Automatic Vectorising Compilation At Glasgow University

Paul Cockshott

University of Glasgow

January 18, 2011

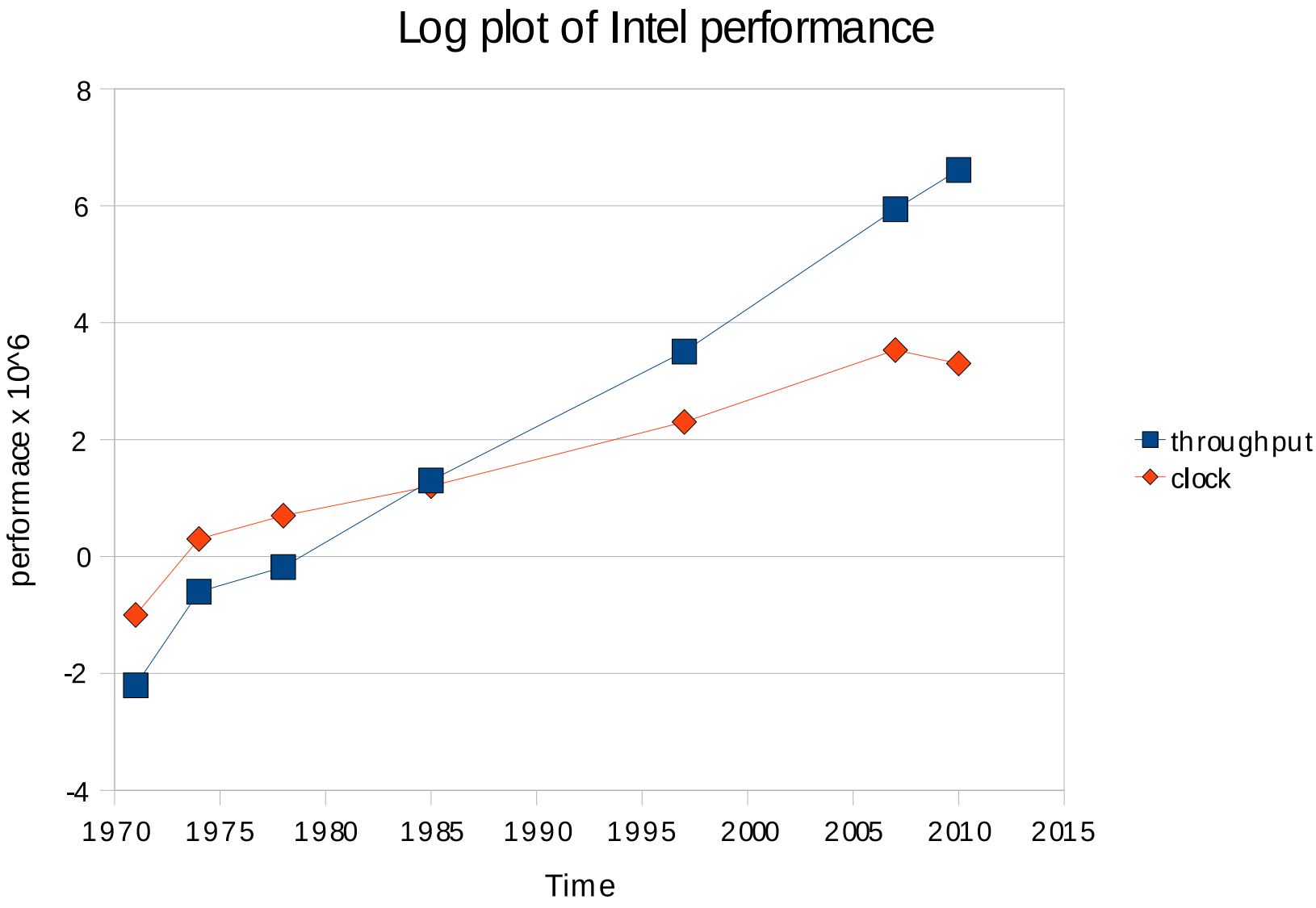
Summary

- Vector Pascal : a sort of merger of APL and Pascal, which is targeted at SIMD multi-cores and uses the most developed of our compiler technologies.
- Our FORTRAN 95 to IBM Cell experiments.

The growth of data parallelism

CPU	year	regs bits	clock MHz	clock/ins	cores	speed MIPS	data rate 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



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

Key points: use the wide data registers, use multiple cores.

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. Nvidia 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 multi core 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
```

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
```

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.
- It is heavily influenced by languages like J, APL and ZPL.
- It aims to be a complete programming language - super set of ISO Pascal, and to semantically extend all operations to data parallel form, and then automatically parallelise them automatically at compile time and run time.

