is compiled to a bit format file using Xilinx place and route utility in the Xilinx ISE 5.1 package [10]. For downloading the bit file on the FPGA and performance analysis we use Visual C++ IDE with the RC-PP-1000 support libraries. The compilation steps are shown in Figures 2 and 3.

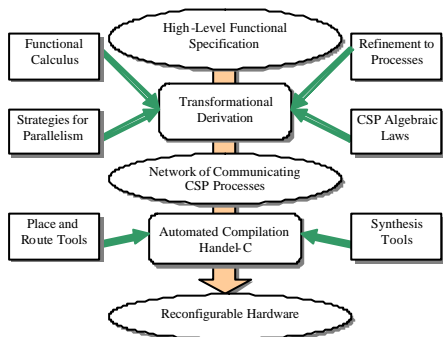


Figure 1. An overview of the transformational derivation and the hardware realization processes.

### BACKGROUND AND PREVIOUS WORK

Abdallah and Hawkins defined in [5] some constructs used in the development model. This looked in some depth at data refinement; the means of expressing structures in the specification as communication behaviour in the implementation. Firstly, streams are defined as a sequence of messages on a single channel, and correspond to a sequential method for communicating a list. Streams facilitate the communication of finite sequences and require some means of signalling the end of transmission (EOT).

Secondly, vectors of items are a means of communicating a list on more than one channel. The assumption is that there are as many channels in the vector as there are items in the list, such that each item is communicated on its own channel. Thirdly, vectors of streams are the parallel composition of  $n$  streams, each communicating a sublist independently as a stream. Each stream has its own end-of-transmission signal (EOT), and they can finish transmitting at different times. Lastly, streams of vectors is defined where a complete sublist is communicated in a single step.

### AIM AND MOTIVATION

The current research is of three main tasks, these tasks converge to a single goal. This goal is presenting and realizing a novel method for rapid prototyping of reconfigurable circuits. As previous work established the method for refining functions into parallel processes from high-level specifications, currently the case is investigating the translation of the processes so derived into Handel-C for mapping onto hardware.

The suggested three tasks could be summarized by, firstly, discussing different implementations of all the conceptual constructs affording rules for implementation. Secondly, completing a set of assisting utility program constructs. Thirdly, targeting engineering and industrial complex applications for testing the applicability of the model and the validity of the suggested favorable implementation rules. Broadening the area of application of the RDM is done by addressing different areas like information coding, computer and communications security, molecular modelling, etc...

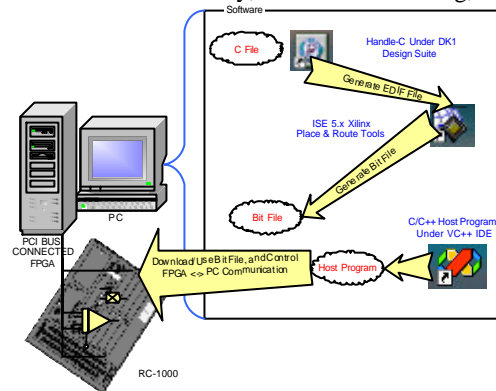


Figure 2. Hardware compilation steps.

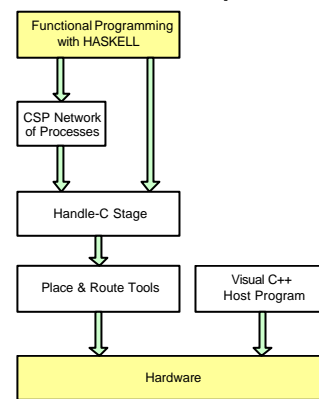


Figure 3. The RDM program development.

### DATA REFINEMENT

In the following we present some datatypes used for refinement.

