Vector Pascal

Paul Cockshott, University of Glasgow

September 17, 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 features derived from the functional languages APL and J, in particular the extension of all operators to work on vectors of data. The type system is extended to handle 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.

Introduction

The introduction of SIMD instruction sets[9][1][20][10] 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][18][17][21]. 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. Whilst processor architectures are moving towards greater levels of parallelism, the most widely used programming languages like C, Java and Delphi are structured around a model of computation in which operations take place on a single value at a time. This was appropriate when processors worked this way, but has become an impediment to programmers seeking to make use of the performance offered by multi-media instructionsets. These problems mean that it is, in practice, significantly harder to write a program that will use SIMD features than it is to write a conventional program.

Vector Pascal aims to provide an efficient and concise 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[12].

If we write the type of an array of Tas type $T[]$. Then if we have a binary operator $\omega : (T \otimes T) \rightarrow T$, in languages derived from APL 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 $k_i = x_i + y_i$.

The basic concept is simple, there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but Iverson provides a consistent treatment of these. The most recent languages to be built round this model are J, an interpretive language[14][3][15], and F[19] a modernised Fortran. In principle though any language with array types can be extended in a similar way. Iverson's approach to data parallelism is machine independent. It can be implemented using scalar instructions or using the SIMD model. The only difference is speed.

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.

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[13] 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[16]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.

Type System

The type system of Pascal is extended by the provision of dynamic arrays and by the introduction of dimensioned types.

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 grammar is given by:

<dimensioned type>	<real type> <dimension >['*' <dimension>]*

<real type>	'real' 'double'

<dimension>	<identifier> ['POW' [<sign>] <unsigned integer>]

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 $\forall_{d \in B} \log_d t_1 = \log_d t_2 \times z$.

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.

Pascal 90[11] 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.

Expressions

The expression syntax of Vector Pascal incorportates extensions to refer to ranges of arrays and to operate on arrays as a whole.

Indexed Ranges

A range of components of an array variable are denoted by the variable followed by a range expression in brackets.

<indexed range>	<variable> '[' <range expression>[',' <range expression>]* ']'

<range expression>	<expression> '..' <expression>

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 byte;
procedure invert(var im:image);begin im:=255-im; end;
var screen:array[0..319,0..199] of byte;
then the following statement would be valid:
invert(screen[40..60,20..30]);

Unary expressions

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 implicit value under the operation. Thus just as **-a = 0-a** so too **/2 = 1/2**. For sets the notation **-s** means the complement of the set **s**. The implicit value inserted are given below.

type	operators	implicit value
number	+,-	0
set	+	empty set
set	-,*	fullset
number	*,/ ,div	1

A unary operator can be applied to an array argument and returns an array result. Similarly any user declared function over a scalar type can be applied to an array type and return an array. 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])**

Dyadic Operations

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

Addition operations

Operation	Left	Right	Result	Effect of *a* **op** *b*
+	integer	integer	integer	sum of *a* and *b*
	real	real	real	sum of *a* and *b*
-	integer	integer	integer	result of subtracting *b* from *a*
	real	real	real	result of subtracting *b* from *a*
+:	0..255	0..255	0..255	saturated addition cliped to 0..255
	-128..127	-128..127	-128..127	saturated addition clipped to -128..127
-:	0..255	0..255	0..255	saturated subtraction clipped to 0..255
	-128..127	-128..127	-128..127	saturated subtraction clipped to -128..127

Assignment

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

1. r**1:= r2[3]; { supported in standard Pascal }**
2. **r1:= /2; { assign 0.5 to each element of r1 }**
3. **r2:= r1*3; { assign 1.5 to every element of r2}**
4. **r1:= \+ r2; { r1gets the totals along the rows of r2}**

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.

Operator Reduction

Any dyadic operator can be converted to a monadic reduction operator by the functional \. Thus if a is an array, \+a denotes the sum over the array. More generally $\backslash\Phi x$ for some dyadic operator Φ means $x_0\Phi(x_1\Phi..(x_n\Phi\iota))$ where ι is the implicit value given 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.

Implementation

The Vector Pascal compiler is implemented in Java. It uses the ILCG[7] portable code generation 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.

The code generators follow the pattern matching approach described in[2][4]and [8], and are automatically generated from machine specifications written in ILCG (Intermediate Language for Code Generators). ILCG is a strongly typed language which supports vector data types and operators over vectors. It is well suited to describing MMX type 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.

Conclusions

Vector Pascal provides a new approach to providing a programming environment for multimedia instructionsets. It borrows notational conventions 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 compare the performance and elegance of implementations of a stereo-matcher algorithm implemented in Vector Pascal with the same algorithm implemented in C using the Intel image processing library.

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

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

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

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

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

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

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

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

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

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

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

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

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

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