# The Abstraction Mechanisms of Vector Pascal

Paul Cockshott, University of Glasgow, Imaging Faraday Partnership

November 21, 2001

**Abstract**

Vector Pascal is a language designed to support elegant and efficient expression of algorithms using the SIMD model of computation. It imports into Pascal abstraction mechanisms derived from functional languages having their origins in APL. In particular it extends all operators to work on vectors of data. The type system is extended to handle pixels and dimensional analysis. Code generation is via the ILCG system that allows retargeting to multiple different SIMD instructionsets based on formal descrition of the instructionset semantics.

## 1 Introduction

The introduction of SIMD instruction sets[10][1][21][11] to Personal Computers potentially provides substantial performance increases, but the ability of most programmers to harness this performance is held back by two factors. The first is the limited availability of compilers that make effective use of these instructionsets in a machine independent manner. This remains the case despite the research efforts to develop compilers for multi-media instructionsets[6][19][18][22]. The second is the fact that most popular programming languages were designed on the word at a time model of the classic von Neuman computer rather than the SIMD model.

The functions of a high level programming language (HLPL) are:

1. To lighten the mental load of the programmer.
2. To help obviate program errors.
3. To allow programs to run fast by making efficient use of hardware.

The abstraction mechanisms of HLPLs make the thinking through of a problem, and its solution by algorithmical means, easier. They allow a problem to be thought of as one concrete instance in a more general family, and, as such, reveal that it can be solved by some very general, powerful and well understood algorithmic techniques. The type and abstraction mechanisms of HLPLs condition the thought processes of their users.

If we consider popular imperative languages like Pascal, C, Java and Delphi we see that:

- They are structured around a model of computation in which operations take place on a single value at a time. The abstraction mechanism they use for bulk data operations is serialisation using **for** loops. This reflects the word at a time model of earlier computers.
- Their type system provides inadequate support for pixels and images.

These limitations make it is, significantly harder than it need be to write image processing programs that will use SIMD parallelism.

Vector Pascal aims to provide an efficient and elegant notation for programmers using Multi-Media enhanced CPUs. In doing so it borrows concepts for expressing data parallelism that have a long history, dating back to Iverson's work on APL in the early '60s[13]. By an elegant algorithm I mean one which is expressed as concisely as possible. Elegance is a goal that one approaches asymtotically, approaching but never attaining[5]. APL and J allow the construction of very elegant programs, but at a cost. An inevitable consequence of elegance is the loss of redundancy. APL programs are as concise, or even more concise than conventional mathematical notation[14] and use a special characterset. This makes them hard for the uninitiated to understand. J attempts to remedy this by restricting itself to the ASCII characterset, but still looks dauntingly unfamiliar to programmers brought up on more conventional languages. Both APL and J are interpretive which makes them ill suited to many of the applications for which SIMD speed is required. The aim of Vector Pascal is to provide the conceptual gains of Iverson's notation within a framework familiar to imperative programmers.

Pascal[17]was chosen as a base language over the alternatives of C and Java. C was rejected because notations like `x+y` for `x` and `y` declared as `int x[4],y[4]`, already have the meaning of adding the addresses of the arrays together. Java was rejected because of the difficulty of efficiently transmitting data parallel operations via its intermediate code to a just in time code generator.

# 2 Array mechanisms for data parallelism

Vector Pascal extends the array type mechanism of Pascal to provide better support for data parallel programming in general, and SIMD image processing in particular. Data parallel programming can be built up from certain underlying abstractions[8]:

- operations on whole arrays
- array slicing
- reduction operations

We will first consider these in general before moving on to look at how they are supported in Vector Pascal.

