Big Iron

Today's topics:

Vector Processors and Supercomputers

VP's came first – now exist as GPGPU's

figure source: text Appendix F

Supercomputers

lots of microprocessors with a fancy interconnect – a look at the top500

Datacenter "cloud" Computing

lots of blades w/ fancy interconnect

AND fancy storage systems (this is not DRAM!)

Review

- Roadblocks to parallelism
  - wide issue & deep pipelines
    - » dynamic OOO issue
      - huge # of instructions on the fly
      - quadratic circuit complexity to keep track of everything
        - – forwarding, ROB size, # of registers
      - power density kills you
      - performance still limited by ILP in the program
    - » VLIW
      - compiler does most of the scheduling work
      - still huge # of instructions on the fly
      - power density is still a problem
        - – this will continue to be a common theme
      - performance also limited by ILP
- Enhancing parallelism
  - multi- threads, cores, sockets
    - » main game today
    - » might be easier to build than program

1st Supercomputers

- **Vector machines**
  - often attributed to Seymour Cray, but he says
    - "I'm certainly not inventing vector processors. There are three kind that I know of existing today. They are represented by the Illiac-IV, th (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors. . . . One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

      talk at LLNL – 1976 – on the introduction of the CRAY-1
- **Alternative programming model**
  - two data types
    - » scalar and vector
      - not wildly dissimilar to map reduce (Google reinvention)
        - – map sub-problems to some set of resources
        - – reduce/combine sub-problem into final answer
  - APL – Iverson's 1969 book
    - » +/(1, 2, 3) = 6

Replace Loops w/ Vector Instructions

- **Vector-Vector add**
  - conventional
    - » 2 pointers to head of two vectors
    - » offset with loop variable
      - A[i] + B[i] for all I
  - vector model
    - » Vadd A, B /1 instruction does a lot of work
    - » no loop or instruction decode overhead
    - » hazard checking only required between vector instructions
- **Issues**
  - each vector has to be contiguous
  - machine has a native vector length
    - » 64 was common
      - pad if actual vector length is not in chunks of 64
  - scientific programmers embraced the vector model
    - » but how do you write a web browser?

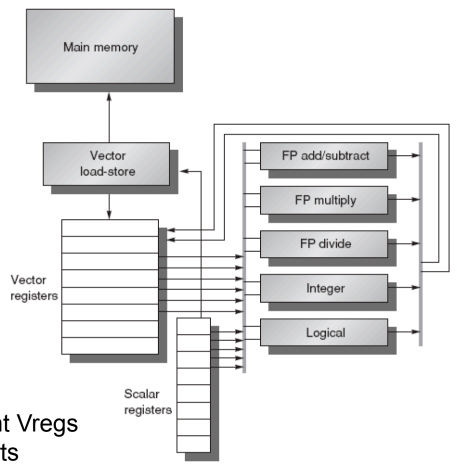
2001 Vector Odyssey

- Vector machines out of fashion
- 2002
  - Japan's Earth Simulator announced
    - » virtual planet
      - predict environmental change impact on world climate
    - » leads top500 list
      - widespread US panic @ government level
        - strategic leadership lost?
        - oh woe is us or U.S.
      - spurs supercomputer development
        - including new vector machines from Cray
- Now
  - wide-SIMD alive and well in GPGPU's
  - short-SIMD alive and well in CPU's
  - SIMD = short vector
    - » same issues apply

Basic Vector Architecture

- 2 parts
  - scalar unit
    - » similar to a normal CPU
      - OOO: NEC SX/5
      - VLIW: Fujitsu VPP5000
  - vector unit
    - » multiple FU's (both int & float)
      - deeply pipelined for high clock frequencies
      - particularly true for FPU's
        - primary focus for the scientific comp folks
- 2 basic architecture types
  - vector-register processors
    - » early CDC machnes
  - memory-memory vector processors (vector RISC)
    - » everything since about 1980
      - CRAY 1, 2, XMP, YMP, C90, T90, SV1, X1
      - NEC SX/2-SX/8, Fujitsu VP200-VPP5000, Hitachi S820 and S8300
      - Convex C-1 through C-4

Top Level Vector-Register VMIPS



64 element Vregs
2 read ports
1 write port
is it enough?

Snippet of Real Machines

Processor (year)	Vector clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983)	118	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray Y-MP (1988)	166					
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/ VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/ logical, shift	1	4
Cray C90 (1991)	240	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
Cray T90 (1995)	460					
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16

VMIPS ISA Snippet 1

Instruction	Operands	Function
ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.

