CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Compiling Vector Pascal to the Xeon Phi

Mozhgan Chimeh, Paul Cockshott*, Susanne B. Oehler, Ashkan Tousimojarad, Tian Xu

[1] *University of Glasgow, School of Computing Science*

SUMMARY

Intel's Xeon Phi is a highly parallel x86 architecture chip made by Intel. It has a number of novel features which make it a particularly challenging target for the compiler writer. This paper describes the techniques used to port the Glasgow Vector Pascal Compiler (VPC) to this architecture and assess its performance by comparisons of the Xeon Phi with 3 other machines running the same algorithms. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Xeon Phi; Many Integrated Core (MIC); Glasgow Vector Pascal; OpenMP; Pre-fetching; Nvidia GPU

1. CONTEXT

This work was done as part of the EU funded CLOPEMA project whose aim is to develop a cloth folding robot using real time stereo vision. At the start of the project we used a Java legacy software package, C3D[1] that is capable of performing the necessary ranging calculations. When processing the robot's modern high resolution images it was prohibitively slow for real time applications, taking about 20 minutes to process a single pair of images.

To improve performance, a new Parallel Pyramid Matcher (PPM) was written in Vector Pascal [2]†, using the legacy software as design basis. The new PPM allowed the use of both SIMD and multi-core parallelism[3]. It performs about 20 times faster on commodity PC chips such as the Intel Sandybridge, than the legacy software. With the forthcoming release of the Xeon Phi it was anticipated to be able to obtain further acceleration running the same PPM code on the Xeon Phi. Hence, taking advantage of more cores and wider SIMD registers, whilst relying on the automatic parallelisation feature of the language. The key step in this would be to modify the compiler to produce Xeon Phi code. However, the Xeon Phi turned out to be considerably more complex compared to previous Intel platforms. Porting of the Glasgow Vector Pascal compiler became an entirely new challenge, and required a different porting approach than previous architectures.

2. PREVIOUS RELATED WORK

Vector Pascal [4, 2] is an array language and as such shares features from other array languages such as APL [5], ZPL [6, 7, 8], Distributed Array Processor Fortran [9], Fortan90 [10] or Single

*Correspondence to: Dr. William Cockshott, School of Computing Science, University of Glasgow, 18 Lilybank Gardens, Glasgow G12 8RZ. E-mail: William.Cockshott@glasgow.ac.uk

Contract/grant sponsor: ¡Contract/grant sponsor name (no number)¿

Contract/grant sponsor: CloPeMa, Collaborative project funded by the EU FP7-ICT; contract/grant number: 288553

†Glasgow Vector Pascal specialises in scientific and image processing computation.

Assignment C [11, 12]. The original APL and its descendent J were interpretive languages in which each application of a function to array arguments produced an array result. Whilst it is possible to naively generate a compiler that uses the same approach it is considered inefficient as it leads to the formation of an unnecessary number of array temporaries. This reduces locality of reference and thus cache performance. The key innovation in efficient array language compiler development was Budd's [13] principle to create a single loop nest for each array assignment and to create temporaries as scalar results. This principle was subsequently rediscovered by other implementers of data parallel languages or sub-languages [14]. It has been used in the Saarbrucken[15] and the Glasgow Vector Pascal compilers.

The first Vector Pascal compiler was reported by Perrot[16] who called his Pascal variant Actus. This was an explicitly data parallel version of Pascal where the syntax of array declarations indicated which dimensions of the array were to be evaluated in parallel.

It was also possible to use explicit ranges of indices within an array access thus `v[a:b]`. Two subsequent Vector Pascal compilers[4, 2] generalised the ISO Pascal .. range notation to denote a range on which vector operations were to occur `v[a..b]`, but the Saarbrucken one[17] retained the `a:b` notation. Parallel execution of multiple statements could be induced by the `within` construct:

```
within i:j do
begin
a[#] := b[#] + incr;
b[#] := b[# shift 1]
end;
```

The # symbol was substituted with the appropriate element of the index set. The current compiler does not have a `within` statement but simple for-loops without sequence dependencies are vectorised.

A similar approach was taken in parallel `if` statements:

```
if a[10:90] < a[10:90 shift - 1] then b[#] := a[#] + 1
else b[#] := a[#] - 1
```

The current Glasgow Vector Pascal compiler does not support parallel `if` statements but does allow parallel `if` expressions to achieve the same result:

```
b[10..90]:=if a < a[iota[0] - 1] then  a + 1
           else a := a - 1
```

Note that the # notation is not supported. Instead index sets are usually elided, provided that the corresponding positions in the arrays are intended. If offsets are intended the index sets can now be explicitly referred to using the predeclared array of index sets `iota`. `iota[0]` indicates the index set for the leading dimension, `iota[1]` the next dimension etc.

Perrott's compiler was targeted at distributed memory SIMD machines. The Turner and Saarbrucken ones aimed at attached vector accelerators, the Glasgow implementation has targeted modern SIMD chips[18, 19, 20, 21], with the first release targeted at single cores like the Intel P3 or the AMD K6. In performing vectorisation the compiler used similar techniques to those reported for the contemporary Intel C [22]. Vectorisation was done by the front end which passed a vectorised semantic tree to machine specific code generators. This paper reports on a new vectorisation strategy in the Glasgow implementation, which uses the compiler back-end.

After multi-core chips became available and after collaboration with the Single Assignment C(SAC)[11] team, SAC-style support for multi-core parallelisation was added. The current Vector Pascal compiler distributes the rows of a two dimensional array computation across cores, rather than offering the Arctus option of selecting along which dimension multi-processor parallelism will be invoked. It inherits from SAC the policy of starting worker threads for all cores on the first invocation of an array-parallel computation. These worker threads then block, but are available for subsequent array computations. As in SAC, multi-core parallelism is invoked automatically, without

DOI: 10.1002/cpe

the need for any pragmas or other programme notation, whenever a map operation is performed over two dimensional arrays.

3. FEATURES OF THE XEON PHI

The Xeon Phi is a co-processor card, with one Many-Integrated-Core (MIC) chip on a PCI board, that plugs into a host Intel Xeon system. The board itself has its own GDDR5 memory. The MIC chip, on the Xeon Phi 5110P [23], contains 60 cores, each with its own cache. Each core is in addition 4-way hyperthreaded. Shared coherent access to the caches allows the chip to run a single Linux image. Using the `p-threads` library a single Linux process may fork threads across different cores sharing the same virtual memory.