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;
s:real;
```

then we can write	to mean
`r1:= 1/2;`	assign 0.5 to each element of `r1`
`r2:= r1*3;`	assign 1.5 to every element of `r2`
`r1:= r1+r2[1];`	add row 1 of `r2` to `r1`
`s:= \ + r1`	$s \leftarrow \sum_{\iota_0} r[\iota_0]$
`r1 = \ * r2`	$\forall_{\iota_0} r1[\iota_0] \leftarrow \prod_{\iota_1} r2[\iota_0, \iota_1]$
`r1 := r1 * iota[0]`	$\forall_{\iota_0} r1[\iota_0] \leftarrow r2[\iota_0] * \iota_0$

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

- `a: array[` T_o `] of` T_2
- `g(p,q:`T_1`):` T_2,
- `x,y:array[`T_o`] of` T_1 ,
- `B: array[`T_1`] of` T_2

statement	means
`a:=f(x)`	$\forall_{i \in T_o}$`a[i]=f(x[i])`
`a:=g(x,y)`	$\forall_{i \in T_o}$`a[i]=g(x[i],y[i])`
`a:=B[x]`	$\forall_{i \in T_o}$`a[i]=B[x[i]]`

Support array slices and dynamic arrays

- ISO Pascal only supported arrays whose size was known at compile time.
- ISO-extended Pascal93 allows array sizes to be dynamically defined.
- Vector Pascal extends this with array sections in Algol68 or Fortran95 style.

```
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
```

Sectioning in graphics

```
type window(maxrow,maxcol:integer)=
      array[0..maxrow,0..maxcol]of pixel;
procedure clearwindow(var w:window);
begin

    w:=black;

end;
var screen:array[0..1023,1..800] of pixel;
begin
   clearwindow(screen[20..49,0..500]);
end;
```

Data reformatting

Given two conformant 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]

Convolution example

procedure *pconv* (**var** *theim* :*tplain* ;*c1* ,*c2* ,*c3* :*real*);
var
tim :**array** [0..*m* ,0..*m*]**of** *pixel* ;
Let *p1*, *p2*, *p3* $\in$ array[0..m]of pixel;
begin
$p1 \leftarrow c1$;
$p2 \leftarrow c2$;
$p3 \leftarrow c3$;
$tim_{1..m-1} \leftarrow theim_{0..m-2} \times p1 + theim_{1..m-1} \times p2 + theim_{2..m} \times p3$;
$tim_0 \leftarrow theim_0$;
$tim_m \leftarrow theim_m$;
$theim_{0..m,1..m-1} \leftarrow tim_{0..m,0..m-2} \times p1 + tim_{0..m,2..m} \times p3 + tim_{0..m,1..m-1} \times p2$;
$theim_{0..m,0} \leftarrow tim_{0..m,0}$;
$theim_{0..m,m} \leftarrow tim_{0..m,m}$;
end ;

Performance

- The convolution example in Vector Pascal runs in 32ms on an image of 1024x1024 pixels
- A C (gcc) implementation of the convolution operation takes 352ms on the same image
  - Both done on the same computer (Fujitsu Laptop, with Centrino Duo, dating from 1996).
- Key factors
  - removal of all array temporaries by the compiler,
  - the evaluation of the code in SIMD registers across both cores.

Algorithm	Implementation	Target Processor	MOPs
conv	Borland Pascal	286 + 287	6
	Vector Pascal	Pentium + MMX	61
	DevPascal	486	62
	Delphi 4	486	86
pconv	Vector Pascal	486	80
	Vector Pascal	Pentium + MMX	820

Measurements done on a 1 core 1GHz Athlon, running Windows 2000.

Similar gains on eliminating set temporaries

Maxlim	1 secs Vector Pascal	2 secs Prospero Pascal	3 ratio
20000	0.73	42	57 to 1
25000	0.91	63	69 to 1
40000	1.30	315	242 to 1

```
begin
    primes← [2..maxlim];
    k← 1;
    for i in primes do
    begin
        j← i × (k + 1);
        while j ≤ maxlim do
        begin
            primes← primes - [j];
            j← j + i;
        end ;
    end ;
    primes← primes + [1];
    for i in primes do WRITELN(i);
end .
```

Seive of Erastostenes

Measurements taken using my old 700 MHz Trans-Meta Crusoe laptop. Vector Pascal compiled to the MMX instruction-set. Columns 1 and 2 give total run time in seconds to find the primes excluding time to print them. Column 3 shows the speed ratio between the two compilers.

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	$\rightarrow$	Pentium.java	$\rightarrow$	Pentium.class
Opteron.ilc	$\rightarrow$	Opteron.java	$\rightarrow$	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 and Pentium family, the compiler will parallelise across multiple cores if the arrays being worked on are of rank 2 rather than 1.

2 D example.

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

Suppose we want to run this on an Opteron that has 2 cores and 4 way parallelism within the instructions we compile as follows

