Automatic Vectorising Compilation

Paul Cockshott

University of Glasgow

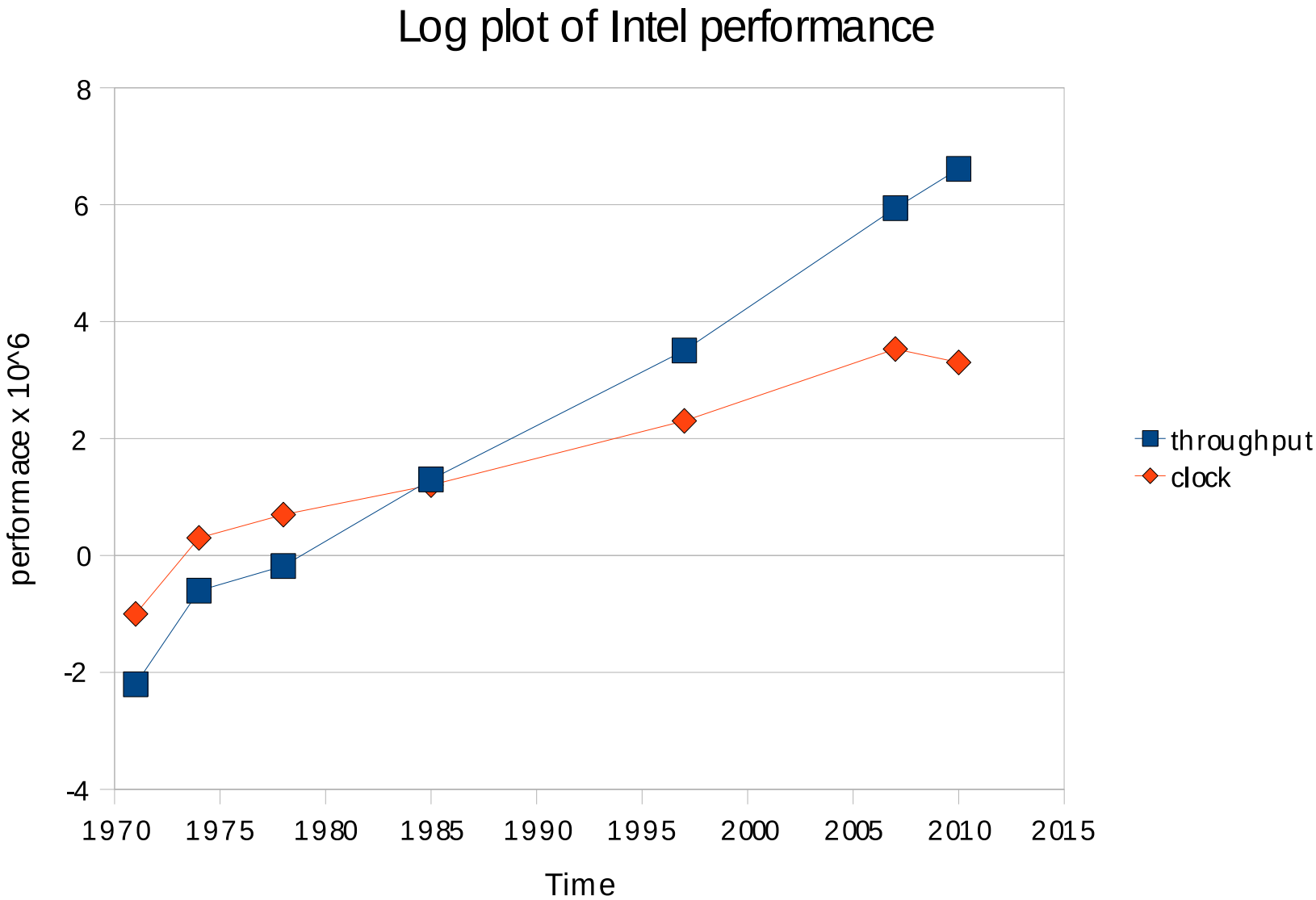
June 19, 2009

cpu	year	regs bits	clock Mhz	clock/ins	cores	speed Mips	datarate MB/s
4004	1971	4	0.1	8	1	0.0125	0.00625
8080	1974	8	2	8	1	0.25	0.25
8086	1978	16	5	8	1	0.33	0.66
386	1985	32	16	3	1	5.0	20
MMX	1997	64	200	0.5	1	400	3,200
Harpertown	2007	128	3400	0.25	4	54,400	870,400
Larrabee	2010	512	2000	0.5	16	64,000	4096,000