The individual cores are a hybrid supporting legacy instructions equivalent to those on the original Pentium, but with an AMD64 register architecture. They are described as in-order cores[24] as opposed to the dynamically scheduled cores of other Xeon models. The Linux runs 64bit with a modified ABI. There are no MMX, SSE or AVX or conditional move instructions.

A whole set of new instructions have been added[25]. These resemble AVX : SIMD registers and a 3 operand format, but the binary and the assembly language are different. SIMD registers are 512 bits versus the AVX 256 bits. Key features are:

- The ability to perform arithmetic on 16 pairs single precision floating point numbers using a single instruction - this is a simple extension of what AVX can do.
- Individual bits of 16 bit mask registers can be set based on comparisons between 16 pairs of floating point numbers.
- The usage of mask registers to perform conditional writes into memory or into vector registers.
- On the fly conversions between different integer and floating point formats, when loading vector registers from memory.
- Performing scattered loads and stores using a combination of a vector register and a general purpose register.

The last point is the most novel feature of the architecture. Let us consider the following Pascal source code samples:

```
for i:=0 to 15 do x[i]:= b[a[i]];
```

I.e. `x` is to be loaded with a set of elements of `b` selected by `a`, where `x` and `b` are arrays of reals and `a` is an array of integers. Vector Pascal allows us to abbreviate this to:

```
x:=b[a]
```

The key step, the expression `b[a]` in this can in principle be performed on the MIC as

```
knot k1,k0
1:vgatherdps ZMM0{k1},[r15+ ZMM8*4 ]
jknzd k1,1b
```

assuming that `r15` points at the base address of array b, `ZMM8` is a vector register that has been preloaded with `a`, and `ZMM0` is the register that will subsequently be stored in `x`. What the assembler says is

1. Load mask register `k1` with `0FFFFH`.
2. Load multiple words into `ZMM0` from the addresses formed by the sum of `r15` and 4 times the elements of `ZMM8`, clearing the bits in `k1` corresponding to the words loaded.
3. Repeat so long as there are non zero bits in `k1`.

4. PASCAL PARALLELISM MECHANISM

Vector Pascal[4, 2], supports parallelism implicitly by allowing array variables to occur on the left hand side of an assignment expression. The expression on the right hand side may use array variables wherever standard Pascal allows scalar variables.

DOI: 10.1002/cpe

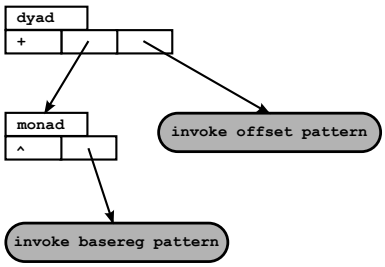


Figure 1

The order in which assignments to the individual array elements are assigned is undefined, the compiler can perform the calculations in a data parallel fashion. In fact, for two dimensional arrays, it parallelises the first dimension across multiple cores, the second dimension using SIMD instructions.

The compiler is re-targeted between different machine architectures using the Intermediate Language for Code Generation (ILCG) [26]. The machine description consists of declarations for registers and stacks, followed by semantic patterns. Patterns specify data types, addressing modes, operators or classes of instructions. The type system supports vector types and its expressions, parallel operations over vectors.

The ILCG compiler translates machine specifications into Java classes for code generators. Let file `AVX32.ilc` define an AVX cpu, the ILCG compiler will produce file `AVX32.java` which translates an abstract semantic tree into semantically equivalent AVX assembler. A compiler flag in Vector Pascal specifies which code-generator class to use [18, 19, 20].

An example patterns used is the specification of base plus offset addressing on an Intel processor:

```
pattern baseplusoffsetf(basereg r, offset s )
means [ +(^(r),s) ] assembles[  r  '+'  s ];
```

The portion `+(^(r),s)` means match the tree in Figure 1. Note that `^` is the dereference operator.

5. VECTORISATION CHALLENGES

The front end of the Pascal compiler is machine independent; the front end queries the code generator to discover types are supported for SIMD and the length of the SIMD registers. Given an array assignment, the front end first generates the tree for a scalar for-loop to perform the requisite semantics. It then checks whether this is potentially vectorisable, and if it is, it checks if the code generator supports SIMD operations on the specified type. If yes then it converts the scalar for-loop into a vectorised for-loop.

The checks for vectorisability involve excluding statements :

1. with function calls, e.g., `a[i]:=ln(b[i])`,
2. or in which the loop elements are not adjacent in memory :
   (a) ones where the iterator `i` is not used as the rightmost index of a multi-dimensional array, e.g., `a[i]:=b[i,j]`,
   (b) statements in which the loop iterator `i` is multiplied by some factor, e.g.,
   ```
   a[i] := b[i * k]
   ```
   (c) statements in which the index of an array is another array, e.g.,
   ```
   a[i] := b[ c[i] ].
   ```

These were reasonable restrictions for the first and second generation of SIMD machines, but some of them are unnecessary for more modern machines such as the Xeon Phi. In particular the availability of gather instructions means that we do not need to be so restrictive in the forms of

DOI: 10.1002/cpe

access that can be vectorised. One can envisage other machines being built that will allow at least certain functions to be performed in SIMD form.

Rather than use complex machine specific tests and tree transformations in the front end, it was decided to keep a common front end and extend the machine specification language. ILCG has been upgraded to allow tree → tree transformations before these are mapped into assembler strings.

The extensions made to ILCG are:

- Patterns can now be recursive.
- Curried patterns using a parameter passing mechanism.
- tree → tree transformer patterns can be defined.
- Pattern matching can be guarded by Boolean predicates.

As a first illustration let us look at a pattern that will vectorise for-loops on the Xeon Phi:

```
transformer pattern vectorisablefor(any i,any start,
                                    any finish, vecdest lhs,
                                    vectorisablalternatives rhs)
 means[for (ref int32)i:=start to finish
             step 1 do lhs[i] := rhs[i]]
 returns[  statement(/* scalar residual loop */
 for i.in:=+( +( -( *( div( +(1, -(finish.in,start.in)),16),16),1),
        start.in),1)
  to finish.in step 1 do lhs.in := rhs.in,
         /* vector loop */
  for i.in:= start.in to +( -( *( div( +(1, -(finish.in,start.in)),
        16),16),1),start.in)
         step 16 do lhs.out:=rhs.out
 )];
```

