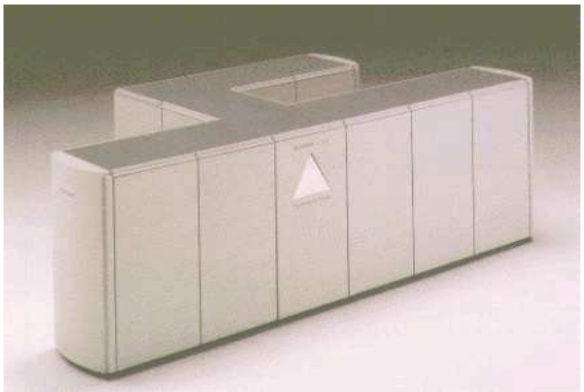
Vector Computers: the Dinosaurs of Computing?

Dr MJ Rutter
mjr19@cam.ac.uk

27th February 2001



S-3600

The Birth of the HPCF

Cambridge's High Performance Computing Facility was founded by a consortium of University Departments, including DAMTP, Physics, Engineering, Chemistry, Materials Science and others.

It has nothing to do with the Computing Service or Prof. Hawking.

Its first computer arrived in 1996: a Hitachi S3600 vector computer kindly donated by Hitachi Ltd. This machine was named 'turing' and is the main subject of this talk. The model was launched c. 1993, and the computer we received had a notional value of £1m.

Later in 1996 Cambridge bought a 96 processor Hitachi SR2201 parallel computer, which is currently still running, now with 256 processors. It is due to be decommissioned this summer.

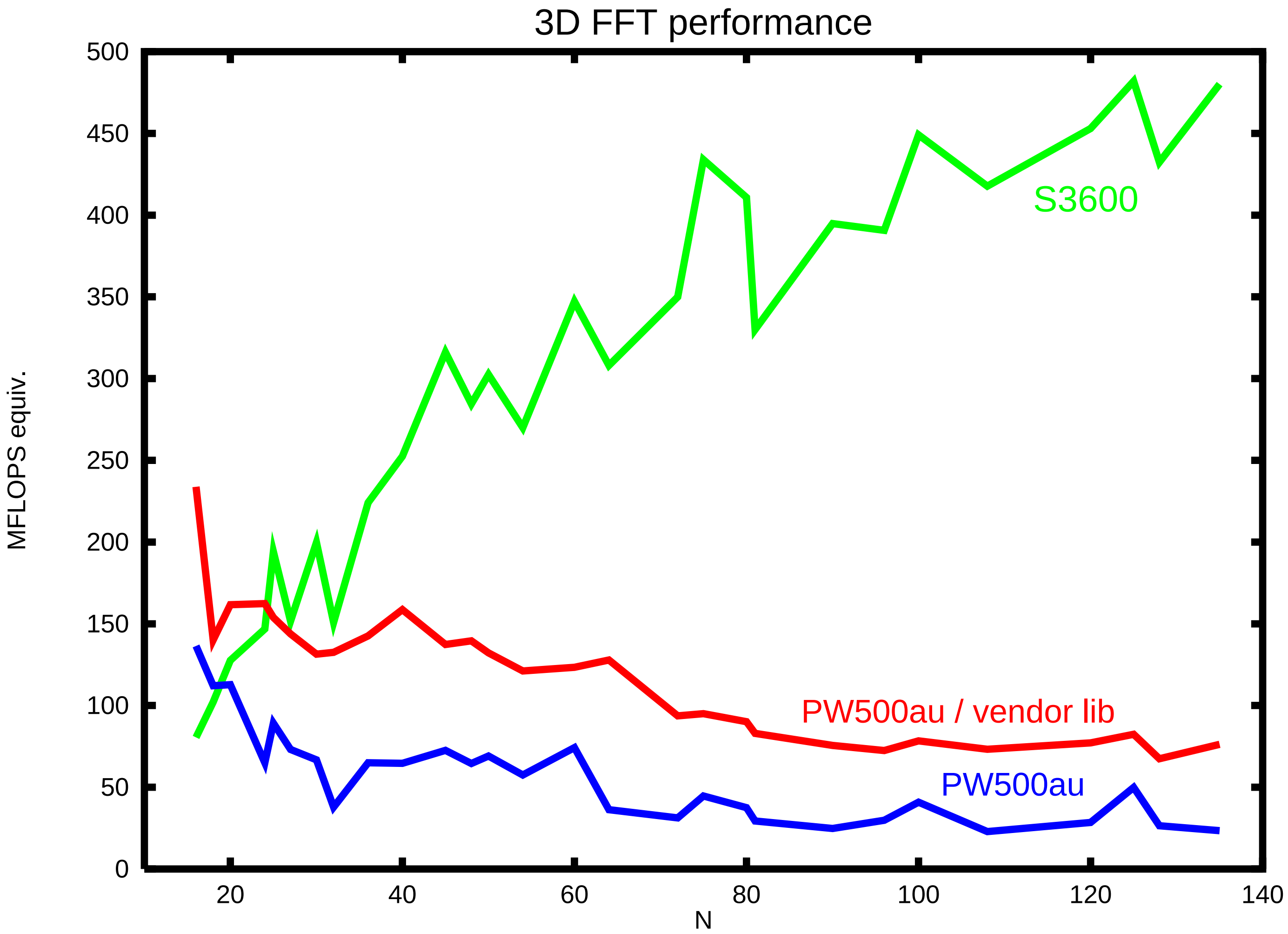
In the summer of 1997 I joined the HPCF as a Computer Associate, and I left a year later.

