Array Programming in Pascal

Paul Cockshott, Ciaran Mcreesh, Susanne Oehler
University of Glasgow Scotland, School of Computing Science
William.Cockshott@glasgow.ac.uk

Youssef Gdura
University of Tripoli Libya
y.gdura@ec.uot.edu.ly

Abstract

A review of previous array Pascals leads on to a description the Glasgow Pascal compiler. The compiler is an ISO-Pascal superset with semantic extensions to translate data parallel statements to run on multiple SIMD cores. An appendix is given which includes demonstrations of the tool.

Categories and Subject Descriptors D3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

Keywords Pascal, SIMD, Vector Processor, GPU

1. Previous Array Pascals

Pascal[17, 20] was one of the first imperative programming languages to be provided with array extensions. The first Array Pascal compiler Actus[24, 25] was roughly contemporary with the comparable Distributed Array Processor Fortran[15, 25].

Turner's Vector Pascal[28], another array extension of the language, was strongly influenced by APL[18]. It was similar in its array features to ZPL[2, 23, 27], Fortan90[9] or Single Assignment C[12, 26]. These all developed to address the challenge of the supercomputers that were coming into use at the time. Later Vector Pascal implementations were developed at Saarland University[10, 21] and the University of Glasgow[4, 5]. Pascal-XSC[13] an extension for scientific data processing provided extensions for vectors, matrices and interval arithmetic but was not a general array language.

In Actus was the syntax of array declarations indicated which dimensions of the array were to be evaluated in parallel.

```
var a:array[1:100,1..50} of integer;
```

Here the : rather than the .. is used to indicate that the dimension is to be evaluated in parallel. Actus provided both parallel assignments using index sets

```
a[range]:=40:56;
```

and parallel compound statements using the `within i:j do` construct.

The implicit assumption behind this design decision appears to have been that there would be distributed processors each with their own memory banks, so that the compiler would spread the array over the banks using the $i : j$ index form as a clue. This idea has not been used in subsequent Vector Pascal dialects which have been designed for machines with a unified memory.

2. Glasgow Vector Pascal

In what follows 'Vector Pascal' will refer to the Glasgow Vector Pascal compiler. The implementation initially targeted modern SIMD chips[7, 8, 11, 19] for which it used vectorisation techniques similar to those in the contemporary Intel C compiler[1]. With the advent of multi-core machines and GPU's subsequent Vector Pascal releases have supported automatic multi-core parallelism as well as SIMD parallelism.

2.1 Parallelism

Vector Pascal uses implicit parallelism obviating the need for a `within` statement. Conventional `for` loops will, in fact, be vectorised if there are no data dependencies, but the spirit of the language is to use APL style array expressions. Thus one can write:

```
type t=array[1..100,0..63] of real;
procedure foo(var a,b,c:t);
begin
a:=b*c;
end;
```

to operate on all corresponding elements of the three arrays. This is semantically equivalent to:

```
procedure foo(var a,b,c:t);
var iota:[0..1]~of~integer;
begin
  for iota[0]:= 1 to 100 do
    for iota[1]:= 0 to 63 do
      a[iota[0],iota[1]]:=b[iota[0],iota[1]]*
                          c[iota[0],iota[1]];
end;
```

The index vector `iota` is implicitly declared with sufficient elements to index the array on the left of the assignment scope covering the right of the assignment statement. Index vectors are usually elided provided that corresponding positions in arrays are intended. `Iota` can be explicitly used to perform things like circular shifts :

```
a:=b*c[iota[0],(iota[1]+1)mod 64];
```

Let us assume that we want to compile `foo` in program `bar.pas` to execute on a 6 core Xeon using the AVX instruction-set and 32 bit addressing, we use the command

```
vpc bar -cpuAVX32 -cores6
```


ARRAY'15, June 13, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3584-3/15/06...$15.00
http://dx.doi.org/10.1145/2774959.2774960