The pattern recognises a for-loop with steps of 1 and integer iterator `i`. The body must be an assignment, whose left hand side is matched by `vecdest` with `i` as a parameter, and the right hand side matches `vectorisablealternatives` curried with `i`. It returns two loops, the first a scalar one for the residual data that may not fit in vector registers. The second is vectorised in steps of 16 and uses the transformed outputs of the two patterns invoked earlier. The transformer pattern `vectdest` illustrates predicates.

```
transformer pattern vecdest(any i,any r, vscaledindex j)
/* this rule recognises scalar addresses that can
   be converted to vector addresses */
 means[ (ref ieee32) mem((int64) + (r, j[i])) ]
 returns[ mem(+(j.in,r.in), ieee32 vector(16)) ]
 precondition[ NOT( CONTAINS(r.in, i.in))];
```

Given a memory reference to an `ieee32` scalar it generates a reference to a 16 element vector of `ieee32` at the same address. A precondition is that the base address `r` does not contain `i`, the loop iterator, which must only occur in the scaled index address part recognised by `j[i]`. The suffixes `.in` and `.out` in patterns refer to the tree the identifier matched pre and post transformation. The rules for the vectorisation of 32 bit floating point loops for the Xeon Phi take up 20 transformer patterns. The two main patterns recognise map and reduce loops. Subsidiary patterns include ones to allow non adjacent vector locations for gather instructions and to check that no dependencies or function calls exist. In addition the machine description includes 320 instruction patterns, which generate assembler code.

The extensions allowed utilisation of gather instructions in bulk parallel array statements without modifying the front end and cause automatic vectorisation of many ordinary Pascal for-loops.

In principle vectorisation could now be removed from the front end, but doing this would involve modifying many legacy ILCG machine descriptions. For the Xeon Phi, however, we disabled vectorisation in the compiler front end. Further additions to ILCG such as polymorphic transformation functions, class inheritance for machine architectures, would allow more concise machine descriptions, but ILCG is a Domain Specific Language(DSL). As art constantly aspires to

Prepared using cpeauth.cls
DOI: 10.1002/cpe

the condition of music[‡], DSLs aspire to the condition of general programming languages. But if that occurs, they cease to be DSLs. One has to know when to stop extending.

5.1. Pre-fetching

The Xeon Phi reportedly uses in-order cores[24], a memory access will hold up the pipeline until the data becomes available. Since there is a significant overhead to accessing the external memory it is desirable that as many accesses as possible are satisfied from the cache. Intel provides special instructions `vprefetch0, vprefetch1` that will speculatively load data into the level 1 and level 2 cache respectively. These instructions are hints to the cache hardware, and will not trigger page faults or exceptions if the address supplied is non resident or protected. In [28] a sophisticated strategy is given for making use of these.

With a compiler primarily for image processing applications a simpler pre-fetching strategy can and has been implemented. In image processing it cannot be assumed that memory fetches for vector operations will be aligned on 64 byte boundaries as expected by the Intel compilers described in [28]. Image processing routinely involves operations being performed between sub-windows, placed at arbitrary origins within an image. In order to support this Vector Pascal allows the applications programmer to define array slices, which in type terms are indistinguishable from whole arrays. At run time a 2D array slice will specify a start address and an inter-line spacing. 2D array slices are particularly useful for defining sub-windows for image processing code. Such code typically takes two or more windows and performs some mathematical operation between corresponding pixels in the two or more windows. Aligned arrays will be the exception rather than the rule which means that vector loads have to be unaligned loads. The dynamic loop peeling technique reported by Bik[22] for dealing with unaligned arrays does not work when you have two or more arrays in an operation that are mutually misaligned. It can at best bring one of the arrays into alignment [29]. Wu et al[30] give a more complex alignment technique, which does handle multiple mutually unaligned arrays. Whilst this was appropriate for some instruction-sets, the Xeon Phi `vloadunpacklps` and `vloadunpackhps` perform essentially the same operations as the longer sequences of instructions proposed in [30].

Even if arrays are mutually misaligned, it is still possible to fetch ahead each time, one performs a vector load. The assumption is that these will mainly occur within loops, so that on subsequent loop iterations, data will already have been loaded into the caches. The following sequence illustrates the use of pre-fetching in assembler output of the compiler.

```
vloadunpacklps  ZMM1,[  rsi+rdi]
vloadunpackhps  ZMM1,[  rsi+rdi+64]
vprefetch0 [  rsi+rdi+256]
vprefetch1 [  rsi+rdi+512]
```

Note that since the alignment is unknown the compiler has to issue two load instructions : `vloadunpacklps` followed by `vloadunpackhps` to load into vector register ZMM1 the low and high portions of a 16 float vector (64 bytes) from two successive 64 byte cache lines. This explicit exposure of the cache lines in the instruction set architecture is a new feature for Intel. In previous Intel instruction sets, a data transfer that potentially spanned two cache lines could be performed in a single instruction. Let us call the lines accessed by the vloadunpack instructions: l_0 and l_1.

It then issues a prefetch for line l_4 into the level 1 cache, and a prefetch for l_8 into the level 2 cache. The fetch for line l_4 is $4 \times 64 = 256$ bytes on from the original effective address specified by the sum of `rsi` and `rdi`. The intention is that by the time 4 iterations of the loop have been performed the data for the next iteration will have been preloaded into the level 1 cache.

[‡]"All art constantly aspires towards the condition of music. For while in all other kinds of art it is possible to distinguish the matter from the form, and the understanding can always make this distinction, yet it is the constant effort of art to obliterate it."[27]

DOI: 10.1002/cpe

Table I. Compliance with ISO standard tests.

Compiler	Free Pascal ver 2.6.2	Turbo Pascal Ver 7	Vector Pascal Xeon Phi	Vector Pascal Pentium
Number of fails	34	26	4	0
Success rate	80%	84.7%	97.6%	100%

5.2. Multi-core issues

Vector Pascal uses p-threads to support multi-core parallelism over matrices. Our experiments showed that the thread dispatch mechanism used in the standard multi-core implementations needed a couple of changes to get the best out of the Xeon Phi.

- The default is to use thread affinity to tie threads to cores. On the Xeon Phi it gives a better performance if thread affinity is disabled.
- The default task scheduling uses p-thread semaphores. On the Xeon Phi it was more efficient to use spin-locks(busy waiting).

6. COMPILER EVALUATION

The porting success of the Glasgow Vector Pascal compiler for the Xeon Phi has been evaluated in semantic terms, efficiency terms, and ease of use. Note: a compiler was considered semantically successful if it correctly compiles and evaluates a language test suite. Hence, in section 6.1 the compiler got evaluated against the ISO Pascal programme test suite. Section 6.2 discusses the ease of use evaluation for which complexity metrics were applied to the micro-benchmark sources in different languages. Where as the efficiency term was evaluated against other language/machine combinations in section 6.3 .