What is a Supercomputer?

Hard to define beyond 'big, fast and expensive.'

Let us first consider the 'fast' aspect, whilst noting that with a peak performance of 2 GFLOPS and just 1 GB of memory, in some ways turing was not enormously fast even in 1996 – workstations even then had peak speeds of 1 GFLOPS and could cope with 1 GB of memory.

Running code on an S3600 is rather hard at the moment: I am unaware of any still in existance, I know there are none in the UK or Europe. Thus the following examples are chosen from exstant data. . .


3D FFT performance
MFLOPS equiv.
N
0
50
100
150
200
250
300
350
400
450
500
20
40
60
80
100
120
140
S3600
PW500au / vendor lib
PW500au

Linpack

(Solve nxn system of simultaneous equations by naive Gaussian elimination)

Linpack 1000x1000

	Theoretical Peak (MFLOPS)	Achieved (MFLOPS)
S3600	2000	1035
DEC 500/500	1000	50
PentiumPro	200	21

[Beware: vendor results for this benchmark use hand-optimised blocking algorithms, and usually achieve at least 70% of peak. The S3600 manages 1672 MFLOPS under these conditions.]

What went wrong?

For something surely did for those workstations.

Memory

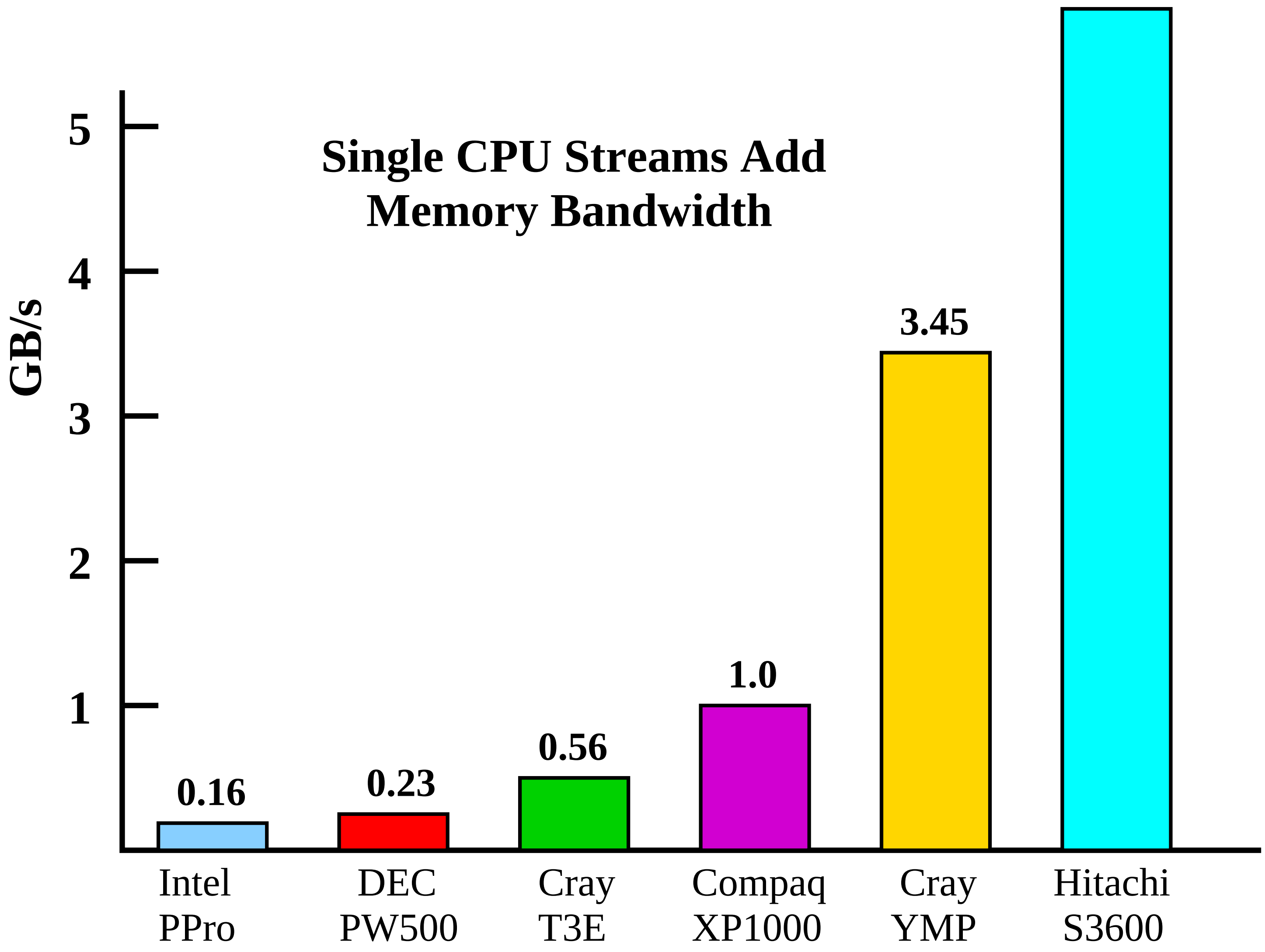
Performance OK whilst data set is held in cache, poor when main memory is accessed.

Pipelining

Given that a floating-point add or multiply typically takes three clock-cycles to complete, in order to achieve 2 MFLOPS per MHz six instructions must be executed simultaneously, and these six instructions must be free of data dependencies.

```
a=b+(c*d)
```

The '+' cannot be started until the result of the '*' is known.


Single CPU Streams Add
Memory Bandwidth
GB/s
5
4
3
2
1
0.16
0.23
0.56
1.0
3.45
Intel
PPro
DEC
PW500
Cray
T3E
Compaq
XP1000
Cray
YMP
Hitachi
S3600


10
9
8
7
6


12
11
11.3

Required Bandwidths

All serious work uses 8 byte floating point numbers.

Thus to achieve 2 GFLOPS with a 12 GB/s memory bandwidth, one needs 1.5 fp operations per load/store.

To achieve 1 GFLOPS with a 250 MB/s memory bandwidth, one needs 32 fp operations per load/store.

This makes it very hard to get certain machines up to their peak speed.

Part II: I want it NOW!