#### Stream of Values

The stream is a purely sequential method of communicating a group of values. It comprises a sequence of messages on

a channel, with each message representing a value. Values are communicated one after the other. Assuming the stream is finite, after the last value has been communicated, the end of transmission (EOT) on a different channel will be signaled. Given some type A, a stream containing values of type A is denoted simply as  $\langle A \rangle$ . The Stream is shown in Figure 4.

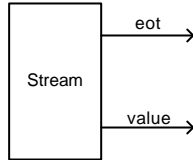


Figure 4. A Stream of values, the size of stream is not known in advance.

### Vector of n Values

Each item to be communicated by the vector will be dealt with independently in parallel. A vector refinement of a simple list of items will communicate the entire structure in a single. The vector is shown in Figure 5. Given some type A, a vector of length n, containing values of type A, is denoted as  $[A]_n$

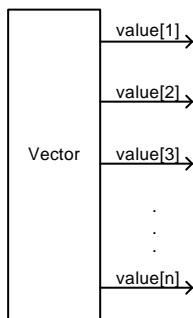


Figure 5. A Vector of size n.

### Refinement of a List of Lists

Whenever dealing with multi-dimensional data structures, for example, lists of lists, implementation options arise from differing compositions of our “primitive” data refinements - streams and vectors. Examples of the combined forms are the Stream of Streams, Streams of Vectors, Vectors of streams, and Vectors of Vectors (See Figure 6). These forms are denoted by:

$\langle S_1 S_2 \dots S_n \rangle$ ,  $[S_1 S_2 \dots S_n]_p$ ,  $\langle V_1 V_2 \dots V_n \rangle$ , and  $[V_1 V_2 \dots V_n]_p$  respectively.

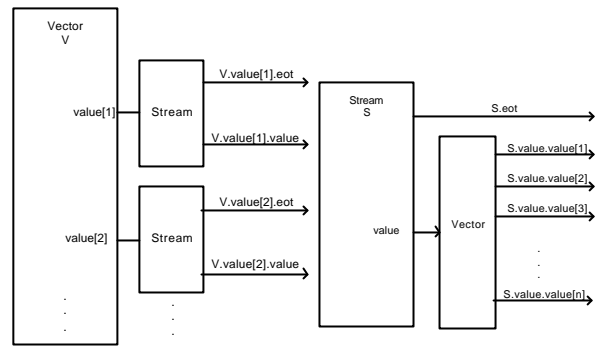


Figure 6. A Vector of Stream, and a Stream of Vectors.

## HIGH-ORDER-FUNCTIONS

Functional programming environments facilitate reusability through high-order-functions. Many algorithms can be built from components which are instances of some more general scheme. In this section we introduce the refinement of some high-order-functions detailed in [5] along with the refinement and implementation of the high-order-function zip-With.

Map applies a function to a list of items. Thus, in the functional setting, we have:

$$\text{map } f [x_1 \ x_2 \ \dots \ x_n] = [f(x_1) \ f(x_2) \ \dots \ f(x_n)]$$

Refining to CSP we have:

$$\text{VMAP}_n(F) =_{i=1}^{i=n} \parallel F[in_i/in, out_i/out, ]$$

Where, F is the refinement of f. A data parallel processes visualization of map  $\text{VMAP}_n(F)$  is shown in Figure 7.

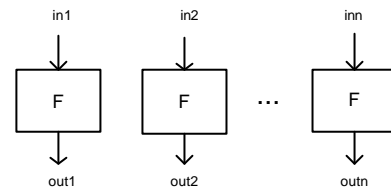


Figure 7. The Process  $\text{VMAP}_n(F)$ .

The fold family of functions is used to “reduce” a list by inserting a binary operator between each neighbouring pair of elements. The basic fold operator ( $/$ ) has no concept of direction and as such requires an associative binary operator to be well defined.

$$\oplus / [x_1 \ x_2 \ \dots \ x_n] = x_1 \oplus x_2 \oplus \dots \oplus x_n$$

Refining to CSP we have:

$$\text{VFOLD}_n(F) =_{i=1}^{i=n} \parallel F[c_i/out, c_{2i}/in_1, c_{2i+1}/in_2]$$

Where, F is the refinement of the operator  $\oplus$ . An instance of (VFOLD) is shown in Figure 9.