6.1. Standards conformance

The ISO-Pascal conformance test suite comprises 218 programmes designed to test each feature of the language standard. Since Vector Pascal is an extension of standard Pascal we prepared a further 79 test programmes to validate language extensions. From the ISO test set a subset[§] was excluded that tests obsolete file i/o features as Vector Pascal follows 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 Xeon Phi. 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. Comparison with the host Vector Pascal compiler gives and indication of residual errors in the code generator. As Table I shows the Xeon Phi system passes 97.6% of the tests. This is still some way short of the 100% compliance achieved on the Xeon host targeting Pentium code.

In order to judge of whether the observed failure rates were good or poor by industry standards, we also ran the test suite through the final version of the highly successful Turbo Pascal compiler for DOS and the Free Pascal compiler for Linux. Both of these use the same Turbo Pascal syntax for file i/o and can validly be tested on the same subset of conformance tests. Table I shows that semantic compliance by these widely used Pascal compilers is significantly below that achieved by the new Xeon Phi compiler. Our aim, in future work will be to achieve 100% compliance.

6.2. Ease of use

The test programs were a matrix multiply micro benchmark and two algorithms taken from real code used in the parallel pyramid matcher : scaling an image using linear interpolation and image

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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Listing 1 The Pascal matrix multiply routine. Note that . is a predefined inner product operator working over arrays.

```
1  type m = array[1..n,1..n] of real;
2  procedure f( var ma,mb,mc:m);
3  begin
4          ma:= mb.mc;
5  end;
```

Listing 2 The C sequential and parallel matrix multiply routines. Note that the parallel loop nesting has been altered to obtain better fetch performance of adjacent data in the inner loop.

```
1  void MatMul_Seq(double* D, double* E, double* F) {
2   for (int i=0;i<N;i++)
3    for(int j=0;j<N;j++)
4     for(int k=0;k<N;k++)
5      F[i*N+j] += D[i*N+k] * E[k*N+j];}
6
7  void MatMul_Par(double* A, double* B, double* C) {
8  #pragma omp parallel shared(A,B,C)
9  {
10 #pragma omp for
11  for (int i=0;i<N;i++)
12   for(int j=0;j<N;j++) {
13    double temp = A[i*N+j];
14    #pragma ivdep
15    for(int k=0;k<N;k++){
16     C[i*N+k] += temp * B[j*N+k]; }}}}
17 return; }
```

convolution. These operations were selected because they take up most of the cycles in our stereo matching algorithm. The images were stored as red, green and blue planes of 32 bit floating point numbers, image sizes ranged from 512 by 512 to 5832 by 5832 pixels rising in steps of $\sqrt{2}$.

As a means of rating the efficiency of the Vector Pascal all 3 algorithms were ran in C as well as Vector Pascal. The C version was compiled with Intel compiler `icc`. Using OpenMP that includes pragma directives, allowed us to create both vectorised and multicore loops. For scaling and convolution CUDA versions were also developed, run and compared against the others.

6.2.1. Matrix Multiply We compare C, Vector Pascal and C with OpenMP matrix multiply in Algorithms 1 and 2. The C code uses the most obvious sequential algorithm. The OpenMP derivation is 60% longer and re-organises the loop nest structure in order to make vectorised fetching of data easier. The Pascal version is the shortest and could normally be shortened to just one line, or even less since inner product is an operator in Vector Pascal allowing one to write matrix expressions like $(A.B) + v$ where A, B matrices and v a vector. The Pascal operation expands into a loop structure analogous to that for the sequential C++ which is then automatically parallelised and vectorised.

In terms of complexity, the ranking for this task is Pascal, C++, OpenMP as shown in Table II.

6.2.2. Image Rescale For image scaling the input image was expanded by $\sqrt{2}$ in its linear dimensions. The inner function used for the Pascal image scaling test is shown in Listing 4. This uses temporary arrays `vert` and `horiz` to index the source image array. The C++ version is in Listing 3. Lines 4-12, is the main computation of the scaling routine. It is within the two for-loops over the number of rows and columns. For the multi-core parallelisation, we use different pragmas to split up the outer loop to run across cores. However, to create a vectorised loop, SIMD construct can be used instead.

DOI: 10.1002/cpe

Table II. Code complexity measures

	Lines	Nonspace chars
Seq C++ Matmul	10	190
Par C++ Matmul	16	327
Pascal Matmul	**4**	**89**
Seq C++ Conv	33	1470
Par C++ Conv	39	1585
Pascal Conv	53	1435
CUDA Conv	227	6181
Seq C++ Scale	37	1072
Par C++ Scale	39	1108
Pascal Scale	**17**	**837**
CUDA Scale	98	2189

Listing 3 : C++ main loop nest for scaling routine. Note that `pimage` is a predefined struct type that has the number of maxcol, maxrow and maxplane of an image. Some lines are elided in the listing.

```
1  void planeinterpolate ( float ***A, float ***B,
2                          int maxplane_index, float dx,
3                          float dy,pimage a,pimage b){
4  ...
5   for (int i=0 ; i< b.maxrow ; i++)
6    for (int j=0 ; j< b.maxcol ; j++)
7     B[tmp][i][j] = (A[tmp][vert[i]][horiz [j]] *
8     (1 - hr [j]) +  A[tmp][vert[i]][horiz [j] + 1] *
9      hr [j] ) * (1 - vr [i]) +
10    (A[tmp][vert[i] + 1][ horiz [j]] * (1 - hr [j]) +
11     A[tmp][vert[i] + 1][ horiz [j] +1 ] * hr [j]) * vr [i];
12 ...}
```

Listing 4 : The Pascal scaling routine.

```
1  PROCEDURE planeinterpolate ( var a,b:Plane);
2    var vert,horiz:pivec;   vr,hr:prvec;
3    begin
4     new (vert, b.maxrow); new(vr,b.maxrow);
5     vert^:=trunc( iota [0] *dy); vr^:=iota [0]*dy -vert^;
6     new( horiz,b.maxcol); new(hr,b.maxcol);
7     horiz^:=trunc( iota [0] *dx); hr^:=iota [0] *dx-horiz^;
8     (* we have computed vectors of horizontal and vertical
9     positions in vert and horiz, the horizontal and vertical
10    residuals in hr and vr *)
11    B [ 0..B.maxrow-1,0..B.maxcol-1]:=
12         (A[vert^[iota[0]]   ,horiz^[iota[1]]]*
13           (1-hr^[iota [1]])+  A[vert^[iota[0]]   ,
14          horiz^[iota[1]]+1]*hr^[iota [1]] )*  (1-vr^[iota[0]])+
15           (A[vert^[iota[0]]+1,horiz^[iota[1]]]*  (1-hr^[iota [1]])+
16            A[vert^[iota[0]]+1,horiz^[iota[1]]+1]*hr^[iota [1]]
17         )*  vr^[iota[0]];
18    dispose(vert);dispose(horiz);dispose(vr);dispose(hr);
19   end;
```

Note that `iota` is the implicit index vector for the expression, so iota[0], iota[1] give the x,y position in the image array.