The compiler then transforms the code into:

```
procedure foo(var a,b,c:t);
 procedure stub(start:integer);
  var    iota:[0..1] of integer;
  begin
    for iota[0]:= start+1 step 6 to 100 do
     for iota[1]:= 0 step 8 to 63 do
       a[iota[0],iota[1]..iota[1]+7]:=
         b[iota[0],iota[1]..iota[1]+7]*
         c[iota[0],iota[1]..iota[1]+7];
  end;
var j:integer;
begin
  for j:=0 to 5 do  post_job(@stub,%ebp,j);
  for j:=0 to 5 do wait_on_done(j);
end;
```

The statement has been broken down into two forms of parallelism: an outer loop that runs on different cores doing every 6th row and an inner loop that operates 8 words at a time. The loops are then placed in a nested `stub` procedure. The threads on the different cores have access to the variables `a, b, c` by virtue of Pascal being a block structured language, but have local copies of `iota`. Access to the enclosing scope by the other cores is ensured by sending a static link from register `%ebp` when posting the job.

Vector Pascal does not support parallel `if` statements, but does allow parallel `if` expressions:

```
b:=if a < c then  a + 1
           else  a - 1
```

2.2 Map and Reduce

Any dyadic operator $\circ$ can be used as a reduction operator using the function form $\backslash\circ$, so * computes the product of a vector, \+ its sum etc.

Function applications map over arrays. The following example uses map and reduce.

```
type r=array[0..63] of real;
var a,b,c:array[1..100] of r;
function zot( p:real;q:r):real;
begin
    zot := p + \* q
end;
begin
  a:= zot(b,c)
end;
```

`zot` returns the scalar `p` added to the product of the elements of `q`. It is mapped over `b,c` as follows

```
for iota[0]:= 1 to 100 do
  for iota[1]:= 0 to 63 do
   a[iota[0],iota[1]]:=
    zot(b[iota[0],iota[1]],c[iota[0]]);
```

If you have a matrix, transposing the matrix amounts to swapping the order of the row and column indices. Thus for matrix `a`

```
a:= trans b;
```

will swap the indices of the right hand side. If `b` is a matrix this is equivalent to `a[i,j]:= b[j,i]` $:\forall i,j \in a$. But if `b` were a vector it is equivalent to `a[i,j]:= b[i]` $:\forall i,j \in a$.

Generalisation of transpose is provided by the `perm` operator which permutes the indices

```
p:=perm[2,0,1]q
```

Table 1. Compliance with ISO standard tests.

Compiler	Failed	% Success
Free Pascal 2.6.2	34	80
Turbo Pascal 7	26	87
Vector Pascal Pentium	0	100
Vector Pascal Xeon Phi	4	97.6

is equivalent to `p[i,j,k]:=q[k,i,j]` $:\forall i,j,k \in p$.

The . operator between arrays performs the scalar product thus: `u.v` is equivalent to the sum of products $\sum_\iota u_\iota \times v_\iota$. To the extent that + and * are overloaded so is scalar product. Thus when `u,v` are vectors of sets this evaluates as pairwise set intersection reduced by set union.

2.3 Types

The Pascal standard[17] supports sets over cardinal types. Vector Pascal extends this to any ordered type.

```
type colour=(red,yellow,green,blue,indigo,
            violet);
var  colours   :set of colour;
     candidates:set of 0..1000;
     names     :set of string[40];
```

For cardinal element types sets are implemented as bitmaps and set expressions vectorised using SIMD. Many Pascal implementations have a maximum set size of 2^8 elements. Vector Pascal supports sets of up to 2^{31} elements. For non-cardinal element types the sets are implemented as balanced trees and no vectorisation is used.

Dynamic sized arrays are supported as described by the Extended Standard[14].

```
type vec(len:integer)=array[1..len] of real;
pvec = ^vec;
var v:pvec;
begin
  new(v,10);
```

`v` is now a pointer to a vector of 10 reals.