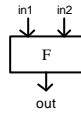


Figure 8. The Process F refinement of  $\oplus$  operator.

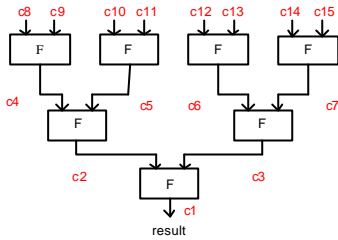


Figure 9. The Process VFOLD<sub>n</sub>(F), where F is the refinement of operator.

The high-order-function *zipWith* is used to zip two lists (taking one element from each list) with a certain operation see Figure 10. The high-order-function *zipWith* is specified as follows:

$$\begin{aligned} zipWith &:: (A \rightarrow B \rightarrow C) \rightarrow [A] \rightarrow [B] \rightarrow [C] \\ zipWith (\oplus) [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \\ &= [x_1 \oplus y_1, x_2 \oplus y_2, \dots, x_n \oplus y_n] \end{aligned}$$

To implement the data parallel version of this high-order-function we refine it to a process VZIP that takes two vector channels as input with their length and zips the two lists with a process F; F is a refined process from the function ( $\oplus$ ).

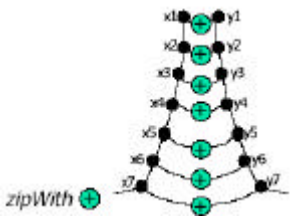


Figure 10. Zipping lists xs and ys with addition.

$$\begin{aligned} vzip (\oplus) &:: [A]_n \rightarrow [B]_n \rightarrow [C]_n \\ VZIP(\oplus) &=_{i=1}^n \parallel^{i=n} F[out_i/out, c_i/in_1, d_i/in_2] \end{aligned}$$

Where, F is the refinement of the operator ( $\oplus$ ). Figure 11 is a visualization of vector zipping with a process F. F could be addition, multiplication or any other operation on two lists.

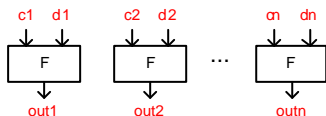


Figure 11. Process VZIP<sub>n</sub>(F), where F is the refinement of operator  $\oplus$ .

The Handle-C implementation is done with the macro *VectorZip*:

```
macro proc VectorZip(VectorSize, VectorChannel1In,
                    VectorChannel2In, ChannelOut, P)
{
    typedef (VectorSize) Index;
    par(Index = 0; Index < VectorSize; Index++)
    {
        P(VectorChannel1In[Index],
          VectorChannel2In[Index],
          ChannelOut[Index]);
    }
}
```

### CASE STUDY: SYNTHESIS OF PARALLEL MATRIX MULTIPLICATION ALGORITHM

In this section we demonstrate the use of the RDM to develop three designs for the refinement of the standard matrix multiplication algorithm. The purposes of this section are: 1) to give an example of applying the RDM, 2) to show the flexibility of design using the RDM, 3) preparing for the performance evaluation step which will lead to various benefits to the realization of the RDM. In each step the functional specification is introduced along with the CSP implementation. The reconfigurable hardware implementation will follow in the next section.