The parallel loop is distributed across cores and then within each, combines iterations of each loop in to vectorisation. The C++ version closely follows the Vector Pascal one, but the CUDA

DOI: 10.1002/cpe

Listing 5 CUDA code for the scaling kernel. Note the use of 'texture memory' which can be indexed by real valued coordinates.

```
__global__ void subsampleKernel(
    float *d_Dst, int imageW,]{
    int imageH, float scalefactorW,  float scalefactorH,
    int imageW2,  int imageH2
)
{
    const int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
    const int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);
    const float  x = (float)ix + 0.5f;
    const float  y = (float)iy + 0.5f;
    if (ix >= imageW2 || iy >= imageH2) return;
    d_Dst[IMAD(iy, imageW2, ix)] =
      tex2D(texSrc, x / scalefactorW,     y / scalefactorH);
}
```

Listing 6: Pseudo code for the row filter kernel of convolution on GPU.

```
For each multiprocessor
  Repeat
    Select an unprocessed block with x y identity
      Copy the block from global to local memory
      Perform horizontal convolution on local memory
      Write back to global memory
  Until all blocks done
```

program consists of three steps: (1). data is first transferred from host to GPU device memory; (2). then the kernel code is executed on the GPU; and (3). finally the results are transferred back to the host from the GPU device. Listing 5 shows the scaling kernel in CUDA to run on an Nvidia GPU. The kernel function is in principal invoked in parallel on all pixel positions. On the GPU it is not necessary to explicitly code for the interpolation. Instead CUDA offers a feature called texture memory that allows implicit linear interpolation of a two dimensional array of data. The interpolation is performed when the array is indexed using the `tex2d` function.

6.2.3. Image Convolution A separable convolution kernel is a vector of real numbers that can be applied independently to the rows and columns of an image to provide filtering. It is a specialisation of the more general convolution matrix, but is algorithmically more efficient to implement ¶. For all tests we used separable kernels of width 5 as required for our stereo vision system.

The Pascal implementation attempts to minimise the cache usage, convolving the image in slices that will fit into the cache. This optimisation, which is not parallelism specific, adds some complexity to the algorithm.

The GPU convolution algorithm is more complex than the corresponding scaling algorithm. In order to reduce the number of idle threads tiling is used. The image is divided into a set of rectangular blocks. From the standpoint of the CUDA programming language, these blocks in the image are defined as 'thread blocks', i.e. rectangular blocks of programme threads that run in SIMD fashion. At run-time each block is automatically allocated to a multiprocessor. All threads within one block execute on the allocated multiprocessor. Listing 6 shows pseudo code for the row filter part of the convolution algorithm that runs on an Nvidia GPU. The column filter pass operates much like the row filter pass.

¶The complete algorithm can be found in Vector Pascal as Listing **??** of the additional materials file

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Listing 7 The Pascal routine that occupies most of the cycles for convolution.

```
PROCEDURE MA5(VAR acc,  p1,p2,p3,p4,p5:plane;k:kernel);
BEGIN
  acc:= p1* k[1] +p2*k[2] +p3*k[3] +p4*k[4]+p5*k[5];
END;
```

Table III. Ratio of Xeon Phi to Ivy Bridge Xeon speeds for different image sizes. 12 threads on Xeon compared to 100 threads on Xeon Phi. Numbers > 1 indicate that the Xeon Phi is faster.

Image size	Scaling	Convolution
512x512	3.8	0.6
768x768	4.3	0.8
1152x1152	4.9	1.3
1728x1728	5.6	1.8
2592x2592	6.1	1.3
3888x3888	6.6	1.0
5832x5832	7.5	1.2

Note that the copying of data into the local memory has to be done explicitly in CUDA, and the convolution code is so written as to ensure that the copying can be done in parallel blocks, 64 bytes wide. This is analogous to the compiler on the Xeon Phi issuing prefetch instructions. The difference is that the programmer on the Xeon Phi does not have to explicitly schedule these transfers.

6.3. Efficiency

To assess the performance of Vector Pascal on the Xeon Phi, the three test programs were ran on four different architectures, for varying numbers of cores. Besides the Intel Xeon Phi 5110P, an Intel Xeon E5-2620 processor, an AMD FX-8120 processor and a Nvidia GeForce GTX 770 GPU were used. For each test timings were obtained over a set of images with varying input sizes.

6.3.1. Matrix efficiency Table V records the efficiency of the three algorithms on two machines. We take gcc with no optimisations enabled on the host as the base performance. The Vector Pascal matrix multiply code compiled on the Xeon does not vectorise, but with 12 threads it shows good acceleration. The single threaded gcc code on the Xeon Phi has very poor performance, running much slower than on one Xeon core. The same C code with OpenMP enabled using icc runs 75 times faster than the baseline. Pascal on the Xeon Phi runs 19 times faster than baseline.

6.3.2. Image scaling efficiency This can be vectorised on the Xeon Phi using gather instructions, whereas for earlier Intel or AMD processors the code can not be effectively vectorised. It is a particularly favourable basis on which to compare the Xeon Phi with standard Xeons. Table III shows that the performance gain on scaling ranges from 3.8 to 7.5. Figure 2 shows a plot of Pascal scaling timings against number of threads used.

- When using the same number of threads on a large image the Ivy Bridge Xeon host processor outperforms the Xeon Phi on scaling.
- The Xeon Phi however, overtakes the host Xeon once the number of threads is greater than 10. It goes on to achieve a peak performance an order of magnitude greater than the host Xeon.
- The Xeon Phi supports hyperthreading and shows a modest performance gain above 60 threads, but this gain is not monotonic, there is a local minimum of time at 100 threads with a local maximum at 120 threads.