## 2.1 Operations on whole arrays

The basic *conceptual* mechanism is the *map*, which takes an operator and and a source array ( or pair of arrays ) and produces a result array by mapping the source(s) under the operator. Let us denote the type of an arry of $T$as $T[]$. Then if we have a binary operator $\omega : (T \otimes T) \rightarrow T$, we automatically have an operator $\omega : (T[] \otimes T[]) \rightarrow T[]$ . Thus if $x, y$ are arrays of integers $k = x + y$ is the array of integers where $k_i = x_i + y_i$:

| 3 | 5 | 9 | = | 2 | 3 | 5 | + | 1 | 2 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|

Similarly if we have a unary operator $\mu$:( T→T) then we automatically have an operator $\mu$:(T[]→T[]). Thus $z$ =sqr$(x)$ is the array where $z_i = x_i^2$:

| 4 | 9 | 25 | = sqr( | 2 | 3 | 5 | ) |
|---|---|---|---|---|---|---|---|

The map replaces the *serialisation* or *for loop* abstraction of classical imperative languages. The map concept is simple, and maps over lists are widely used in functional programming. For array based languages there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but Iverson[13] provided a consistent treatment of these. Recent languages built round this model are J, an interpretive language[15][3][16], High Performance Fortran[8] and F[20] a modern Fortran subset. In principle though any language with array types can be extended in a similar way. The map approach to data parallelism is machine independent. Depending on the target machine, a compiler can output sequential, SIMD, or MIMD code to handle it.

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 2 | 4 | 8 |
| 1 | 2 | 4 | 16 |
| 1 | 2 | 8 | 512 |

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 2 | 4 | 8 |
| 1 | 2 | 4 | 16 |
| 1 | 2 | 8 | 512 |

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 2 | 4 | 8 |
| 1 | 2 | 4 | 16 |
| 1 | 2 | 8 | 512 |

Figure 1: Different ways of slicing the same array

### 2.2 Array slicing

It is advantageous to be able to specify sections of arrays as values in expression. The sections may be rows or columns in a matrix, a rectangular sub-range of the elements of an array, as shown in figure 1. In image processing such rectangular sub regions of pixel arrays are called regions of interest. It may also be desirable to provide matrix diagonals[23].

### 2.3 Reduction operations

In a reduction operation, a dyadic operator injected between the elements of a vector, the rows or columns of a matrix etc, produces a result of lower rank. Examples would be the forming the sum of a table or finding the maximum or minimum of a table. So one could use + to reduce | 1 | 2 | 4 | 8 | to 1+2+4+8=15

## 3 Data parallelism in Vector Pascal

Vector Pascal incorporates Iverson's approach to data parallelism. Its aim is to provide a notation that allows the natural and elegant expression of data parallel algorithms within a base language that is already familiar to a considerable body of programmers and combine this with modern compilation techniques.

### 3.1 Assignment maps

Standard Pascal allows assignement of whole arrays. Vector Pascal extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. Given

**r0:real; r1:array[0..7] of real; r2:array[0..7,0..7] of real**

then we can write

r**1:= r2[3]; { supported in standard Pascal }**
**r1:= 1/2; { assign 0.5 to each element of r1 }**
**r2:= r1*3; { assign 1.5 to every element of r2}**
**r2[3]:= r2[4]+r1 ; { self explanatory }**

The variable on the left hand side of an assignment defines an array context within which expressions on the right hand side are evaluated. Each array context has a rank given by the number of dimensions of the array on the left hand side. A scalar variable has rank 0. Variables occuring in expressions with an array context of rank *r* must have *r* or fewer dimensions. The *n* bounds of any *n* dimensional array variable, with $n \leq r$ occuring within an expression evaluated in an array context of rank *r* must match with the rightmost *n* bounds of the array on the left hand side of the assignment statement.

Where a variable is of lower rank than its array context, the variable is replicated to fill the array context. This is shown in examples 2 and 3 above. Because the rank of any assignment is constrained by the variable on the left hand side, no temporary arrays, other than machine registers, need be allocated to store the intermediate array results of expressions.

Maps are implicitly and promiscously defined on both monadic operators and unary functions. If **f** is a function or unary operator mapping from type **r** to type **t** then if **x** is an array of **r** then **a:=f(x)** assigns an array of **t** such that **a[i]=f(x[i])**.

Functions can return any data type whose size is known at compile time, including arrays and records. A consistent copying semantics is used.