VMIPS ISA Snippet 2

LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--V.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

DAXPY: MIPS vs. VMIPS

```
              L.D       F0,a           ;load scalar a
              DADDIU    R4,Rx,#512     ;last address to load
       Loop:  L.D       F2,0(Rx)       ;load X(i)
              MUL.D     F2,F2,F0       ;a × X(i)
              L.D       F4,0(Ry)       ;load Y(i)
              ADD.D     F4,F4,F2       ;a × X(i) + Y(i)
              S.D       0(Ry),F4       ;store into Y(i)
IC = 6 vs 600 DADDIU    Rx,Rx,#8       ;increment index to X
              DADDIU    Ry,Ry,#8       ;increment index to Y
              DSUBU     R20,R4,Rx      ;compute bound
              BNEZ      R20,Loop       ;check if done
```

Here is the VMIPS code for DAXPY.

```
              L.D       F0,a           ;load scalar a
              LV        V1,Rx          ;load vector X
              MULVS.D   V2,V1,F0       ;vector-scalar multiply
              LV        V3,Ry          ;load vector Y
              ADDV.D    V4,V2,V3       ;add
              SV        Ry,V4          ;store the result
```

Performance

- **Vector execution time**
  - **f(vector length, structural hazards, data hazards)**
    - » **initiation rate: # of operands consumed or produced per cycle**
    - » **multi-lane architecture**
      - **each vector lane can carry n values per cycle**
        - **often 2 or more**
      - **# vector lanes * lane width = initiation rate**
  - **also dependent on pipeline fill and spill**
- **Convoys (made up term)**
  - **set of independent vector instructions**
    - » **similar to an EPIC VLIW bundle**
- **Chime**
  - **time it takes to execute 1 convoy**
- **Start up time**
  - **time it takes to load the vector registers and fill the pipe**
- **All contribute to execution time**

Vector Memory Systems

- Lots of bandwidth required to feed lots of XU's
  - very wide data bus
  - banked memory
    - » each bank indpendently addressed
      - not interleaved
      - multiple load and stores issued per cycle
      - each bank serves a particular load or store
        - – assuming no bank conflict
        - – compiler tries hard to avoid conflict
      - latency can be high for DRAM based memory
        - – but bandwidth can be quite good
        - – early CRAY machines used SSRAM's – too expensive today
    - » addressing? where are the bank select bits?

Vector Length Control

- Vec.reg.length != operand.vec.size (OVS)
  - MVL = vec.reg.length
  - enter VLR
    - » specifies the operand vec size for a vector instruction
      - actual vector size often not known until run time
      - may even change based on a call parameter
      - APL rho(V)→ length (or structure if vector of vectors of …)
    - » controls XU's and Vector_Ld_Store Unit
    - » VLR value <= vector.reg.length
      - hence not known until run time
      - statically known then compiler can issue LVLRI – immediate
  - compiler generates "strip-mine" code based on OVS
    - » rem(OVS, MVL) = odd size piece
      - do odd size piece first or last
    - » OVS/MVL (truncating divide)
      - = number of loops or unrolls for MVL sized chunks

Vector Stride

- What happens when vectors are in contiguous addresses
  - consider MatMul
    - » same problem in column (FORTRASH) or row (C) major order allocation
  - stride = address distance between logically adjacent elements
  - solution = stride operand to vector load store unit
    - » on reads = gather
    - » on writes = scatter
  - ➔ scatter-gather load store unit
    - » a key innovation for vector processors
    - » LVWS instruction – load vector w/stride
    - » Note:
      - banking conflicts may occur
      - start up time increases
      - CDC Star (APL like machine language) classic example

Enhancing Vector Performance 1

- Chaining
  - originally developed in the CRAY-1
    - » now fairly ubiquitous
    - » same basic idea as forwarding
  - consider

```
MULV.D      V1,V2,V3
ADDV.D      V4,V1,V5
```

    - » RAW hazard – but if drive entire vector to forward destination
    - » essentially chains XU's together
  - problems?