#### Functional Specification

A functional specification of matrix multiplication is formulated as a function *mmult* that takes two matrices as inputs and returns a matrix as a result. In this definition, we assume the first matrix is represented as a list of rows and the second matrix is represented as a list of columns.

$$\begin{aligned} mmult &:: [[Int]] \rightarrow [[Int]] \rightarrow [[Int]] \\ mmult \text{ ass } bss &= map(vmmult \text{ ass}) bss \\ vmmult &:: [Int] \rightarrow [[Int]] \rightarrow [Int] \\ vmmult \text{ bs } ass &= map(scalarp \text{ bs}) ass \\ scalarp &:: [Int] \rightarrow [Int] \rightarrow Int \\ scalarp \text{ as } bs &= sum (zipwithmul \text{ as } bs) \end{aligned}$$

$$\begin{aligned} sum &:: [Int] \rightarrow Int \\ sum \text{ rs} &= foldr1 (+) \text{ rs} \end{aligned}$$

$$\begin{aligned} zipwithmul &:: [Int] \rightarrow [Int] \rightarrow [Int] \\ zipwithmul \text{ as } bs &= zipwith(*) \text{ as } bs \end{aligned}$$

The suggested algorithm for multiplying two matrices is done by mapping the function (*vmmult ass*), named vector-matrix-multiplication with a list of lists *ass* (list of rows with

dimensions  $n \times m$  over the elements of list  $bss$  (list of columns whose dimensions are  $m \times k$ ).

The function  $vmmult$  takes two inputs  $ass$  and  $bs$  and returns a list (a column in the resulting matrix). The  $vmmult$  function maps the function ( $scalarp\ bs$ ) over the list  $ass$ .

The function  $scalarp$  defines the scalar product of two vectors. This function is the composition of the two functions  $zipwith$  ( $*$ ) and  $sum$ . The inputs are two lists  $as$  and  $bs$ , the output is a single value that is the result of zipping the two input lists and then folding the list elements over addition. The function  $sum$  adds up all the numbers in a given list.

The functional specification of  $zipwithmul$  takes two input lists  $as$  and  $bs$  and outputs a list of the same length. The  $i$ th element of the output list is the multiplication of the  $i$ th elements of the input lists.

### First Design

Recalling the high level description of  $mmult$ :

$$mmult :: [[Int]] \rightarrow [[Int]] \rightarrow [[Int]]$$

$$mmult\ ass\ bss = map(vmmult\ ass)\ bss$$

In this design we consider the refinement of the input  $bss$  and the output  $css$  as vectors of items where each item is a list.

The CSP implementation of the functional specification of the matrix multiplication algorithm  $mmult$  realizes this function as a process  $MMULT$ . The CSP process  $MMULT$  is interleaving with renaming of the process  $VMMULT$  (the refinement of  $vmmult$ ) for all columns of  $bss$ :

$$mmult(ass) :: \lfloor [Int] \rfloor_k \rightarrow \lfloor [Int] \rfloor_k$$

$$MMULT(ass) = VMAP_k(VMMULT(ass))$$

This design can be pictured as follows:

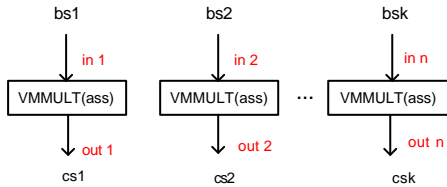


Figure 12. The Process MMULT.

The list  $ass$  is passed as an argument to each of the processes  $VMMULT(ass)$  in the above design. The list  $ass$  could be explicitly passed to the process  $VMMULT$  by exploiting the following algebraic identity:

$$VMMULT(ass) = PRD(ass) \triangleright VMMULT$$

The effect of applying this step to the previous design can be visualized as follows:

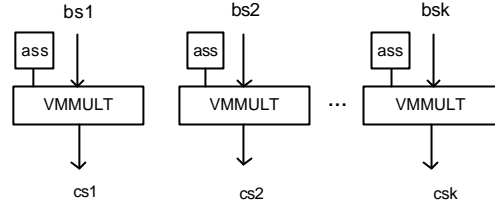


Figure 13. The Process MMULT, an alternative.