## 3.2 Operator Reduction

Maps take operators and arrays and deliver array results. The *reduction* abstraction takes a dyadic operator and an array and returns a scalar result. It is denoted by the functional form \. Thus if a is an array, \+a denotes the sum over the array. More generally $\backslash \Phi x$ for some dyadic operator $\Phi$ means $x_0 \Phi (x_1 \Phi .. (x_n \Phi \iota))$ where $\iota$ is the identity element for the operator and the type. Thus we can write \+ for $\sum$, * for $\prod$ etc. The dot product of two vectors can thus be written as

**x:=\+(y*z);**

instead of

**x:=0;**

**for i:=0 to n do x:= x+ y[i]*z[i];**

A reduction operation takes an argument of rank *r* and returns an argument of rank *r-1* except in the case where its argument is or rank 0, in which case it acts as the identity operation. Reduction is always performed along the last array dimension of its argument.

## 3.3 Operations on implicit indices

Assignment maps and reductions involve implicit indices. It can be useful to have access to these.

### 3.3.1 iota

The syntactic form **iota** *i* returns the *i*th current implicit index. Thus given the definitions

**v1:array[1..3]of integer; v2:array[0..4]of integer;**

the program fragment **v1:=iota 0; v2:=iota 0 *2;**

would set v1 and v2 as follows:

```
v1= 1 2 3
v2= 0 2 4 6 8
```

whilst given the definitions

**m1:array[1..3,0..4] of integer;m2:array[0..4,1..3]of integer;**

then the program fragment

**m2:= iota 0 +2*iota 1;**

would set m2

```
m2=
2 4 6
3 5 7
4 6 8
5 7 9
6 8 10
```

The argument to iota must be an integer known at compile time within the range of implicit indices in the current context.

### 3.3.2 trans

The syntactic form **trans***x* transposes a vector [1]matrix, or tensor. It achieves this by cyclic rotation of the implicit indices. Thus if **trans** *e* is evaluated in a context with implicit indices

**iota** *0..* **iota** *n*

then the expression e is evaluated in a context with implicit indices

**iota**'*0..* **iota**'*n*

where

**iota**'*x* = **iota** ( (*x+1*)**mod** *n+1*)

It should be noted that transposition is generalised to arrays of rank greater than 2.

**Examples** Given the defintions used above in section 3.3.1, the program fragment:

**m1:= (trans v1)*v2;**

**m2 := trans m1;**

will set m1 and m2:

```
m1=
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
m2=
0 0 0
2 4 6
4 8 12
6 12 18
8 16 24
```

# 4 Unary operators

A unary expression is formed by applying a unary operator to another unary or primary expression. The unary operators supported are **+, -, *, /, div, not, round, sqrt, sin, cos, tan, abs, ln, ord, chr, succ, pred** and **@**.

Thus the following are valid unary expressions**: -1, +b, not true, sqrt abs x, sin theta.** In standard Pascal some of these operators are treated as functions,. Syntactically this means that their arguments must be enclosed in brackets, as in **sin(theta)**. This usage remains syntactically correct in Vector Pascal.

The dyadic operators **+, -, *, /, div** are all extended to unary context by the insertion of an identity element under the operation. This is a generalisation of the monadic use of + and - in standard pascal where **+a=0+a** and **-a = 0-a** with 0 being the additive identity, so too **/2 = 1/2** with 1 the multiplicative identity. For sets the notation **-s** means the complement of the set **s**. The implicit value inserted are given below.

| type | operators | identity elem |
|---|---|---|
| number | +,- | 0 |
| set | + | empty set |
| set | -,* | fullset |
| number | *,/ ,div,mod | 1 |
| boolean | and | true |
| boolean | or | false |

[1]Note that **trans** is not strictly speaking an operator, as there exists no Pascal type corresponding to a column vector.

#### 4.0.3 Dyadic Operations

Dyadic operators supported are **+, +:, -:, -, *, /, div, mod , **, pow,** <, >, >=, <=, =, <>, **shr, shl, and, or, in, min, max**. All of these are consistently extended to operate over arrays. The operators **, pow denote exponentiation and raising to an integer power as in ISO Extended Pascal. The operators +: and -: exist to support saturated arithmetic on bytes as supported by the MMX instructionset.