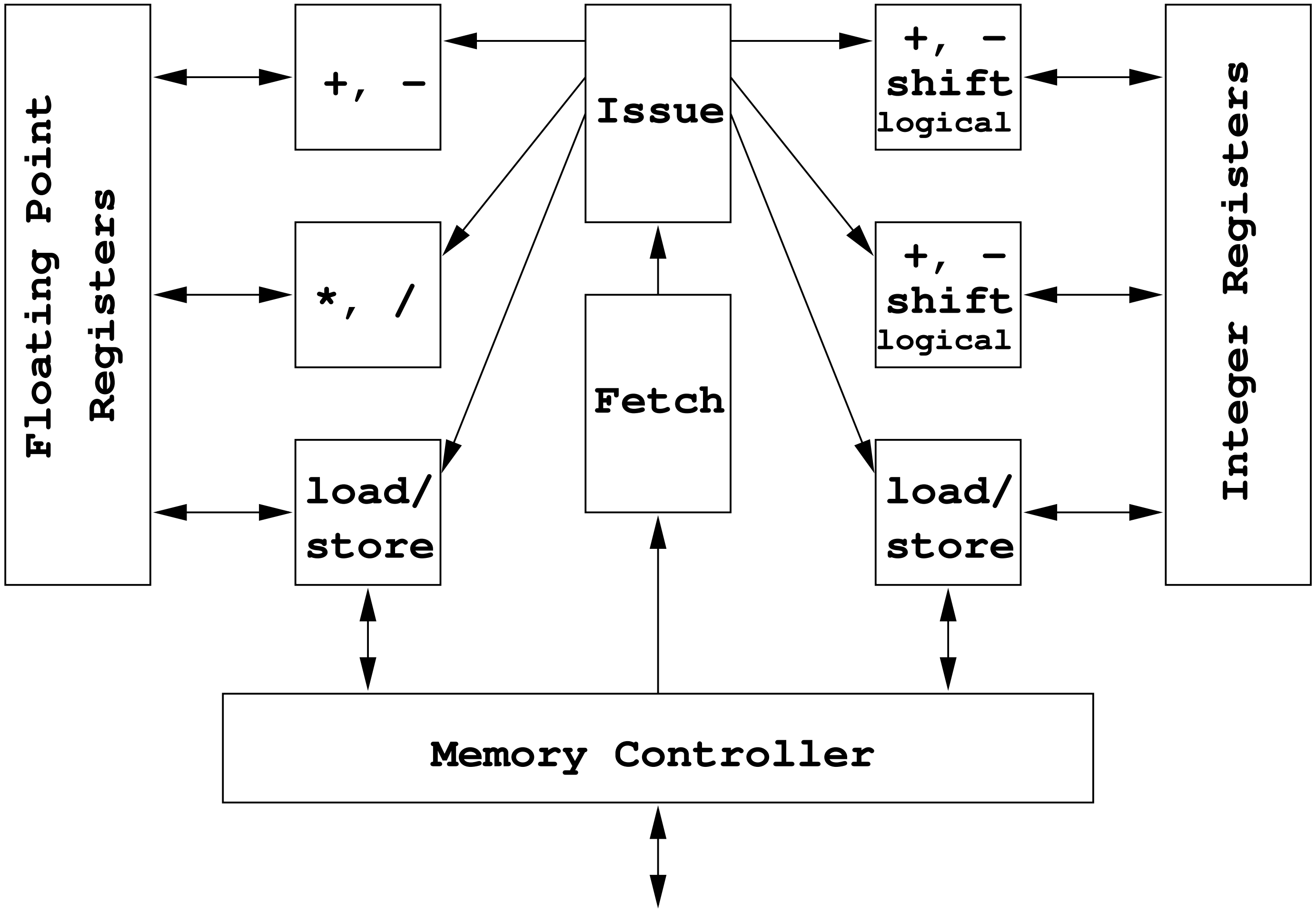
In many cases memory latency is a worse problem than memory bandwidth.

A cache miss on a modern computer typically takes around 100ns to resolve. In that time at least 100 instructions could be executed.

So one needs to issue loads over 100 instructions before the data are required by an operation, and one needs to be able to cope with a few dozen in-process memory requests.