Sub-array expressions of the form `a[i..j]` return a dynamic array with bounds `0..j-i`.

Literate programming[22] is supported by the compiler. The `-L` flag on the compiler command line causes a LaTeX documentation file to be created. Comments delimited thus `(*! *)` are treated as inline LaTeX, those delimited as `{ }` are rendered as marginal notes. Pascal code is reformatted to typographically distinguish reserved words and variable names. Formulae are rendered with appropriate maths notation.

Source code can be in UTF8 Unicode, and variable names can be in Roman, Greek, Cyrillic or CJK characters. Chinese equivalent reserved words are supported.

3. Implementation

The compiler is in Java and is released from SourceForge under GPL. It uses the `gcc` toolchain for linking and targets a range of contemporary and recent instruction-sets: Pentium, Opteron[19], SSE, SSE2, AVX, Playstation2(MIPS), Playstation3(Cell)[11], Nvida and the Intel Knights Ferry[6, 16]. The relevant assemblers must, of course, be installed. In addition non supported architectures can be targeted by the `-cpuC` option which translates Vector Pascal to C and uses `gcc` to generate binary. For the Cell and Nvidia implementations, the compiler generates code for an abstract SIMD machine that is implemented either in C on the vector processors or in CUDA on the GPU.

Performance achieved on Intel AVX and SSE architectures is comparable to the use of C with Vector Intrinsics and threaded building blocks[3]. However when compared to GPUs performance it is not as performant as Cuda. Though vector pascal source code tends to be more compact than C or Cuda for the same task.

Compliance with the ISO language standard is above that of some other leading Pascal compilers, see Table 1. The ISO-Pascal conformance test suite comprises 218 programmes designed to test each feature of the language standard. From the ISO test set a subset[1] was excluded that tests obsolete file i/o features as all three compilers follow the Turbo Pascal syntax for file operations. We ran the test suite using the host Vector Pascal compiler and in cross compiler mode for the XeonPhi. A programme was counted as a pass if it compiled and printed the correct result. A fail was recorded if compilation did not succeed or the programme, on execution, failed to deliver the correct result.

4. Future Work

We have a number ongoing student projects both to extend the Pascal system, and to add new front ends to it.

1. We are extending parallel reduction operations in Pascal to allow arbitrary dyadic functions, as opposed to operators to be used for reduction.
2. We are building a front end for the Haggis language used for teaching in Scottish schools, that uses the code generator subsystems used in the Pascal compiler.
3. We have a prototype Vector C front end for the compiler. This supports similar parallelisation mechanisms to Vector Pascal using a Matlab style array syntax. For example:

```
int main(int argc, char ** argv)
{
    int *p, yy[2][5]={{1,2,3,4,5},
                      {110,120,130,140,150}};
    p=malloc(20);
    p[0:4]=yy[1][1:4]+4;
    printf(" %d ",yy[0:2][0:4]);  printf("\n");
    printf(" %d ",p[0:4]);        printf("\n");
}
```

when compiled and executed produces as output:

```
$ Gcc test.c
$ a.out
 1  2  3  4  110  120  130  140
 124  134  144  154
```

`Gcc` here stands for Glasgow C Compiler. This prototype is not yet fully conformant with the C standard.

Acknowledgments

Thanks to the many Glasgow University students whose term projects contributed to the compiler and to CloPeMa, Collaborative project funded by the EU FP7-ICT , 288553

References

[1] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002.

[2] Bradford L Chamberlain, Sung-Eun Choi, C Lewis, Calvin Lin, Lawrence Snyder, and W Derrick Weathersby. Zpl: A machine independent programming language for parallel computers. *Software Engineering, IEEE Transactions on*, 26(3):197–211, 2000.

[3] P Cockshott, Y Gdura, and Paul Keir. Array languages and the n-body problem. *Concurrency and Computation: Practice and Experience*, 26(4):935–951, 2014.