Figure 3 shows the time taken to perform scaling and convolution against image size for four systems. It can be seen that for larger images convolution takes about the same time whatever architecture we use. For scaling, on which the gather instructions can be used, the Xeon Phi shows

Prepared using cpeauth.cls
DOI: 10.1002/cpe

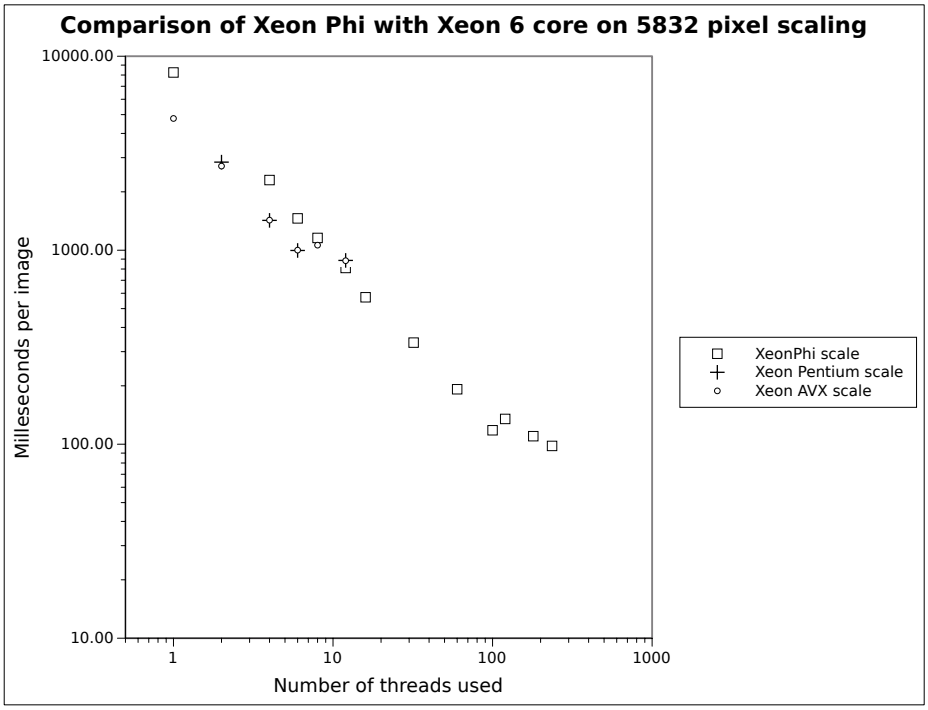


Figure 2. Log/Log plot of performance using Xeon Phi and Xeon 6 core processor for scaling images. The input images were 5832 pixels square. For each processor the maximum number of threads uses was twice the number of physical cores.

a considerable advantage over the two AVX machines. Also the Xeon Phi performs best on large images.

Since the work to be done scales as the square of the image edge, we would expect that on a Log/log scale the points for each machine should lie on a line with gradient +2. This indeed is almost what we observe for the Nvidia. But the Xeon Phi initially has a gradients of substantially less than 2. The Xeon Phi has a slope of 1.28, indicating that the machine is more efficient for large images than for small images. How can we explain this?

We attribute it to the overheads of starting large numbers of tasks, which is expensive for small images. The Nvidia, being a SIMD machine does not face this overhead. It can be seen that for larger images we observe a scaling slope closer to 2 for the Xeon Phi.

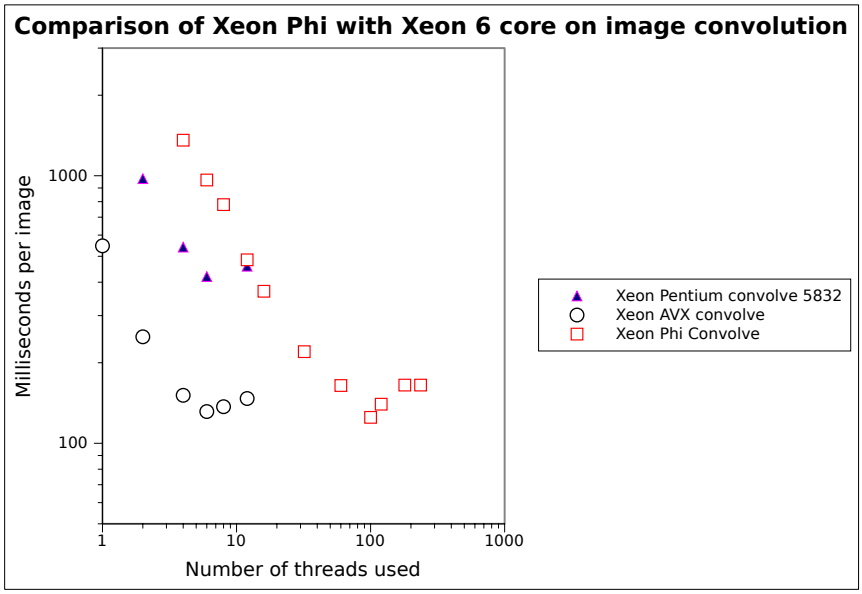
The GPU bears considerable costs in transferring data from the host machine into graphics memory to perform convolution or scaling. If one has a multi stage algorithm that can avoid repeated transfers of this sort by making use of the graphics memory to store intermediate results, then the GPU is likely to show an advantage over the Xeon Phi. Against this, the retention of a basic x86 architecture on the Xeon Phi means that it is possible to port code to it by simply recompiling the same source with a different set of command line flags.

6.4. Image convolution performance

Caching When convolving a colour image the algorithm must perform 60×32bit memory fetches per pixel: 3 colour planes $\times$ 2 passes $\times$ (5 kernel values + 5 pixel fetches). We can assume that half these memory access, to the kernel parameters, are almost bound to be satisfied from the level 1 cache, but that means that some 120 bytes have to be fetched from either main memory or level 2 cache for each pixel accessed, so the algorithm is heavily memory-fetch bound. An obvious optimisation is to process the image in slices such that two copies of each slice will fit into the level 2 cache; two copies because the algorithm uses a temporary buffer of the same size. The algorithm shown in section 6.4 is embedded within a recursive routine that, if the working-set

DOI: 10.1002/cpe

(a)



(b)