```
$ vpc prog -cpuOpteron -cores2
```

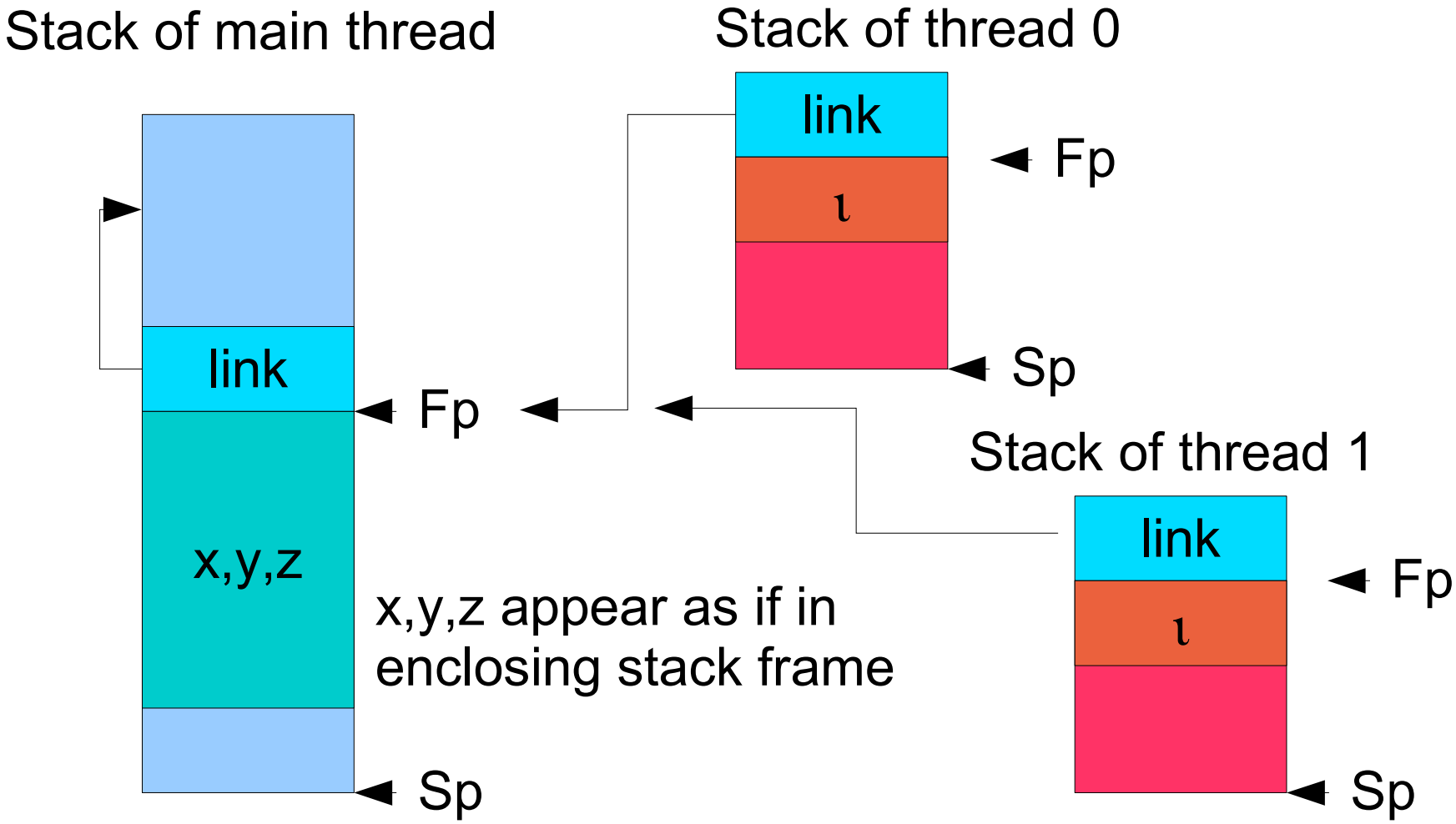
and it performs the following transformation

Individual task procedure

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