- Instruction speed $s_i = pc/i$ where p is processor cores, c is the clock and i clocks per instruction

- data throughput $d = s_i w$ where w is the register width in bytes

Growth of clock speed versus maximum throughput



Note how much of the increase in performance comes from increasing data parallelism.

Importance of Graphics Operations

The driving force in processor data throughput over the last decade has been graphics. We can see 4 stages in this evolution:

1. Intel introduce saturated parallel arithmetic for working on pixel arrays with the MMX instruction set.
2. AMD and Intel introduce parallel operations on 32 bit floats for working on co-ordinate transformations for 3D graphics in games.
3. Nvida and ATI develop programmable Miltie-core GPUs able to operate on 32 bit floats for games graphics.
4. Sony[1] and Intel[2] respond by developing general purpose multicore CPUs optimised for 32bit floating point vector operations.

[1] Cell

[2] Larrabee

Use the right types!

To get the best from current processors you have to be able to make use of the data-types that they perform best on : 8 bit saturated integers, and 32 bit floats. Parallel operations are possible on other data-types but the gain in throughput is not nearly so great.

Operate on whole arrays at once

The hardware is capable of operating on a vector of numbers in a single instruction

processor	byte	int	float	double
	Vector Lengths			
MMX	8	2	-	-
SSE2	16	4	4	2
Cell	16	4	4	2
Larrabee	64	16	16	8

Thus a programming language for this sort of machine should support whole array operations. Provided that the programmer writes the operation as operating on a whole array the compiler should select the best vector instructions to achieve this on a given architecture.

Use multiple cores

If the CPU has multiple cores the compiler should parallelise across these without the programmer altering their source code.

Working with Pixels

When operating with 8 bit pixels one has the problem that arithmetic operations can wrap round. Thus adding two bright pixels can lead to a result that is dark. So one has to put in guards against this. Consider adding two arrays of pixels and making sure that we never get any pixels wrapping round in C:

```
#define LEN 6400
#define CNT 100000
main()
{
  unsigned char v1[LEN],v2[LEN],v3[LEN];
  int i,j,t;
  for(i=0;i<CNT;i++)
     for (j=0;j<LEN;j++) {t=v2[j]+v1[j];if( t>255)t=255; v3[j]=t;}
}
[wpc@maui tests]$ time C/a.out
real    0m2.854s
user    0m2.813s
sys     0m0.004s
```

Doing it with the hardware

Intel provide an instruction PADDUSB which can add 8 pixels in one cycle simultaneously ensuring that there is no wrap around. If we code the same program up in assembler we get much better performance.

Assembler

```
SECTION .text ;
        global  main
LEN equ 6400
 main: enter  LEN*3,0
    mov ebx,100000      ; perform test 100000 times for timing
l0:
    mov esi,0       ; set esi registers to index the elements
    mov ecx,LEN/8       ; set up the count byte
l1: movq mm0,[esi+ebp-LEN]      ; load 8 bytes
    paddusb mm0,[esi+ebp-2*LEN] ; packed unsigned add bytes
    movq [esi+ebp-3*LEN],mm0    ; store 8 byte result
    add esi,8           ; inc dest pntr
    loop l1             ; repeat for the rest of the array
    dec ebx
    jnz l0
    mov eax,0
    leave
    ret
[wpc@maui tests]$ time asm/a.out
real    0m0.209s
user    0m0.181s
sys     0m0.003s
```

Why the difference?

- Semantic gap between source language and hardware capabilities.
  - C is a von Neuman single word at a time language.
  - Machine is a vector machine.
- Compiler tends to select the scalar instructions not the vector ones.
- Operator set of the language does not match the operator set of the hardware – it is less powerful than the hardware.