# 5 Extensions to the Pascal Type System

## 5.1 Dynamic Arrays

A dynamic array is an array whose bounds are determined at run time. Operations on dynamic arrays are essential in general purpose image processing software since the size of an image loaded from a file may not be known at compile time.

Pascal 90[12] introduced the notion of schematic or parameterised types as a means of creating dynamic arrays. Thus where **r** is some integral type one can write

**type z(a,b:r)=array[a..b] of t;**

If **p:^z**, then

**new(p,n,m)**

where **n,m:r** initialises **p** to point to an array of bounds **n..m**. The bounds of the array can then be accessed as **p^.a, p^.b**. Vector Pascal incorporates this notation from Pascal 90 for dynamic arrays [2].

## 5.2 Indexed Ranges

Image processing applications often have to deal with regions of interest, rectangular sub-images within a larger image. Vector Pascal extends the array abstraction to define sub-ranges of arrays. A sub-range of an array variable are denoted by the variable followed by a range expression in brackets.

The expressions within the range expression must conform to the index type of the array variable. The type of a range expression **a[i..j]** where **a: array[p..q] of t** is **array[0..j-i] of t.**

Examples

**dataset[i..i+2]:=blank;**

**twoDdata[2..3,5..6]:=twoDdata[4..5,11..12]*0.5;**

Subranges may be passed in as actual parameters to procedures whose corresponding formal parameters are declared as variables of a schematic type. Hence given the following declarations:

**type image(miny,maxy,minx,maxx:integer)=array[miny..maxy,minx..maxx] of pixel;**

**procedure invert(var im:image);begin im:= - im; end;**

**var screen:array[0..319,0..199] of pixel;**

then the following statement would be valid:

**invert(screen[40..60,20..30]);**

An array may be indexed by another array. If **x:array[t0] of t1** and **y:array[t1] of t2**, then **y[x]** denotes the virtual array of type **array[t0] of t2** such that **y[x][i]=y[x[i]]**. This construct is useful for performing permutations.

[2] It should be noted that the use of schematic array types preserves the semantic distinction present in Standard Pascal, between assigning a pointer to an array, and assigining an array itself. This distinction is confused with the "open array" construct allowed by Delphi and Free Pascal.

**Example** Given the declarations

**const perm:array[0..3] of integer=(3,1,2,0);**
**var ma,m0:array[0..3] of integer;**
then the statements
**m0:= (iota 0)+1;**
**ma:=m0[perm];**
would set the variables such that

```
m0= 1  2  3  4
perm= 3 1 2 0
ma= 4 2 3 1
```

## 5.3 Pixels

Standard Pascal is a strongly typed language, with a comparatively rich collection of type abstractions : enumeration, set formation, sub-ranging, array formation, cartesian product[3] and unioning[4]. However as an image processing language it suffers from the disadvantage that no support is provided for pixels and images. Given the vintage of the language this is not surprising and, it may be thought, this deficiency can be readily overcome using existing language features. Can pixels not be defined as a subrange 0..255 of the integers, and images modeled as two dimensional arrays?

They can be, and are so defined in many applications, but such an approach throws onto the programmer the whole burden of handling the complexities of limited precision arithmetic. Among the problems are:

1. When doing image processing it is frequently necessary to subtract one image from another, or to create negatives of an image. Subtraction and negation implies that pixels should be able to take on negative values.

2. When adding pixels using limited precision arithmetic, addition is nonmontonic due to wrap-round. Pixel values of $100+200 = 300$, which in 8 bit precision is truncated to 44 a value darker than either of the starting values. A similar problem can arise with subtraction, for instance $100 - 200 = 156$ in 8 bit unsigned arithmetic.

3. When multiplying 8 bit numbers, as one does in executing a convolution kernel, one has to enlarge the representation and shift down by an appropriate amount to stay within range.