```
procedure sub2d;
type range=0..127;
var x,y,z:array[range,range] of real;
  procedure label12(start:integer);
  var ι;array[0..1] of integer;
  begin
    for ι0:=start to range step 2 do
      for ι1:=0 to range step 4 do
       x[ι0,ι1..ι1+3]:=y[ι0,ι1..ι1+3]-z[ι0,ι1..ι1+3];
  end;
begin
 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};
end;
```

Memory structure



Parallelism on Heterogeneous Multiprocessors

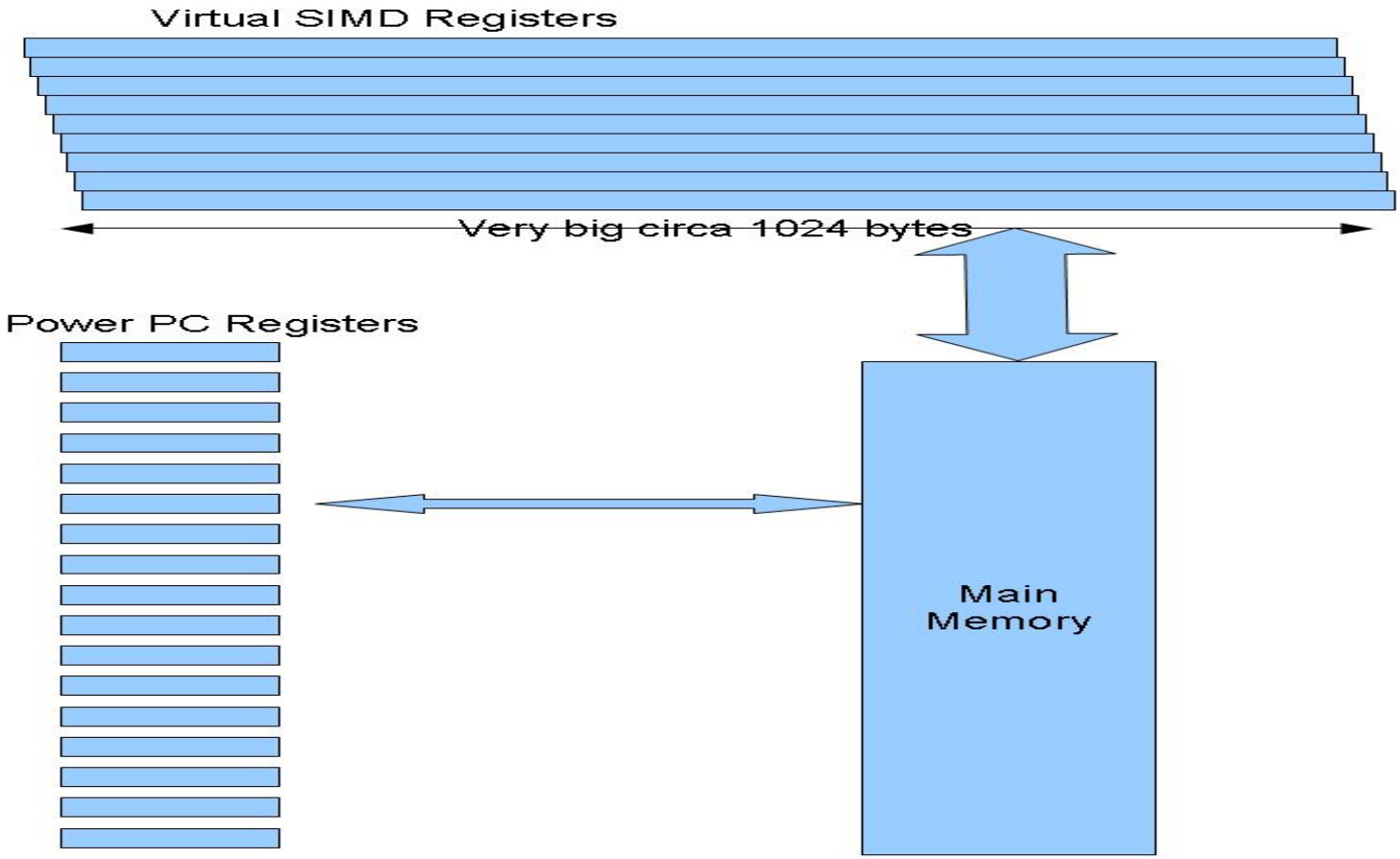
Cell has

- Two way threaded main processor 128 bit Power PC
  - main memory
- 8 vector processors (SPE) 128 bits
  - run in private 256k memory each
  - no instruction access to main memory
  - dma block transfers to/from main memory
- Main and vector processors use different instruction sets

We have tried 2 approaches to compiling to this

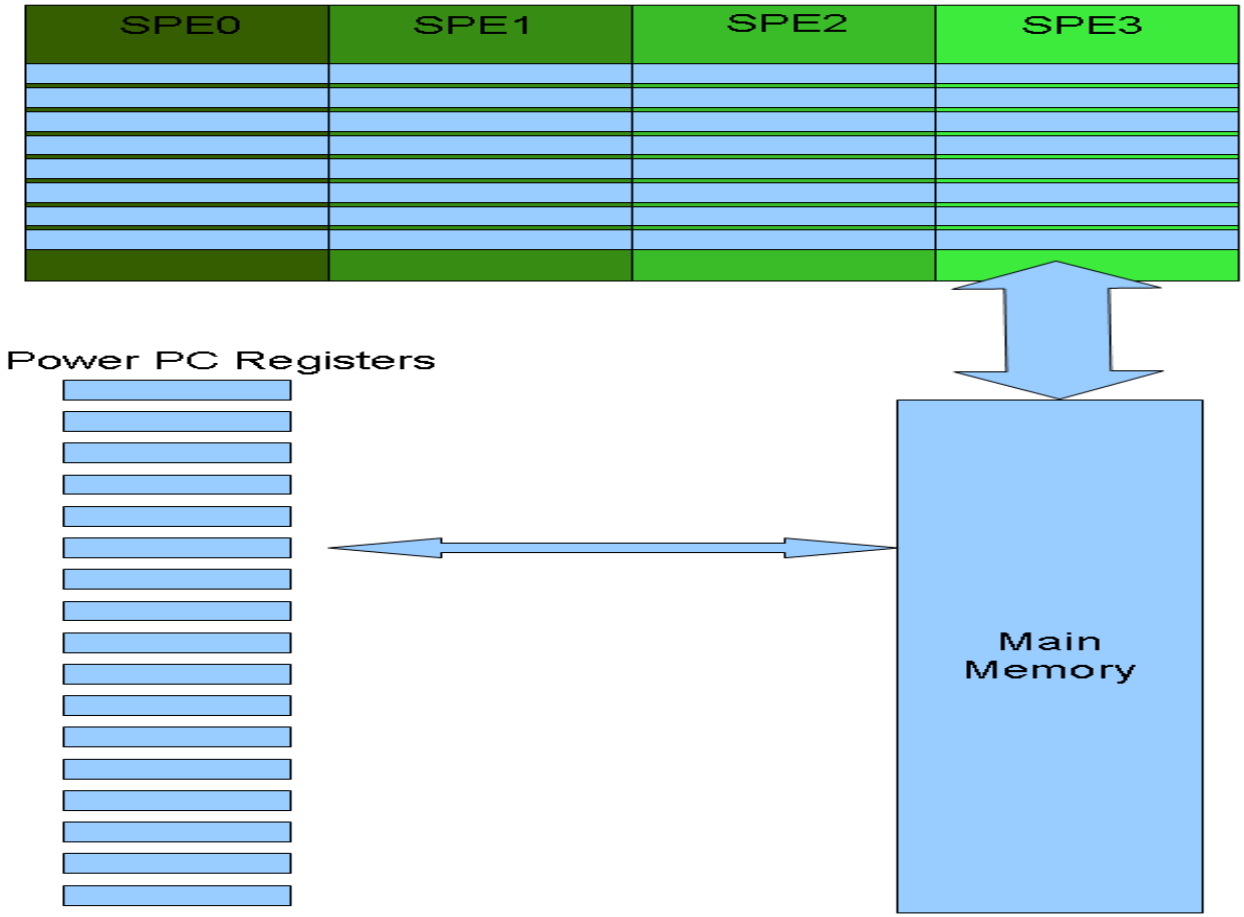
- Virtual SIMD machines
- Mapping to Offload blocks in C++

Virtual SIMD



This is the model we compile to: SIMD with load store architecture

Implemented Split over SPEs



How do we compile to it?

1. Augment the ILCG specification of the Power PC with additional registers
2. Augment with semantic specification of additional OP codes
3. Automatically build parallelising code generator from the description
4. Implement SIMD op codes as loads of messages into the SPE input fifos, which act as the instruction fetch buffers for the virtual machine
5. Implement the machine as interpreter running in parallel in n SPEs each acting on 1/n th of a virtual register
6. Then just use the existing unmodified Vector Pascal compiler

We have also demonstrated that the same technique can be used to compile VP to a virtual SIMD machine on NVIDIA cards - in this case performance gain is less.

1) Augment the ILCG specification.....