[4] Paul Cockshott. Vector pascal reference manual. *SIGPLAN Not.*, 37(6):59–81, 2002.

[5] Paul Cockshott and Greg Michaelson. Orthogonal parallel processing in vector pascal. *Computer Languages, Systems & Structures*, 32(1):2–41, 2006.

[6] William Paul Cockshott, Susanne Oehler, and Tian Xu. *Developing a compiler for the XeonPhi (TR-2014-341)*. University of Glasgow, 2014.

[7] W.P. Cockshott and A. Koliousis. The SCC and the SICSA multi-core challenge. In *4th MARC Symposium*, December 2011.

[8] Peter Cooper. Porting the Vector Pascal Compiler to the Playstation 2. Master's thesis, University of Glasgow Dept of Computing Science, http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/PS2.pdf, 2005.

[9] AK Ewing, H Richardson, AD Simpson, and R Kulkarni. *Writing Data Parallel Programs with High Performance Fortran*. Edinburgh ParallelComputing Centre, 1998.

[10] A. Formella, A. Obe, WJ Paul, T. Rauber, and D. Schmidt. The SPARK 2.0 system-a special purpose vector processor with a VectorPASCAL compiler. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume 1, pages 547–558. IEEE, 1992.

[11] Youssef Omran Gdura. *A new parallelisation technique for heterogeneous CPUs*. PhD thesis, University of Glasgow, 2012.

[12] C. Grelck and S.-B. Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[13] R Hammer, M Neaga, and D Ratz. Pascal xsc. *New Concepts for Scientific Computation and Numerical Data Processing*, pages 15–44, 1992.

[14] Tony Hetherington. An introduction to the extended pascal language. *ACM SIGPLAN Notices*, 28(11):42–51, 1993.

[15] DAP ICL. Fortran language reference manual. *ICL Technical Publication TP6918*, 1979.

[16] Intel Corporation. *Intel Xeon Phi Product Family: Product Brief*, April 2014.

[17] ISO. *Pascal ISO 7185*, 1990.

[18] K. Iverson. *A programming language*. Wiley, New York, 1966.

[19] Iain Jackson. Opteron Support for Vector Pascal. Final year thesis, Dept Computing Science, University of Glasgow, 2004.

[20] Kathleen Jensen, Niklaus Wirth, Andrew B Mickel, and James F Miner. *Pascal: user manual and report*, volume 3. springer-Verlag New York, 1975.

[21] Christoph W Kessler, Wolfgang J Paul, and Thomas Rauber. Scheduling vector straight line code on vector processors. In *Code Generation Concepts, Tools, Techniques*, page 73..91. Springer, 1992.

[22] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[23] Calvin Lin and Lawrence Snyder. Zpl: An array sublanguage. In *Languages and Compilers for Parallel Computing*, pages 96–114. Springer, 1994.

[24] R. H. Perrott. A Language for Array and Vector Processors. *ACM Trans. Program. Lang. Syst.*, 1(2):177–195, October 1979.

[25] R. H. Perrott and A. Zarea-Aliabadi. Supercomputer languages. *ACM Comput. Surv.*, 18(1):5–22, 1986.

[26] S.-B. Scholz. —Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[1] Tests 1,3,5, 19, 54, 67..76,78,90..92, 111..115, 118, 121, 131, 141, 160, 197, 198, 202, 203, 212, 213.

[27] L Snyder. *A Programmer's Guide to ZPL.* MIT Press, Cambridge, 1999.

[28] T Turner. *Vector Pascal a Computer Programming Language for the Array Processor.* PhD thesis, PhD thesis, Iowa State University, USA, 1987.