These and similar problems make the coding of image filters a skilled task. The difficulty arises because one is using an inappropriate conceptual representation of pixels.

The *conceptual model* of pixels in Vector Pascal is that they are real numbers in the range $-1.0..1.0$. This representation overcomes the aforementioned difficulties. As a signed representation it lends itself to subtraction. As an unbiased representation, it makes the adjustment of contrast easier, one can reduce contrast 50% simply by multiplying an image by 0.5 [5]. Assignment to pixel variables in Vector Pascal is defined to be saturating - real numbers outside the range $-1..1$ are clipped to it. The multiplications involved in convolution operations fall naturally into place.

The *implementation model* of pixels used in Vector Pascal is of 8 bit signed integers treated as fixed point binary fractions. All the conversions necessary to preserve the monotonicity of addition, the range of multiplication etc, are delegated to the code generator which, where possible, will implement the semantics using efficient, saturated multi-media arithmetic instructions.

[3]The **record** construct.

[4]The **case** construct in records.

[5]When pixels are represented as integers in the range 0..255, a 50% contrast reduction has to be expressed as $((p - 128) \div 2) + 128$.

### 5.4 Dimensioned Types

Dimensional analysis is familiar to scientists and engineers and provides a routine check on the sanity of mathematical expressions. Dimensions can not be expressed in the otherwise rigourous type system of standard Pascal, but they are a useful protection against the sort of programming confusion between imperial and metric units that caused the demise of a recent Mars probe. They provide a means by which floating point types can be specialised to represent dimensioned numbers as is required in physics calculations. For example:

**kms =(mass,distance,time);**
**meter=real of distance;**
**kilo=real of mass;**
**second=real of time;**
**newton=real of mass * distance * time POW -2;**
**meterpersecond = real of distance *time POW -1;**

The identifier must be a member of a scalar type, and that scalar type is then refered to as the *basis space* of the dimensioned type. The identifiers of the basis space are refered to as the dimensions of the dimensioned type. Associated with each dimension of a dimensioned type there is an integer number refered to as the power of that dimension. This is either introduced explicitly at type declaration time, or determined implicitly for the dimensional type of expressions.

A value of a dimensioned type is a dimensioned value. Let $\log_d t$ of a dimensioned type $t$ be the power to which the dimension $d$ of type $t$ is raised. Thus for $t =$newton in the example above, and $d =$time, $\log_d t = -2$

If $x$ and $y$ are values of dimensioned types $t_x$and $t_y$respectively, then the following operators are only permissible if $t_x = t_y$: +, - ,<, >, =, <=, >=. For + and -, the dimensional type of the result is the same as that of the arguments. The operations. The operations *, / are permited if the types $t_x$and $t_y$ share the same basis space, or if the basis space of one of the types is a subrange of the basis space of the other.

The operation **POW** is permited between dimensioned types and integers.

#### Dimension deduction rules

1. If $x = y * z$ for $x : t_1, y : t_2, z : t_3$ with basis space $B$ then $\forall_{d \in B} \log_d t_1 = \log_d t_2 + \log_d t_3$.
2. If $x = y/z$ for $x : t_1, y : t_2, z : t_3$ with basis space $B$ then $\forall_{d \in B} \log_d t_1 = \log_d t_2 - \log_d t_3$.
3. If $x = y$ **POW** $z$ for $x : t_1, y : t_2, z : integer$ with basis space for $t_2$, $B$ then $\forall_{d \in B} \log_d t_1 = \log_d t_2 \times z$.

## 6 Implementation

The Vector Pascal compiler is implemented in Java. It uses the ILCG[7](Intermediate Language for Code Generators) portable code generator generator system. A Vector Pascal program is translated into an abstract semantic tree implemented as a Java datastructure. The tree is passed to a machine generated Java class corresponding to the code generator for the target machine. Code generator classes currently exist for the Intel 486, Pentium with MMX, and P3 and also the AMD K6. Output is assembler code which is assembled using the NASM assembler and linked using the gcc loader. Vector Pascal currently runs under Windows98 , Windows2000 and Linux. Separate compilation using Turbo Pascal style units is supported. C calling conventions allow use of existing libraries.