```
/* Defining SPE register */
define(VECLEN,1024)
register ieee32 vector(VECLEN) NV0 assembles[' 0'];
register ieee32 vector(VECLEN) NV1 assembles[' 1'];
...
pattern nreg means[NV0|NV1|.... ];


instruction pattern speLOADFLT( naddrmode rm, nreg r1)
   means[ r1 :=(ieee32 vector(VECLEN))^(rm)]
   assembles['li 3, ' r1
          '\n la 4,0(' rm ')'
          '\n bl speLoadVec'];
instruction pattern speADDFLT(nreg r0,nreg r1 )
   means[r0:= +(^(r0),^(r1))]
   assembles['li 3, ' r0
          '\n li 4,' r1
          '\n bl speAddVec'];
```

4) Implement SIMD op codes as loads .....

```
void speLoadVec(unsigned int reg,unsigned int mem ) {
   msgs[0]=(LOAD<<24)+((reg<<24)>>24);
   broadCast2Msg(mem);
}
```

Speedups versus the host processor

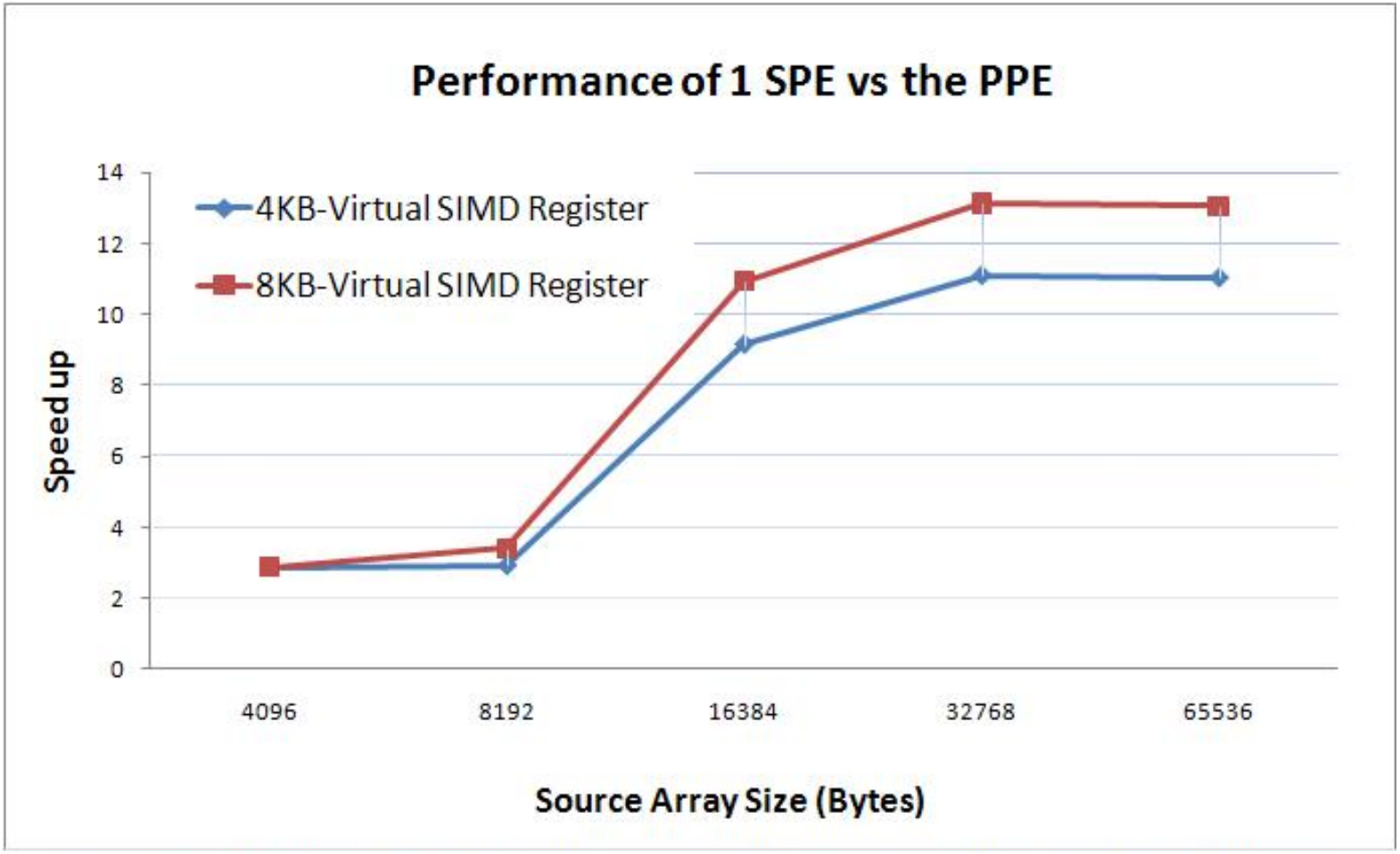
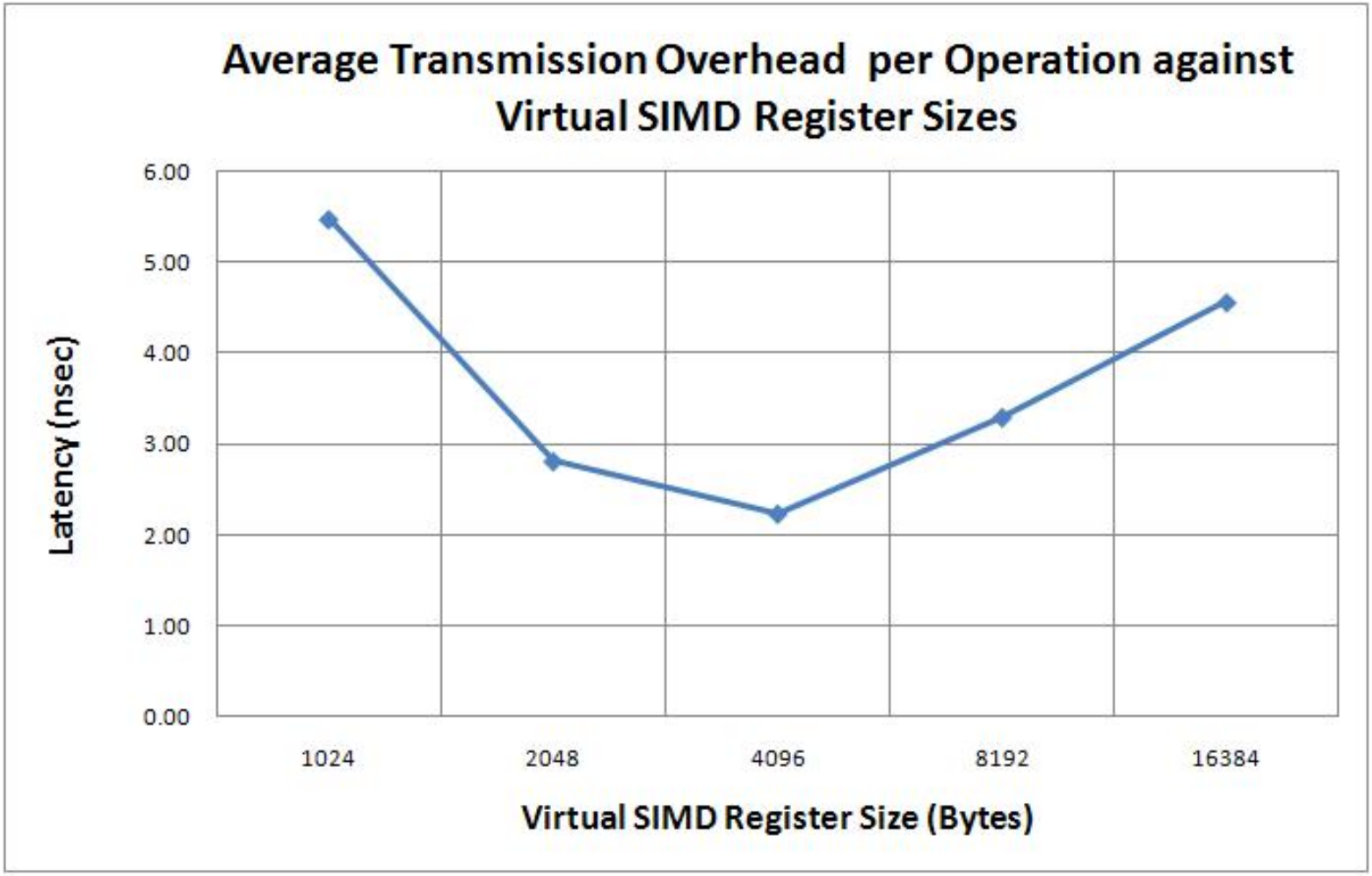
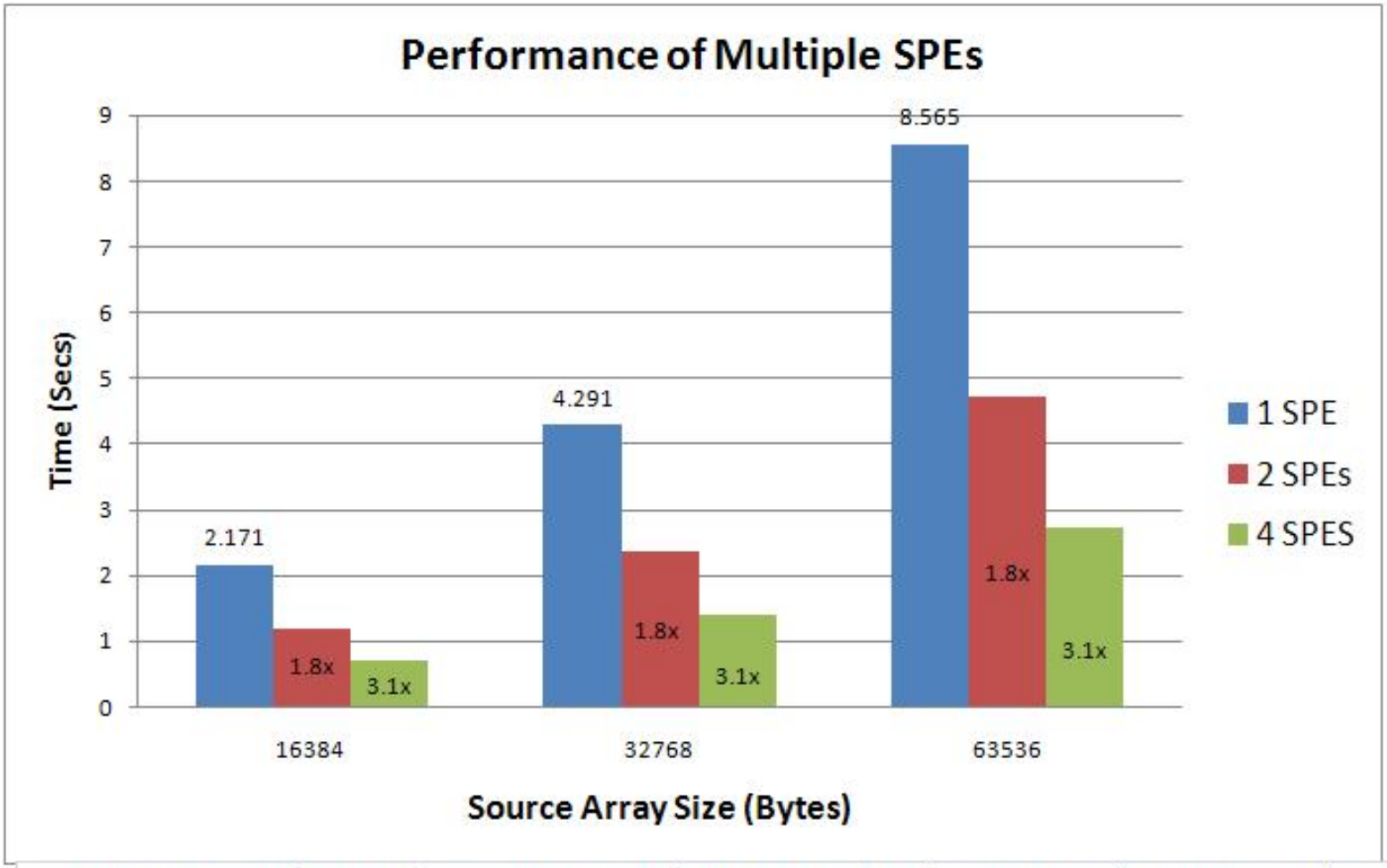


Figure 1. Speed up the execution time of C = C + A* B (single-precision real) on one SPE versus the PPE using two different virtual SIMD register sizes (4KB & 8KB)

Explore optimal SIMD register size



Speedups with multiple SPEs used



Execution time of C = C + A* B using a 4 KB virtual SIMD register (single-precision real, 100K Iterations)

e# : Fortran to C++

- Fortran; not **FORTRAN**
- Targeting Offload C++
- The e# compiler
- Benchmarks

Fortran Overview

- Originally developed in the 1950s at IBM by John Bachus and others
- An evolving language
  - Fortran 2003 : Cray (apart from "International Character Sets")
  - Fortran 2008 standard due for final ratification 2011
- A High Performance Language
  - Comparable or better performance than C
  - Good compatibility with Open-MP and MPI
  - Explicit pointer targets; unboxed types; fixed loop iterations

Array Expressions in Fortran 90

- Implicitly Parallel
- Result independent of order of evaluation
- Evaluate right-hand side first
- First class arrays
- Mandatory array procedures; e.g. size, lbound
- Elemental operations: e.g. sin, cos, (+), (-)