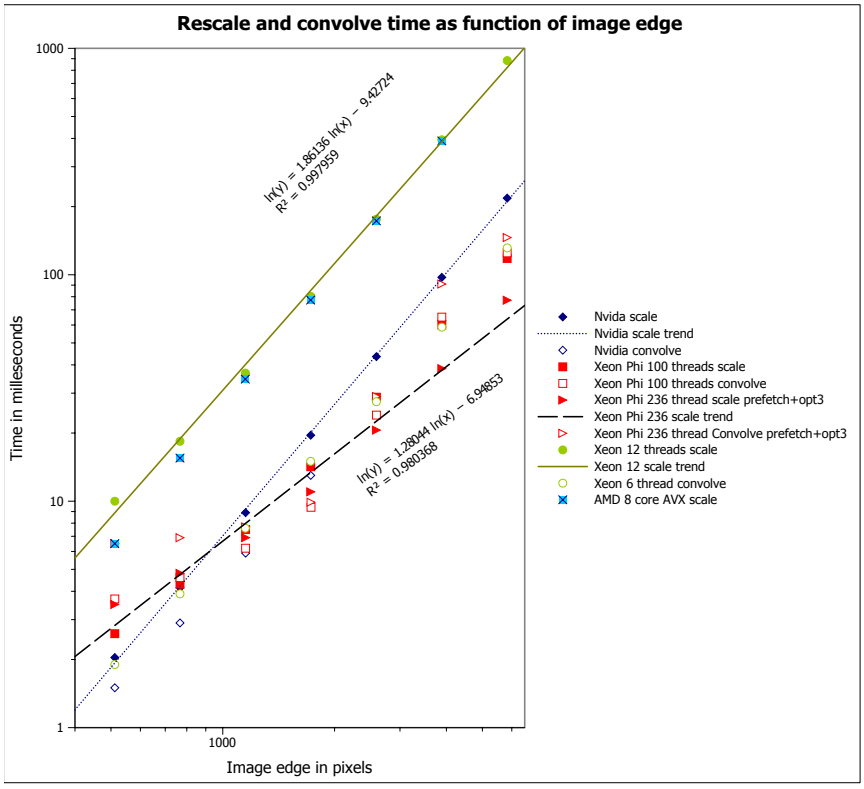


Figure 3. (a) Log/log plot of convolution times on Xeon Phi and 6 core Xeon host, against number of threads used. The input images were 5832 pixels square; (b) Log/log plot of scaling and convolution times on Xeon Phi, Nvidia, AMD 8 cores and a Xeon 6 cores, against image edge in pixels.

needed is greater than `cachesize`, will split the image horizontally and call itself sequentially on the upper and lower halves. We systematically varied the declared `cachesize` and found that performance peaked at 48MB. If the declared size working is too big performance falls off. When cache size is too small, performance again declines since the number of raster lines in the image is too small to provide a balanced workload for all threads.

DOI: 10.1002/cpe

Table IV. Scaling and convolution timings in milliseconds for 100 threads with different languages and settings, over square colour images of different sizes represented as single precision floating point numbers. Working set size for the convolution algorithm set at 32 MB. C++ code compiled using OpenMP and the `icpc` compiler.

Language Image edge size	C++ OpenMP	Pascal No_FMA	Pascal FMA	Pascal FMA Pre-fetch
Scaling	Time	Time	Time	Time
1152x1152	2.4	7.3	7.6	6.9
1728x1728	4.7	14.8	13.9	13.1
2592x2592	8.4	29.7	25.6	26.4
3888x3888	18.3	55.6	55.6	54.8
5832x5832	42.3	131	129	126.0
Convolution	Time	Time	Time	Time
1152x1152	0.85	7.9	6.9	7.3
1728x1728	1.99	11.5	10.5	10.6
2592x2592	4.15	22	22.9	20.8
3888x3888	8.77	72	70.7	65
5832x5832	19.56	145	140.7	126

Scaling and Convolution with Pre-fetching Table IV shows that prefetching actually slowed down performance in some cases when using 100 threads. Although in the larger images prefetching was consistently beneficial for small image sizes it produces erratic results. Intuitively we can see that for smaller images, threads may prefetch unnecessary data beyond the end of the scan lines they are tasked with. The surplus prefetching will generate extra bus traffic. We thus do not reproduce the results of Krishnaiyer et al [28] who found that pre-fetching was generally beneficial, including for convolution.

Krishnaiyer et al [28] published only results for the maximal number of threads supported by the hardware. We have found that performance does not necessarily peak at this point.

Enabling the pre-fetching flag for the C++ code did not make much difference in the timings. So, the best performance obtained by disabling pre-fetching. Vectorisation was enabled using pragma ivdep[‖].

As Figure 3 shows, the peak performance for image convolution occurs at 100 threads after which performance degrades.

On image convolution, an operation that can be vectorised well on either and AVX or a Phi instruction set, the performance gain of the Xeon Phi is much more modest than it was for resizing. We had expected the relative advantage to be greater in the case of image scaling since for this task the Xeon Phi can use gather instructions, which are not available on the ordinary Xeon.

If we contrast Figure 2 with Figure 3 we see that the Xeon Phi performs worse core for core as compared to the Xeon using vectorised AVX code and the Xeon running scalar Pentium convolution code. Even at the optimal number of threads (100) the Xeon Phi only slightly outperforms the host processor.

Fused Multiply-Accumulate The Xeon Phi has a couple of instructions that perform the calculation `a := a + b*c` on vectors of 16 floats. The very high performance figures quoted by Intel for the machine appear to be on the basis of these instructions which effectively perform 32 floating point operations per clock cycle per core. After enabling fused multiply accumulate instructions in the compiler we did see a noticeable acceleration of performance on both our image processing primitives as shown in Table IV.

In terms of comparing the parallel frameworks, OpenMP exhibited very good performance. We would like to add other parallel frameworks, such as GPRM [31] to the comparisons. We also plan

[‖]pragma ivdep tells the compiler to ignore assumed data dependencies that inhibit vectorisation.

DOI: 10.1002/cpe

Table V. Performance vs. complexity, 3 machines 4 compilers.

	cores	compiler	seconds	speedup	code_size in_char	relative size	perf/size ratio
Matrix Multiply(2048×2048)							
Xeon	1	gcc	53	100%	190	100%	100%
Xeon	1	vpc	42	126%	89	46%	269%
Xeon	12	vpc	7.1	746%	89	46%	1593%
Xeon Phi	1	gcc	541	10%	190	100%	10%
Xeon Phi	236	vpc	2.78	1906%	89	46%	4070%
Xeon Phi	240	icc	0.7	7571%	327	172%	4399%
Scaling(1728×1728)							
Xeon	1	gcc	0.528	100%	1072	100%	100%
Xeon	1	vpc	0.415	127%	837	78%	163%
Xeon	12	vpc	0.081	651%	837	78%	834%
Xeon Phi	100	vpc	0.013	4061%	837	78%	5201%
Xeon Phi	100	icc	0.0047	11234%	1108	103%	10869%
Nvidia	1536	cuda	0.019	2778%	2189	204%	1360%
Convolution(1728×1728)							
Xeon	1	gcc	0.35	100 %	1470	100.0%	100%
Xeon	1	vpc	0.35	100%	1435	97%	102%
Xeon	12	vpc	0.042	833%	1435	97%	854%
Xeon Phi	100	vpc	0.01	3500%	1435	97%	3585%
Xeon Phi	100	icc	0.002	17500%	1585	107%	16355%
Nvidia	1536	cuda	0.013	2692%	6181	420%	640%

to parallelise the image algorithms over the RGB planes to find out whether better performance can be achieved.