In this version the list  $ass$  is locally produced and fed to each process  $VMMULT$  in the vector. The effect of having  $k$  parallel copies of  $PRD(ass)$  communicating with  $k$  instances of  $VMMULT$  can be achieved by factorizing the process  $PRD(ass)$  and broadcasting its output to the relevant processing elements in the network. Applying this rule will result in a semantically equivalent version of  $MMULT$  which has a different layout. We have:

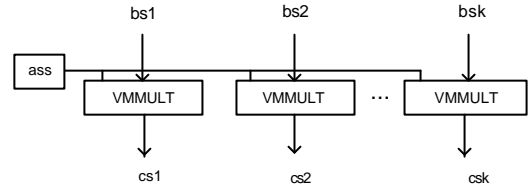


Figure 14. The Process MMULT, optimized implementation.

The formal rule that justifies the above transformation is:

$$VMAP_k( PRD(ass) \triangleright VMMULT ) =$$

$$BROADCAST_k(ass)[d / out] \triangleright_k VMAP_k(VMMULT)$$

Now we turn our attention to the refinement of the function  $vmmult$ .

$$vmmult :: [Int] \rightarrow [[Int]] \rightarrow [Int]$$

$$vmmult\ bs\ ass = map(scalarp\ bs)\ ass$$

Clearly,  $vmmult\ (bs)$  is a map pattern. Since  $map$  has two different implementations. We will consider them in turn. In this design, the CSP implementation realizes the function  $vmmult$  as a process  $VMMULT$  with a vector of items  $ass = [as_1, as_2, \dots, as_n]$  as input and a vector of items  $cs$  as output. By refining  $vscalarp(x)$  into  $VSCALARP(x)$ , the CSP implementation of  $vmmult\ (bs)$  is the off the shelf refinement of a vector map:

$$vmmult(bs) :: \lfloor [Int] \rfloor_m \rightarrow \lfloor [Int] \rfloor_n$$

$$VMMULT(bs) = VMAP_n(VSCALARP(bs))$$

By appealing to the same technique already used in the refinement of *VMMULT* we get:

$$VSCALARP(bs) = PRD(bs) \triangleright VSCALARP$$

This leads to a new design of *VMMULT*(*bs*):

$$VMAP_n(PRD(bs) \triangleright VSCALARP) =$$

$$BROADCAST_n(bs) \triangleright_n VSCALARP$$

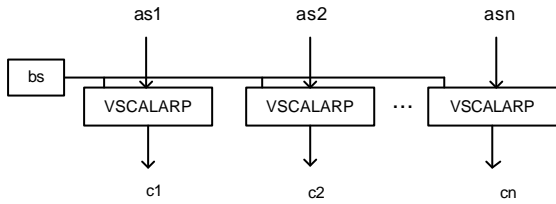


Figure 15. The Process *VMMULT*.

Figure 15 shows the process *VMMULT*. This step also shows clearly the replication of the process *VSCALARP*, which is an indicator for the later replication of the hardware implementation.

Figure 16 expands the main building block in Figure 13 by corresponding configuration in Figure 15. This gives a two dimensional visualization of the process *MMULT*.

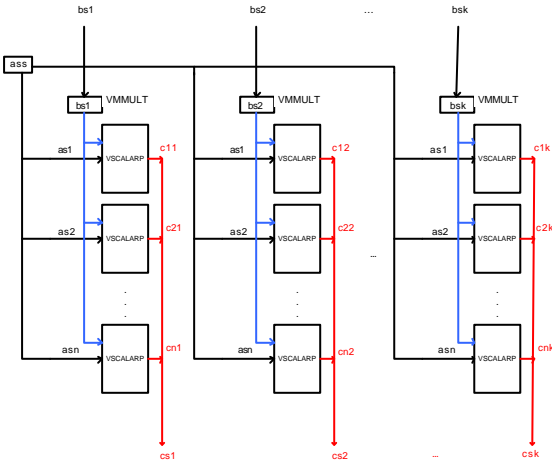


Figure 16. Process *MMULT*.