The code generators follow the pattern matching approach described in[2][4]and [9], and are automatically generated from machine specifications written in ILCG . ILCG is

a strongly typed language which supports vector data types and the mapping of operators over vectors. It is well suited to describing SIMD instructionsets. The code generator classes export from their interfaces details about the degree of parallelism supported for each data-type. This is used by the front end compiler to iterate over arrays longer than those supported by the underlying machine. Where supported parallelism is unitary, this defaults to iteration over the whole array.

Selection of target machines is by a compile time switch which causes the appropriate code generator class to be dynamically loaded.

# 7 Conclusions

Vector Pascal provides a new approach to providing a programming environment for multimedia instructionsets. It borrows abstraction mechanisms that have a long history of sucessfull use in interpretive programming languages, combining these with modern compiler techniques to target SIMD instructionsets. It provides a uniform source language that can target multiple different processors without the programmer having to think about the target machine. Use of Java as the implementation language aids portability of the compiler accross operating systems. Work is underway to port the BLAS library to Vector Pascal, and to develop an IDE and literate programming system for it.

# References


[1] Advanced Micro Devices, 3DNow! Technology Manual, 1999.

[2] Aho, A.V., Ganapathi, M, TJiang S.W.K., Code Generation Using Tree Matching and Dynamic Programming, ACM Trans, Programming Languages and Systems 11, no.4, 1989, pp.491-516.

[3] Burke, Chris, J User Manual, ISI, 1995.

[4] Cattell R. G. G., Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems, 2(2), pp. 173-190, April 1980.

[5] Chaitin. G., Elegant Lisp Programs, in The Limits of Mathematics, pp. 29-56, Springer, 1997.

[6] Cheong, G., and Lam, M., An Optimizer for Multimedia Instruction Sets, 2nd SUIF Workshop, Stanford University, August 1997.

[7] Cockshott, Paul, Direct Compilation of High Level Languages for Multi-media Instruction-sets, Department of Computer Science, University of Glasgow, Nov 2000.

[8] Ewing, A. K., Richardson, H., Simpson, A. D., Kulkarni, R., Writing Data Parallel Programs with High Performance Fortran, Edinburgh Parallel Computing Centre, Ver 1.3.1.

[9] Susan L. Graham, Table Driven Code Generation, IEEE Computer, Vol 13, No. 8, August 1980, pp 25..37.

[10] Intel, Intel Architecture Software Developers Manual Volumes 1 and 2, 1999.

[11] Intel, Willamette Processor Software Developer's Guide, February, 2000.

[12] ISO, Extended Pascal ISO 10206:1990, 1991.

[13] Iverson K. E., A Programming Language, John Wiley & Sons, Inc., New York (1962), p. 16.

[14] Iverson, K. E. . Notation as a tool of thought. Communications of the ACM, 23(8), 444-465, 1980.

[15] Iverson K. E, A personal View of APL, IBM Systems Journal, Vol 30, No 4, 1991.

[16] Iverson, Kenneth E., J Introduction and Dictionary, Iverson Software Inc. (ISI), Toronto, Ontario, 1995.

[17] Jensen K., and Wirth N., Pascal User Manual and Report, Springer, 1978.

[18] Krall, A., and Lelait, S., Compilation Techniques for Multimedia Processors, International Journal of Parallel Programming, Vol. 28, No. 4, pp 347-361, 2000.

[19] Leupers, R., Compiler Optimization for Media Processors, EMMSEC 99/Sweden 1999.

[20] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.

[21] Peleg, A., Wilke S., Weiser U., Intel MMX for Multimedia PCs, Comm. ACM, vol 40, no. 1 1997.

[22] Srereman, N., and Govindarajan, G., A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol. 28, No. 4, pp 363-400, 2000.

[23] van der Meulen, S. G.,ALGOL 68 Might Have Beens, SIGPLAN notices Vol. 12, No. 6, 1977.