Now lets use an array language compiler

```
program vecadd;
type byte=0..255;
var v1,v2,v3:array[0..6399]of byte;
    i:integer;
begin
        for i:= 1 to 100000 do v3:=v1 +: v2;
        { +: is the saturated add operation }
end.
[wpc@maui tests]$ time vecadd
real    0m0.094s
user    0m0.091s
sys     0m0.005s
```

So the array language code is about twice the speed as the assembler.

Vector Pascal

I will focus on the language Vector Pascal, an extension of Pascal that allows whole array operations, and which both vectorises these and parallelises them across multiple CPUs. It was developed specifically to take advantage of SIMD processors whilst maintaining backward compatibility with legacy Pascal code. It stands in a similar relationship to ISO Pascal as FORTRAN 95 stands to FORTRAN 77.

Extend array semantics

Standard Pascal allows assignment of whole arrays. Vector Pascal extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. For example, given:

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

then we can write:

```
1. r1:= 1/2;
 2. r2:= r1*3;
 3. r1:= \+ r2;          {\⊙ reduces using operator ⊙}
 4. r1:= r1+r2[1];
```

Line 1 assign 0.5 to each element of r1.
Line 2 assign 1.5 to every element of r2.
In line 3, r1 gets the totals along the rows of r2.
In line 4, r1 is incremented with the corresponding elements of row 1 of r2.

Data reformatting

Given two con-formant matrices a, b
the statement

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

will transpose the matrix b into a.
For more general reorganisations you can permute the implicit indices thus

```
a:=perm[1,0] b ;{ equivalent to a:= trans b }
z:=perm[1,2,0] y;
```

In the second case z and y must be 3 d arrays and the result is such that z[i,j,k]=y[j,k,i]
Given `a:array[0..10,0..15] of t;` then

`a[1]`	`array [0..15] of t`
`a[1..2]`	`array [0..1,0..15] of t`
`a[][1]`	`array[0..10,0..0] of t`
`a[1..2,4..6]`	`array[0..1,0..3] of t`

Equivalent loops

These are defined to be equivalent to the following standard Pascal loops:

1'. `for` ι_0`:=0 to 7 do r1[`ι_0`]:=1/2;`

2'.
```
for ι0:=0 to 7 do
    for ι1:=0 to 7 do r2[ι0,ι1]:=r1[ι1]*3;
```

3'.
```
for ι0:=0 to 7 do
begin
    t:=0;
    for ι1:=7 downto 0 do t:=r2[ι0,ι1]+t;
    r1[ι0]:=t;
end;
```

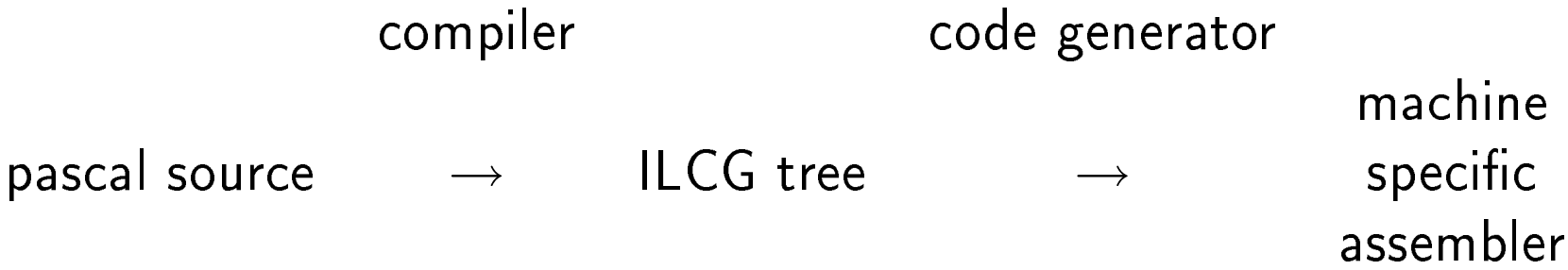
4'. `for` ι_0`:=0 to 7 do r1[`ι_0`]:=r1[`ι_0`]+r2[1,`ι_0`];`

The compiler has to generate an implicit loop. In the above ι_0, ι_1, t are temporary variables created by the compiler. The implicit indices ι_0, ι_1 etc are accessible to a coder using the syntax `iota[0]`, `iota[1]` etc.