The final step is give a detailed definition of the new basic building block *VSCALARP* corresponding to a CSP refinement of the function:

$$vscalarp :: [Int] \rightarrow [Int] \rightarrow Int$$

$$vscalarp \ as \ bs = sum \ (zipwithmul \ as \ bs)$$

This function can be refined as the piping of two processes *VZIP*(*MUL*) and *VFOLD*(*ADD*) corresponding to refinements of the functions *sum* and *zipwithmul* respectively.

$$vscalarp :: \lfloor [Int] \rfloor_m \rightarrow \lfloor [Int] \rfloor_m \rightarrow Int$$

$$VSCALARP = VZIP_m(MUL) \gg_m VFOLD_m(ADD)$$

This description is depicted in Figure 17.

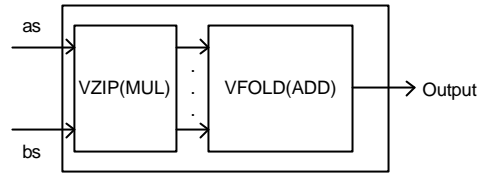


Figure 17. *VSCALARP* as a piping of two processes

For completeness, the CSP implementations of the simple addition and multiplication functions are:

$$ADD = (in_1 ? a \rightarrow SKIP \parallel in_2 ? b \rightarrow SKIP); out! a + b \rightarrow SKIP$$

$$MUL = (in_1 ? a \rightarrow SKIP \parallel in_2 ? b \rightarrow SKIP); out! a * b \rightarrow SKIP$$

## Second Design

Recall the function *vmmult*:

$$vmmult :: [Int] \rightarrow \lfloor [Int] \rfloor \rightarrow [Int]$$

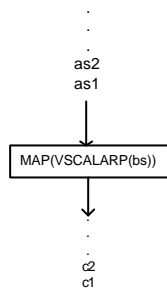
$$vmmult \ bs \ ass = map(scalarp \ bs) \ ass$$

We view the input list *ass* as a stream of values  $ass = \langle as_1, as_2, \dots, as_n \rangle$ , and the output list as a stream of values as well. The CSP refinement of *vmmult*(*bs*) is directly obtained from the off the shelf stream based implementation of the higher order function *map*:

$$vmmult(bs) :: \langle [Int] \rangle \rightarrow \langle [Int] \rangle$$

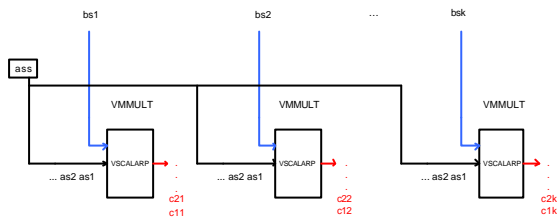
$$VMMULT(bs) = MAP(VSCALARP(bs))$$

Figure 18 shows the new version of the process *VMMULT*. This step also shows clearly that there is no more replication of the process *VSCALARP*, which is an indicator for the later reduction in use of hardware resources.



**Figure 18. The Process VMMULT the input and output refined as streams of values.**

Keeping the refinement of the remaining functions the same, *MMULT* process looks as in Figure 19.



**Figure 19. Process MMULT.**

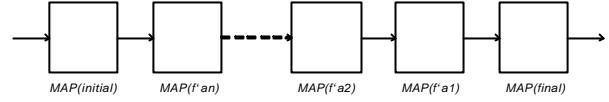
### Third Design

In this design we will make use of pipelined parallelism, which is a very effective means for achieving efficiency in numerous algorithms. Pipelined parallelism in general is much harder to detect than data parallelism. For this task we use the function decomposition strategy found in [9], which aims at exhibiting pipelined parallelism in functional programs. Recalling the transformation rule, consider:

$$\begin{aligned}
 &spec :: A \rightarrow B; h :: [T] \rightarrow A \rightarrow B; m :: [T]; e :: B; \\
 &f :: T \rightarrow A \rightarrow B \rightarrow B \\
 &spec = map(h \ m) \\
 &h \ [] \ x = e \\
 &h \ (a : s) \ x = f \ a \ x \ (h \ s \ x) \\
 &----- \updownarrow ----- < \\
 &spec = (o) / [final*] ++ (map \circ f' * m) + [initial*] \\
 &f' a \langle x, y \rangle = \langle x, f \ a \ x \ y \rangle \\
 &initial \ x = \langle x, e \rangle \\
 &final \ \langle x, y \rangle = y
 \end{aligned}$$