Appendix Demo Description

Here is a scaled up version of the programme described earlier

```
$ cat bar.pas
program bar;
type t=array[1..800,1..1024] of real;
procedure foo(var a,b,c:t);
begin
   a:=b*c+c;
end;
var p,q,r:t; i:integer;
begin
   for i:= 1 to 100 do foo(p,q,r)
end.
```

It performs 2*800*1024*100= 163 million arithmetic operations, we can compile it for the default Pentium code model and produce a LaTeX listing file thus:

```
$ vpc bar -L
Glasgow Pascal Compiler (with vector exensions)
 11                              bar.pas->TeX
compiled
as --32  --no-warn -g -o p.o p.asm
gcc -g -m32 -o bar p.o   /home/wpc/bin/rtl.c
```

Running it on an AMD A6 we get

```
$ time bar
real 0m1.888s
user 0m1.870s
sys 0m0.008s
```

We can now compile it for the AVX instruction-set

```
$ vpc bar -cpuAVX32
```

This vectorises the code so it runs much faster

```
$time bar
real 0m0.356s
....
```

It can be further accelerated by multicore compilation. Note it is not worth using more than 2 cores on this model of CPU as there are only 2 vector floating point units shared between the 4 cores.

```
$ vpc bar -cpuAVX32 -cores2
$ time bar
real 0m0.300s
```

Although on programmes as small as this gains from parallelism are not guaranteed. We get the following code for the inner loop:

```
mov       eax,DWORD[ebp+16]; pntr to C
lea       eax,[eax+ebx -4096]
mov       edi,DWORD[ebp-12]
vmovdqu   YMM0,[eax+edi* 4]; 8 words C
imul      ecx,DWORD[ebp-12],4
mov       eax,DWORD[ebp+8];  pntr to A
lea       edx,[ebx+ecx -4096]
mov       edi,DWORD[ebp+12]; pntr to B
lea       esi,[ecx+ebx -4096]
vmovdqu   YMM1,[edi+esi]  ;  8 words B
vmulps    YMM1, YMM1, YMM0;  C*B
vaddps    YMM1, YMM1, YMM0;  C*B+C
vmovdqu   [eax+edx],  YMM1;  8 words to A
```

Now let us look at the listings,

```
$ cat bar.lis
listing of file bar.pas
       +---A 'P' at the start of a line
          indicates the line has been SIMD
          parallelised
       |+--An 'M' at the start of a line
           indicates the line has been
           multi-core parallelised
       ||
       vv
    1  program bar;
    2  type t=array[1..800,0..1024] of real;
    3
    4  procedure foo(var a,b,c:t);
    5  begin
    6 PM a:=b*c+c;
    7  end;
    8  var p,q,r:t; i:integer;
    9  begin
   10     for i:= 1 to 100 do foo(p,q,r)
   11  end.
```

Or we can run `latex` on the `bar.tex` file and get a pretty print version looking like this

4.1 bar

program *bar* ;
type
  t =**array** [1..800,0..1024] **of** *real* ;

procedure *foo* (**var** *a* ,*b* ,*c* :*t*); (see Section 4.2)
var
  Let p, q, $r \in$ t;
  Let $i \in$ integer;
begin

  for $i \leftarrow 1$ **to** 100 **do**
    foo (p, q, r);
end .

4.2 foo

procedure *foo* (**var** *a* ,*b* ,*c* :*t*);
begin
  $a \leftarrow b \times c + c$;
end ;

Next let us compare the performance of Vector Pascal with C when blurring a 1024x1024 pixel colour image. The same separable convolution algorithm is used in both cases:. The addition of the C file on the compiler command line instructs it to link the Pascal and C in a single binary.

```
$ vpc blurtime cconv.c
$ blurtime
PASCAL         0.03 per run
C             0.442 per run
```

Pascal outperforms C in this example because it uses saturated SIMD arithmetic on pixels.

Finally as a bit of fun, matrix product of numbers and strings to print a Roman number:

```
$ cat roman.pas
program roman;
const rom:array[0..4] of string[1]=
                  ('C', 'L', 'X', 'V', 'I');
     numb:array[0..4] of integer=(2,1,1,0,3);
var s:string;
begin
    s:= numb . rom;
    writeln(s);
end.
$ roman
CCLXIII
```