Implicit mapping

Maps are implicitly defined on both operators and functions.
If `f` is a function or unary operator mapping from type T_1 to type T_2 and `x : array of` T_1
then `a:=f(x)` assigns an array of T_2 such that `a[i]=f(x[i])`.
Similarly if we have `g(p,q:`T_1`):` T_2,
then `a:=g(x,y)`
for `x,y:array of` T_1
gives `a[i]=g(x[i],y[i])`

Method of translation



ILCG

Intermediate language for code generation.
It is a machine level array language which provides a semantic abstraction of current processors.

1. We can translate source code into ILCG.
2. We can describe hardware in ILCG too.

This allows the automatic construction of vectorising code generators.

Translation from source to ILCG

Pascal

```
v3:=v1 +: v2;
```

ILCG

```
mem(ref uint8 vector ( 6400 ),  +(PmainBase, -25600) ):=
     +:(^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -12800))),
        ^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -19200))))
```

Note that all operation are annotated with type information, and all variables are resolved to explicit address calculations in ILCG – hence close to the machine, but it still allows expression of parallel operations.

^ is the dereference operation.

Key instruction specifications in ILCG

These are taken from the machine specification file gnuPentium.ilc

saturated add

```
instruction pattern PADDUSB(mreg m, mrmaddrmode ma)
 means[(ref uint8 vector(8))m :=
      (uint8 vector(8))+:((uint8 vector(8))^(m),
                          (uint8 vector(8))^(ma))]
 assembles ['paddusb 'ma ',' m];
```

vector load and store

```
instruction pattern MOVQL(maddrmode rm, mreg m)
  means[m := (doubleword)^(rm)]
  assembles['movq ' rm ',' m'\n prefetchnta 128+'rm];
instruction pattern MOVQS(maddrmode rm, mreg m)
  means[(ref doubleword)rm:= ^(m)]
  assembles['movq 'm ','rm];
```

Automatically build an optimising code generator

	ILCG Compiler		Java Compiler	
Pentium.ilc	$\longrightarrow$	Pentium.java	$\longrightarrow$	Pentium.class
Opteron.ilc	$\longrightarrow$	Opteron.java	$\longrightarrow$	Opteron.class

To port to new machines one has to write a machine description of that CPU in ILCG. We currently have the Intel and AMD machines post 486 plus Beta versions for the PlayStation 2 and PlayStation 3.

Vectorisation process

Basic array operation broken down into strides equal to the machine vector length. Then match to machine instructions to generate code.

ILCG input to Opteron.class

```
mem(ref uint8 vector ( 6400 ),  +(PmainBase, -25600) ):=
     +:(^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -12800))),
        ^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -19200))))
```

Assembler output by Opteron.class

```
   leaq          0,%rdx                ; init loop counter
l1:cmpq          $ 6399, %rdx
   jg            l3
   movq          PmainBase-12800(%rdx),%MM4
   prefetchnta 128+PmainBase-12800(%rdx) ; get data 16 iterations
                                         ;  ahead into cache
   paddusb     PmainBase-19200(%rdx),%MM4
   movq        %MM4,PmainBase-25600(%rdx)
   addq        $ 8,%rdx
   jmp         l1
l3:
```

Extend to Multi-cores

Vectorisation works particularly well for one dimensional data in which there is locality of access, since the hardware wants to work on adjacent words.
But newer chips have multiple cores. For the Opteron, the β version of our compiler will parallelise across multiple cores if the arrays being worked on are of rank 2 rather than 1.