Accordingly, the CSP network SPEC, which refines the functional specification  $map(h \ m)$  is shown in Figure 20 Using the above rule, we decompose the definition of the function *mmult*.

$$\begin{aligned}
 &mmult \ ass \ bss = map(vmmult \ ass) \ bss \\
 &vmmult \ [] \ bs = [] \\
 &vmmult \ (a : ass) \ bs = \\
 &\quad vscalarp \ a \ bs : vmmult \ ass \ bs
 \end{aligned}$$



**Figure 20. Process SPEC.**

The recursive function in this case is *vmmult*. The value to be passed to the next stage of the pipe is a tuple, whose first is the input vector and second is result of applying *vscalarp* on the input vector from the matrix *bss* and the present argument from the matrix *ass*. The pipelined network is shown in Figure 21. In this design, the matrix *bss* is input to the network as a stream of vectors (columns)  $\langle bs_1, bs_2, \dots, bs_n \rangle$ . The matrix *ass* vectors (row by row) are passed as arguments in the pipe stages. The result is considered as a stream of streams  $\langle cs_1, cs_2, \dots, cs_n \rangle$  The first result to appear from the network is the output stream (column)  $\langle cs_n \rangle$  corresponding to the first input vector  $bs_n$ . The resultant matrix is to appear then step by step.

### Fourth Design

This design makes use of an optimization of the previous design. In this case, the input matrix *bss* is refined as a stream of vectors  $\langle bs_1, bs_2, \dots, bs_n \rangle$  and the matrix *ass* is refined as arguments in the pipeline stages. The output is taken from each stage as a vector of streams as shown in Figure 22. For instance, the first vector to be fed in to the pipe is the vector  $bs_n$ , accordingly the output vector of streams will give the first resultant vector  $cs_n$ .

### General Evaluation

Generally, the suggested algorithms inherit all the advantages from the rapid development method applied. The key issue in the adopted model is the production of engineering efficient, scalable, reusable, and correct solutions by construct as opposed to trial and testing. Correctness is ensured by construction through the functional specification step. Reusability at its best appears when using “off-the-shelf” building blocks of code. High-order-functions serve as the basic building blocks to construct the parallel algorithm. The usage of message passing rather than shared memory in the rapid development model affords scalability. The targeted hardware contributes to the adopted model by being faster and smaller than truly general-purpose hardware such as a workstation. Also, compared to an ASIC, it has smaller non-recurring engineering (NRE) costs, for it can be easily reconfigured.

The next direct step is the Handle-C implementation step where these algorithms, compiled and mapped onto the FPGA for testing and performance analysis.

## CONCLUSION

In this paper we have presented an extension to the implementation stage of a methodology that can take intuitive, high level specifications of algorithms. These algorithms are specified in the functional style and then refined into efficient, parallel implementations to be later compiled in Handel-C and mapped onto reconfigurable hardware circuits. The targeted hardware is the RC-1000 Virtex 2000E FPGA from Celoxica. A case study for matrix multiplication is presented where several radically different designs are systematically derived from the common specification. At the time of writing this paper, we did not have the performance data corresponding to the outlined designs. Future work will include broadening the area of application for the RDM to cover algorithms in digital coding, molecular modeling, DNA matching and cryptography.