Enhancing Vector Performance 1

- **Chaining**
  - **originally developed in the CRAY-1**
    - » **now fairly ubiquitous**
    - » **same basic idea as forwarding**
  - **consider**

```
MULV.D      V1,V2,V3
ADDV.D      V4,V1,V5
```

    - » **RAW hazard – but if drive entire vector to forward destination**
    - » **essentially chains XU's together**
  - **problems?**
    - » **the usual = exceptions**
      - **same barrier to performance**
      - **solution – chained or not modes**
      - **debug in not – do real run in chained mode**

Enhancing Vector Performance 2

- **Conditionals**
  - **also a performance barrier**
- **Solution similar to the normal CPU case**
  - **employ conditional instructions**
    - » **predication in EPIC**
    - » **but borrow mask idea from early SIMD machines**
      - **like ILLIAC IV**
    - » **execute the predicate on the vector**
      - **create a mask vector of the same length**
    - » **then do the real operation in masked fashion**

```
LV        V1,Ra        ;load vector A into V1
LV        V2,Rb        ;load vector B
L.D       F0,#0        ;load FP zero into F0
SNEVS.D   V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D    V1,V1,V2     ;subtract under vector mask
CVM                    ;set the vector mask to all 1s
SV        Ra,V1        ;store the result in A
```

Enhancing Vector Performance 3

- **Sparse matrix computation**
  - **matrices too big to store directly**
  - **hence use indirection**

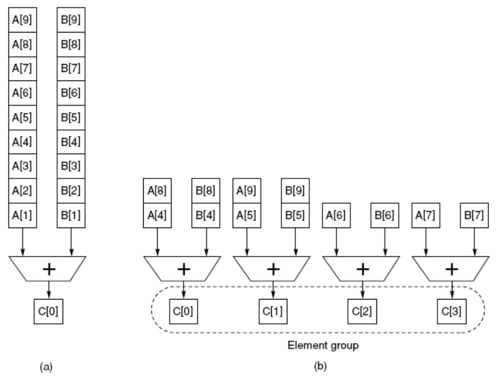
```
                do    100 i = 1,n
 100                  A(K(i)) = A(K(i)) + C(M(i))
```

    - » **scatter-gather memory now takes a bit longer**
      - **same idea however**
  - **or use a bit vector to indicate valid entries**
    - » **use as a mask**
  - **new instructions (set up time increases though)**
    - » **load and store vector indexed: LVI and SVI**

```
LV        Vk,Rk         ;load K
LVI       Va,(Ra+Vk)    ;load A(K(I))
LV        Vm,Rm         ;load M
LVI       Vc,(Rc+Vm)    ;load C(M(I))
ADDV.D    Va,Va,Vc      ;add them
SVI       (Ra+Vk),Va    ;store A(K(I))
```

Enhancing Vector Performance 4

- **Increase lanes**
  - **expensive but throw more XU's and datapath at the problem**


A[9] A[8] A[7] A[6] A[5] A[4] A[3] A[2] A[1]
B[9] B[8] B[7] B[6] B[5] B[4] B[3] B[2] B[1]
+
C[0]
(a)
A[8] A[4] B[8] B[4] A[9] A[5] B[9] B[5] A[6] B[6] A[7] B[7]
+ + + +
C[0] C[1] C[2] C[3]
Element group
(b)

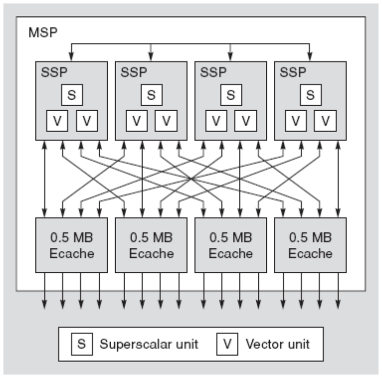
Vectorizing Compilers

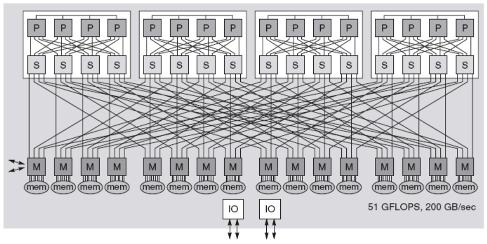
- **Take advantage of data-parallelism**
  - **not as easy as it seems but lots of success after years of effort (D. Kuck – UIUC, KAI → Portland group)**

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Cray X1 (2002)

- **Lots of processor cores with streaming caches**
  - **unified global memory**
  - **fully connected interconnect**