7. CONCLUSIONS AND FUTURE WORK

The Xeon Phi is a challenging machine to target because it combines a high level of SIMD parallelism along with a high degree of thread parallelism. Making the best use of both requires changes to well established compilation techniques. We have described an extended syntax for machine descriptions which allows machine specific vectorisation transformation to be used. Experiments with some simple image processing kernel operations indicate that, where these transformations can be used, the Xeon Phi shows a much greater advantage over an Ivy Bridge Xeon (Table III). Overall for floating point image processing operations the Xeon Phi also appears competitive with more conventional GPU architectures. However for complete application the use of NVIDIA GPU's requires a lot more code to be composed then required when processing images on the Xeon Phi. Although the Vector Pascal compiler in semantic terms fulfils better the requirements of ISO Pascal then commercial compilers it does not perform as well as Intel C in absolute performance. One of the reasons for this is the slightly misleading description of the Xeon Phi to be an in order machine. Inspecting Intel C compiler outputs show instruction reordering that only make sense if load instructions do not block, but on the contrary can overlap with subsequent register to register operations so long as these do not depend on the load. The current Vector Pascal for the Xeon Phi has no assembly instruction re-order pass. Future work on the Vector Pascal compiler for Xeon Phi would therefore require further, machine specific, assembly code optimisation.

ACKNOWLEDGEMENTS

We wish to express our gratitude for the sponsoring of our work by the FP7 Frame Work.

References

DOI: 10.1002/cpe

1. Siebert J, Urquhart C. C3d^{TM}: a novel vision-based 3-d data acquisition system. *Image Processing for Broadcast and Video Production*. Springer, 1995; 170–180.
2. Cockshott P. Vector pascal reference manual. *SIGPLAN Not.* 2002; **37**(6):59–81, doi:10.1145/571727.571737.
3. Cockshott W, Oehler S, Xu T, Siebert J, Camarasa GA. Parallel stereo vision algorithm. *Many-Core Applications Research Community Symposium 2012*, 2012. URL `http://eprints.gla.ac.uk/72079/`.
4. Turner T. Vector pascal a computer programming language for the array processor. PhD Thesis, PhD thesis, Iowa State University, USA 1987.
5. Iverson K. *A programming language*. Wiley: New York, 1966.
6. Chamberlain BL, Choi SE, Lewis C, Lin C, Snyder L, Weathersby WD. Zpl: A machine independent programming language for parallel computers. *Software Engineering, IEEE Transactions on* 2000; **26**(3):197–211.
7. Lin C, Snyder L. Zpl: An array sublanguage. *Languages and Compilers for Parallel Computing*. Springer, 1994; 96–114.
8. Snyder L. *A Programmer's Guide to ZPL*. MIT Press: Cambridge, 1999.
9. Perrott RH, Zarea-Aliabadi A. Supercomputer languages. *ACM Comput. Surv.* 1986; **18**(1):5–22, doi:10.1145/6462.6463.
10. Ewing A, Richardson H, Simpson A, Kulkarni R. *Writing Data Parallel Programs with High Performance Fortran*. Edinburgh ParallelComputing Centre, 1998.
11. Grelck C, Scholz SB. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters* 2003; **13**(3):401–412.
12. Scholz SB. —Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 2003; **13**(6):1005–1059.
13. Budd T. An APL Compiler for a Vector Processor. *ACM Transactions on Progeramming Languages and Systems* July 1984; **6**(3).
14. Chakravarty MM, Leshchinskiy R, Peyton Jones S, Keller G, Marlow S. Data parallel haskell: a status report. *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, ACM, 2007; 10–18.
15. Keßler CW, Paul WJ, Rauber T. Scheduling vector straight line code on vector processors. *Code GenerationConcepts, Tools, Techniques*. Springer, 1992; 73–91.
16. Perrott RH. A Language for Array and Vector Processors. *ACM Trans. Program. Lang. Syst.* Oct 1979; **1**(2):177–195, doi:10.1145/357073.357075. URL `http://doi.acm.org/10.1145/357073.357075`.
17. Formella A, Obe A, Paul W, Rauber T, Schmidt D. The SPARK 2.0 system-a special purpose vector processor with a VectorPASCAL compiler. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, vol. 1, IEEE, 1992; 547–558.
18. Cooper P. 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.
19. Jackson I. Opteron Support for Vector Pascal. Final year thesis, Dept Computing Science, University of Glasgow 2004.
20. Gdura YO. A new parallelisation technique for heterogeneous cpus. PhD Thesis, University of Glasgow 2012.
21. Cockshott W, Koliousis A. The SCC and the SICSA multi-core challenge. *4th MARC Symposium*, 2011. URL `http://eprints.gla.ac.uk/58983`.
22. Bik AJC, Girkar M, Grey PM, Tian X. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.* 2002; **30**(2):65–98, doi:10.1023/A:1014230429447.
23. Intel Corporation. *Intel Xeon Phi Product Family: Product Brief* April 2014. URL `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf`.
24. Chrysos G. Intel Xeon Phi^{TM} coprocessor (codename Knights Corner). *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.
25. Intel Corporation. *Intel®Xeon Phi™Coprocessor Instruction Set Architecture Reference Manual* 2012. URL `http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf`.
26. Cockshott P, Michaelson G. Orthogonal parallel processing in vector pascal. *Computer Languages, Systems & Structures* 2006; **32**(1):2–41.
27. Pater WH. *The Renaissance with an Introduction by Arthur Symons*, chap. The School of Giorgione. Modern Library, 1873.
28. Krishnaiyer R, Kultursay E, Chawla P, Preis S, Zvezdin A, Saito H. Compiler-based data prefetching and streaming non-temporal store generation for the intel (r) xeon phi (tm) coprocessor. *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, IEEE, 2013; 1575–1586.
29. Naishlos D. Autovectorization in GCC. *Proceedings of the GCC Developers Summit*, 2004; 105..118.
30. Peng Wu AW Alexandre Eichenberger. Efficient SIMD Code Generation for Runtime Alignment & Length Conversion. *CGO*, 2005.
31. Tousimojarad A, Vanderbauwhede W. The Glasgow Parallel Reduction Machine: Programming shared-memory many-core systems using parallel task composition. *EPTCS* 2013; **137**:79–94, doi:10.4204/EPTCS.137.7.