Typical compilers issue loads around 5 instructions before the data are needed, and typical CPUs can cope with just a couple of simultaneous memory requests.

Schematic of Typical RISC CPU



An example

```
d=0.0

do i=1, n
  d=d+a(i)*b(i)
enddo
```

Ignore caches, assume 2 outstanding memory requests allowed, that memory latency is 100 cycles, and that add and multiply have a three cycle latency.

Time

0	start load a(1)
1	start load b(1)
. . .	
100	receive a(1), request a(2)
101	receive b(1), request b(2)
102	calc tmp=a(1)*b(1)
. . .	
105	calc d=d+tmp
. . .	
108	d calculated
. . .	
200	receive a(2), request a(3)

Time $= 100n$ cycles.

The vector advantage

Assume each processor register can hold a vector 64 scalar values.

0	start load of a(1)..a(64)
1	start load of b(1)..b(64)
...	
100	get a(1)
101	get a(2), b(1)
...	
164	get b(64)
165	calc tmp(1)=a(1)*b(1), req. a(65)..a(128)
166	calc tmp(2)=a(2)*b(2), req. b(65)..b(128)
...	
231	calc sum(tmp)
...	
c.300	d=d+sum(tmp)

Time $= 165 + \frac{135}{64} n$ cycles

The differences

One can pick plenty of holes in the detail of the above examples, but:

	RISC	Vector
Conc. mem. req.	1-5	100-500
FPs in regs.	32	500-8000
Mem. b/w	$<$1 byte/FLOP	$>$4 byte/FLOP

Note also that vector units have a significant start-up time. For short ($<$10) loops a vector unit is usually slower than a scalar one, and long ($>$100) loops are needed to get close to full speed.

Caches and Strides

In case anyone thinks that cache-line prefetching solves all RISC problems. . .

```
integer index(n)
real*8 a(n),dot

dot=0.0
do i=1,n
  dot=dot+a(index(i))**2
enddo
```

	PW500	Octane	SR2201	S3600
MHz	500	175	150	250
MFLOPS	1000	350	300	2000
stride 1	36	41	52	860
stride 5	12	14	52	730
stride 128	8	7	13	23
random	9	4	52	760

Cambridge's S3600/180

Total Memory	1GB
VPU Clock	250MHz
Peak GFLOPS	2
VPU Pipes	4 *,+
	4 +
	1 /
	4 load,store
	4 load
Vector regs	32 x (512 x 64 bit)
Mask regs	32 x (512 x 1 bit)
FP represent	IBM 360 (not IEEE!)
Addressing	31 bit
O/S	UNIX
Power	200V 3 phase 60kW
Cooling	Forced air
Weight	c. 10 tons

Pictures

Disks are c.70GB.

That console

Eight colours, 80x24 characters plus three status lines, 69 displayable symbols.

No:

- lower case letters
- escape key
- control key
- terminfo entry
- full screen editor

Lowercase letters displayed uppercase, uppercase displayed by preceding each with the Yen symbol. Entered in the same fashion. That an uppercase character is twice the width of a lowercase one messes up most formatting from ‘`ls`’ onwards.

Output paged, no automatic scrolling.

Extended Storage

With just 1 GB of memory and a >12GB/s bandwidth to it, it would be a pity have to use slow disks for swap space or smallish amounts of I/O.

Thus many vector computers provide some form of fast solid-state storage device for ‘small’ amounts of storage.

Turing had a 4 GB device using DRAMS with a theoretical transfer rate of 1GB/s. About 1.5 GB was used for swap space, the rest for such scratch files as users wished to place there.

B I G

The extended store uses 4 MBit DRAMS. Including ECC, that is over 9,000 chips for the 4 GB turing had. The S3600 supported up to 16GB of extended store.

The eight inch disk drives need no further comment.