2 D example.

```
procedure sub2d;
type range=0..127;
var x,y,z:array[range,range] of real;;
begin
        x:=y-z;
end;
```

Top level ILCG translation when compiled for a dual core Opteron

```
procedure(sub2d,
 procedure (label12 ... see below )
 post_job[label12,~(%rbp),1];    /* send to core 1 */
 post_job[label12,~(%rbp),0];    /* send to core 0 */
 wait_on_done[0];
 wait_on_done[1];
)
```

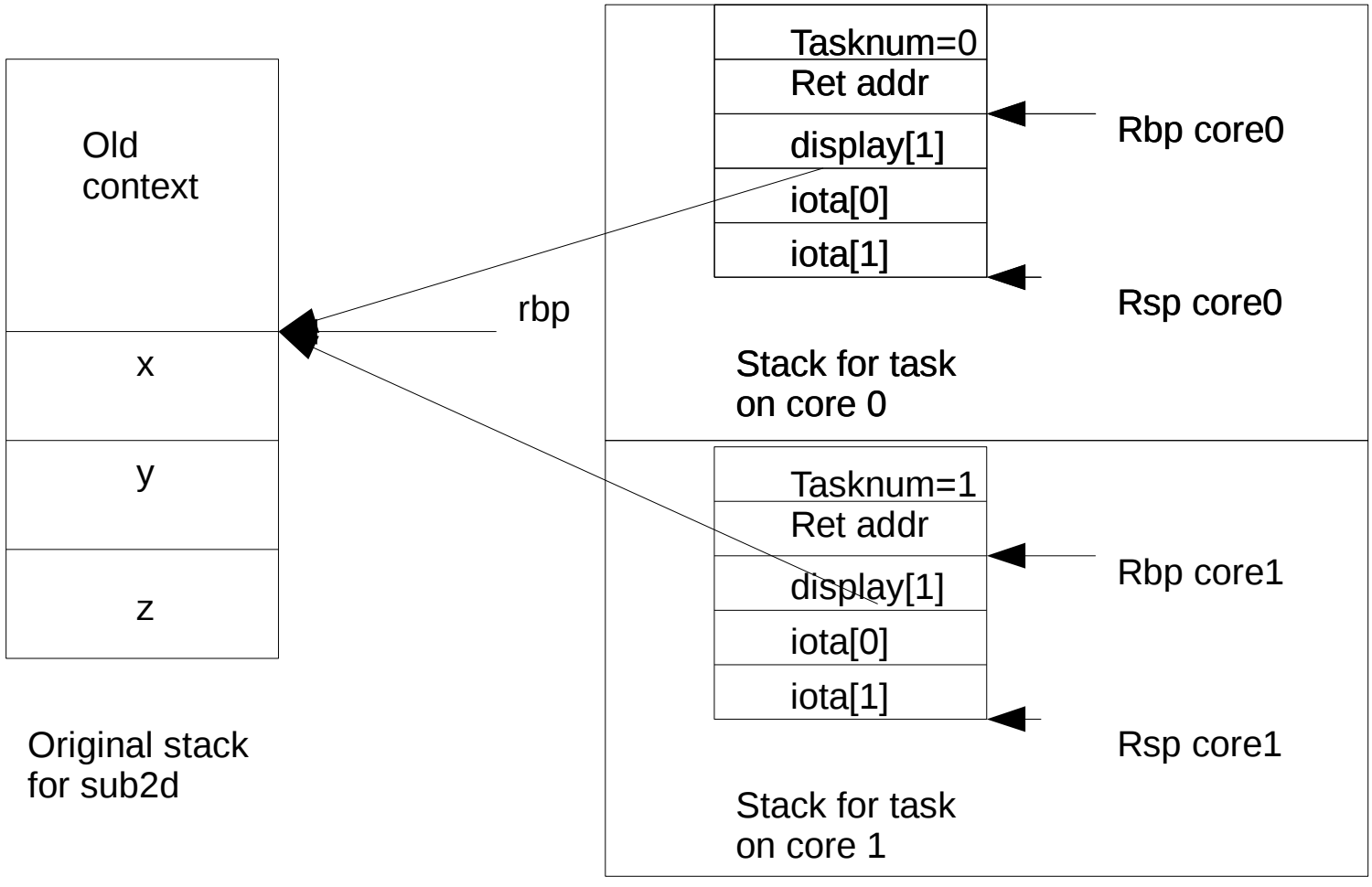
Individual task procedure

The statement x:=y-z is translated into a procedure that can run as a separate task, the ILCG has been simplified for comprehensibility!

```
procedure (label12 /* internal label*/ ,
 for(mem(+(^(%rbp),-24)),^(mem(+(^(%rbp),16))),127   ,   2,
    /*iota [0]                task number          limit  step*/
  var(mem(+(^(%rbp),-32))),/* iota[1] */
  for(mem(+(^(%rbp),-32)), 0    ,127,     4 ,
    /*iota [1]              start limit  step*/
   mem(ref ieee32 vector ( 4 ), /* x[iota[0],iota[1]] */
       +(+(*(^(mem(+(^(%rbp),-24))),512),
         +(*(^(mem(+(^(%rbp),-32))),  4),-131072)),
           ^(mem(+(^(%rbp),-8))))):=
      -(^(mem(ref ieee32 vector ( 4 ),/* y[iota[0],iota[1]] */
        +(+(*(^(mem(+(^(%rbp),-24))),512),
          +(*(^(mem(+(^(%rbp),-32))),  4),-196608)),
            ^(mem(+(^(%rbp),-8))))))),
        ^(mem(ref ieee32 vector ( 4 ),/* z[iota[0],iota[1]] */
          +(+(*(^(mem(+(^(%rbp), -24))),512),
            +(*(^(mem(+(^(%rbp), -32))),  4),-262144)),
              ^(mem(+(^(%rbp),-8)))))))))),

)
```

Memory organisation



Practical Exercise

The exercise is to write a parallel image blurring program in Vector Pascal.
The blurring program should use a simple 3x3 separable kernel applied first to the rows and then the columns.
You should use the type `pixel` for your arithmetic described on page 17 of the manual.
You should read in a bmp file, down loadable as www.dcs.gla.ac.uk/~wpc/testimage.bmp. This is a 256 by 256 pixel test image. You should output the file blur.bmp as your result.
Look at page 63 of the manual for commands on how to invoke the compiler.