SSP – single streaming processor
MSP – multi-stream processor

Today's SuperComputers

- www.top500.org Nov 2009
  - Cray Jaguar XT5-HE (Oak Ridge NL, 1.7 Peta-Flops)
    - » 6 core Opteron CPU's – 2.6 GHz
    - » Black Widow interconnect (YARC switches)
  - IBM Roadrunner Blade Center QS22/L21 Cluster (LANL)
    - » PowerXCell 8i 3.2 GHz
    - » Opteron DC 1.8 GHz
    - » Voltaire infiniband interconnect
  - Kraken XT5 (Cray XT5-HE) – Univ. Tenn (NICS)
  - Jugene (IBM BlueGene/P) – research center Juelich, GER
  - Tianhe (Milky Way) – China Defense Univ.
  - SGI Pleiades
    - » Nehalem quad core 3 GHz
- Bottom line
  - high-end microprocessors, lots of memory, fancy interconnect
  - lots of watts and dollars

Cray Jaguar

source: www.pnl.gov

Cray Jaguar

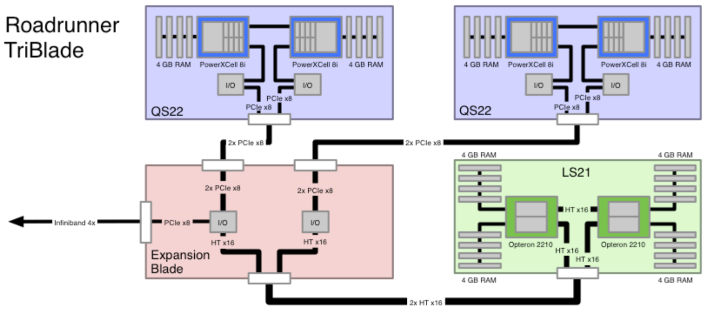
- **Comprising**
  - 45,000 quad-core Opteron CPU's (180K cores)
  - 362 TB of main memory
    - » 578 GB/sec main memory bandwidth
    - » 284 GB/sec I/O bandwidth
  - 10 PB of disk
  - Fat tree interconnect moving to 3D Torus
    - » radix 64 YARC swithes
- **Cost**
  - $100M – government contract
    - » no way to know if they made money
  - 7 MegaWatts of power
    - » plus another 7 MW to run the chillers
      - typical cooling energy = machine energy
    - » @ current Tennessee industrial rates 6.45 cents/kWh
      - $7,910,280 per annum power bill
  - your tax payer dollars at work

IBM RoadRunner

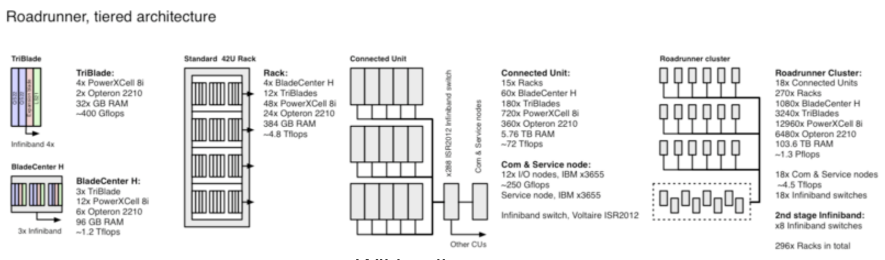
source: news.cnet.com

IBM RoadRunner

- **Comprising**
  - **13,824 cores**
    - » **6912 Opteron sockets**
      - • **6480 for compute, 432 for managment**
    - » **12,960 PowerXCell 8i sockets (2/Opteron)**
      - • **1 Power5 core and 8 SPE cores = 116,640 cores**
  - **Packaging in Tri-blades (source Wikipedia)**



RoadRunner in the Large



source: Wikipedia

- **CU – 60 BladeCenterH modules in a cabinent**
- **Other**
  - **103.6 TB Ram**
  - **216 x3755 I/O nodes –**
  - **26 radix 288 ISR2012 Infiniband 4x DDR switches**
  - **296 racks**
  - **2.35 MW operating power**

Concluding Remarks

- For supercomputers – what matters most?
  - blade configuration
  - rack configuration
  - interconnect
    - » on the blade
    - » in the rack
    - » between racks
  - how memory is partitioned
    - » remote vs. local access latencies and bandwidths
  - memory capacity and organization
  - the cores
  - power and performance on big benchmarks
- If you choose one for your final HW
  - there's a lot of advertizing copy
  - try to dig past that