## References

- [1] A. E. Abdallah, *Derivation of Parallel Algorithms from Functional Specifications to CSP Processes*, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS 947, (Springer Verlag, 1995) pp 67-96
- [2] A. E. Abdallah, *Functional Process Modelling*. In K Hammond and G. Micheals on (eds), *Research Directions in Parallel Functional Programming*, (Springer Verlag, October 1999). pp339-360
- [3] I. Damaj, A. E. Abdallah, *Reconfigurable Hardware Synthesis for Parallel Cyclic Coding Algorithms*, Proceedings of PGNET 2002 Liverpool, United Kingdom (June 17-18 2002) v. 2 pp. 104-109
- [4] I. Damaj, and A. E. Abdallah, *Synthesis of Massively Parallel Algorithms & Their Mapping onto Reconfigurable Systems*, Poster presentation & Abstract in the proceedings of IEEE/EPSC PREP 2002. Nottingham University, Nottingham-United kingdom (April 17th-19th 2002) v. 1 pp. 50-51
- [5] J Hawkins and A. E. Abdallah, *Calculational Design of Special Purpose Parallel Algorithms*, Proceedings of the 7th IEEE International Conference on Electronics, Circuits and Systems. Beirut-Lebanon, (December 17th-20th 2000)
- [6] Torkelsson K., J. Ditmar, *Header Compression in Handel-C An Internet Application and A New Design Language*, Euromicro Symposium on Digital Systems Design (2001) 2 -7
- [7] I. Damaj, Diab H., *Performance Evaluation of Linear Algebraic Functions Using Reconfigurable Computing*, *International Journal of Super Computing*, Kluwer (2003) i. 1 v. 24 pp. 91-107
- [8] Page Ian - Algorithm, *Logarithmic Greatest Common Divisor Example in Handel-C*, Embedded Solutions, (May 1998)
- [9] Sullivan Mark, *Multiplication substitutions enable fast algorithm implementations in FPGAs*, Personal Engineering, (June 1995).
- [10] Xilinx, *The Programmable Gate Array Data Book*, Xilinx Inc., San Jose, California, 1991
- [11] Handel-C *Documentation*, Available from Celoxica (<http://www.celoxica.com/>).
- [12] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, (1985).

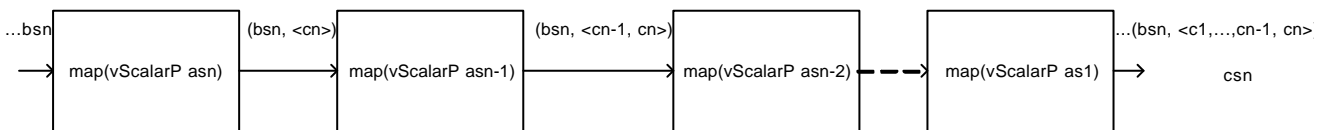


Figure 21. Matrix multiplication algorithm implementation as a pipeline of processes.

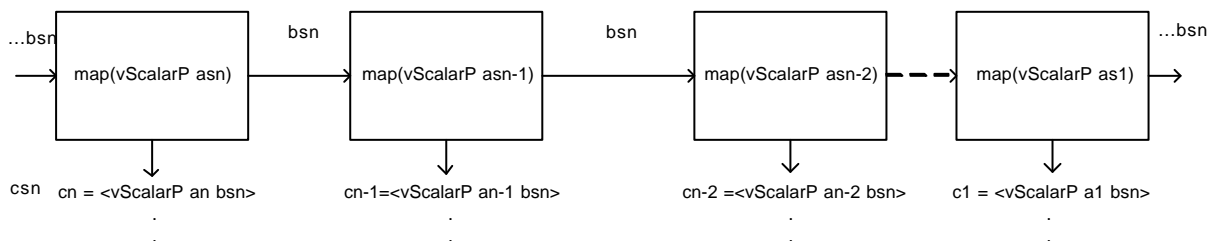


Figure 22. Matrix multiplication algorithm implementation as an optimized pipeline of processes.