If you have a file called blur.pas you should type
vpc blur -cpuOpteron -cores4
to compile a program for the Opteron instruction set.

Outline Algorithm

This should be expressed in whole array operations if possible.

Horizontal blur

Recall we can blur horizontally if

$p'_i = 0.25p_{i-1} + 0.25p_{i+1} + 0.5p_i$

So to form temporary images b, c from image a thus:

$b = 0.5a, c = 0.25a$

the horizontal blurred image is then

$b + c$ shifted left one $+c$ shifted right one

2D blur

1. blur horizontally
2. transpose
3. blur horizontally
4. transpose

Graphio

You should use the library Graphio to read and write the files, it has the interface.

```
type
 image(maxplane,maxrow,maxcol:integer)=
 array[0..maxplane,0..maxrow,0..maxcol]of pixel;
 pimage=^image;

procedure storebmpfile(s:string;var im:image) ;
(*! This procedure will store an image im as
a Microsoft .bmp file with name s *)
function loadbmpfile(s:string;var im:pimage):boolean ;
(*!   This function returns true if it has sucessfully
   loaded the bmp file  s . The image pointer  im
is initialised to point to an image on the heap. The program
explicity discard the image after use by calling  dispose .
```

Example program declarations

```
program blur; uses graphio;
 const sourcefile='testimage.bmp';
       destfile ='blur.bmp';
       red=0;green=1;blue=0;
       maxrow=255;maxcol=255;
 type { declare a type for a plane of the image }
      plane (ymax,xmax:integer) =
      array[0..ymax ,0..xmax ] of pixel;
 var inbuffer,outbuffer:pimage;
     colour  :integer;
  procedure blurplane(var inplane,outplane:plane);
  var temp:^plane;
  begin
  { you fill this in as the exercise }
  end;
```

Program body

```
begin
   if loadbmpfile(sourcefile,inbuffer ) then
   begin
     new(outbuffer,2, maxrow,  maxcol);
     for colour := red to blue do
       blurplane(inbuffer^[colour ],outbuffer^[colour]);
     storebmpfile(destfile,outbuffer^);
     dispose(outbuffer);
    end
    else writeln(sourcefile, ' not found');
end.
```