```
pure function foo(a,b) result(c)
 real :: a(64,64), b(0:63,0:63)
 real :: c(size(a,1),size(a,2))
 real :: s
 c = (a * 2) + sin(b) + s
 c = matmul(transpose(c),c)
end function
```

Fortran to C++ translation

- Compiler written in Haskell
  - SYB, Parsec and Pretty packages
- ACL LANL Chasm Interop
  - No standard ABI for Fortran Arrays (dope vectors)
  - Template ArrayT<C,T,R,D> class interface
- Fortran run time library abstraction layer
  - API layer uses C++ function overload resolution to choose e.g. `_gfortran_matmul_r8` given `matmul(a,b)`
  - If using the Gfortran (GCC) run time library
- Backends: Offload C++ or Pthreads

Parallelising Array Expressions

- Parallelise *elemental* ops.
- Hoist non-elemental terms.
- Partition across outermost array dimension
- Generated SPU code manages staged data transfer
- Strided data access also possible (array sectioning)
- Object code created using Offload C++ compiler

`a,b,c,d` are conforming; e.g. `b` could be scalar
`f` is a non-elemental function

```
a  = b + sin(b) - f(c * d)
```

```
t1 = b + sin(b) - f(c * d)
a  = t1
```

```
t2 = f(c * d)
t1 = b + sin(b) - t2
a  = t1
```

```
t3 = c * d
t2 = f(t3)
t1 = b + sin(b) - t2
a  = t1
```

Mandelbrot Benchmark

1024 x 1024

10 iterations

Kernel times

ppu-gfortran 4.1.1	27.4 secs.
e♯ (ppu-g++ 4.1.1)	35.5 secs.
e♯ (offloadcpp 1.0.4)	2.7 secs.

3.2GHz CellBE
Fedora Core 7

gfortran 4.3.4	5.1 secs.
g95 0.94	5.7 secs.
e♯ (g++ 4.3.4)	5.2 secs.

Intel Core2 Duo E8400 3GHz
Windows XP (Cygwin)

Conclusions

- It is possible to effectively automatically parallelise data parallel imperative languages accross, SIMD, multi-core and hetrogenous multi-core machines.
- Significant speedups can be attained.
- The resulting code can be retargeted without any changes to the source code
- Parallelising compilers can be retargeted without any change to the main body of the compilers using automatic code generator generator techniques.