The following is from one of Hitachi's manuals:

```
-- Ultrahigh-speed ECL Logic LSI

Almost all the logic of the S-3600 is imple-
mented in a number of ultrahigh-speed high-
density 2000 and 5000 gate LSIs with gate
delays of 200 ps and 250 ps, respectively.

-- LSI for ultrahigh-speed vector registers

An LSI chip for the vector registers contains
7K bits of high-speed RAM (with an access
time of 2.5 ns) and 2500 gates of logic for
accomplishing superhigh-speed vector
operations.
```

CISC

(AFAICT) Cray's assembler specifies with each vector instruction how many elements of the vector register to use, from 1 to 64 (Y/MP). For longer loops, one writes an explicit loop. If the loop count is guarenteed to be $<$64, no explicit loop is needed.

When moving to a Cray T90, the vector length is 128, and compiler directives flagging short loops ought to be revised. Moving the other way they must be revised.

The S3600 allows vector lengths of up to 2^{31} to be specified. Microcode takes care of forming the required loop if this is longer than the real registers. Thus one can move easily between the four different S3600 versions which have vector lengths varying from 128 to 512, as well as different numbers of pipelines.

A few unusual instructions exist: sum vector, inner product of two vectors, find max (min) value in vector.

The C@*? Language

C is a hopeless language for numerical HPC work. Turing running Linpack in FORTRAN achieves around 1 GFLOPS. In C, around 10 MFLOPS. This is because data dependencies are impossible to analyse decently in C. Consider:

```
vadd(double *a,double *b,double *c,int n)
{
   int i;
   for(i=0;i<n;i++) c[i]=a[i]+b[i];
}
```

Easy to vectorise?

NO!

```
vadd(double *a,double *b,double *c,int n)
{
   int i;
   for(i=0;i<n;i++) c[i]=a[i]+b[i];
}
```

```
  double fib[NMAX]

  fib[0]=fib[1]=1.0;
  vadd(&fib[0],&fib[1],&fib[2],NMAX-2);
```

The wonders of C's aliasing have let us set things so that the loop in `vadd` is

```
for(i=0;i<NMAX-2;i++)
   fib[i+2]=fib[i]+fib[i+1];
```

so one must wait until `c[n]` is stored before loading `b[n+1]` or `a[n+2]`.

The FORTRAN standard does not permit this madness (arguments must not overlap).

C again

C has many other problems too when it comes to optimising. The lack of a `for` loop of the simplicity of FORTRAN's is another major problem.

[In FORTRAN, provided a loop contains no `goto`, the iteration count can be determined on entry to the loop. The loop counter may not be modified in the loop, the exit condition is used at the beginning to calculate how many iterations will be done, and is not evaluated again. In C modifying the loop counter, modifying the variables used in the exit condition, and doing same via variables aliased with these, is permissible.]

Fortunately anyone who knows C can learn FORTRAN77 in half a day. The problems come when people program in C but do not actually understand the language. . .

The End of a Munition

Turing's appetite for electricity (over 100kW including air conditioning) made it rather expensive to run: several pounds an hour, or several tens of thousands of pounds a year, just for the electricity. Hitachi's maintenance contract was not free either. And then there was the valuable space which turing occupied.

So in 1999 turing was finally switched off, and shipped to a scrap yard authorised to receive and utterly destroy this dangerous munition.

Thus ended the life of probably the only S3600 seen outside Japan, and of a machine surrounded with bizarre restrictions concerning use by citizens of certain 'terrorist' countries and use in the design of nuclear weapons or code breaking.

The Future of Vectors

Vector computers will not run Word faster, nor even run Quake faster, so their future is not very bright.

However, a vector computer does not have to be built in custom logic surrounded by GBytes of SRAM: CMOS will do!

Hitachi's SR2201 (and SR8000) are examples of CMOS-based computers using 'standard' RISC processors (HP's PA-RISC and IBM's Power respectively) with enhancements to make them perform much more like vector processors. As Cambridge's SR2201 will be decommissioned this summer, CUCPS can start talking about it soon!

Vector Computers: the Dinosaurs of Computing?

Large, ruled the earth a long time ago, awkward house-mates, mostly extinct.

But still hanging on in small specialised corners into the 21st century. See the coelacanth, tuatara, horse-shoe crab and Wollemi pine for more details. . .