Advanced Topics in Computer Architecture

Lecture 7
Data Level Parallelism: Vector Processors

Marenglen Biba
Department of Computer Science
University of New York Tirana

Cray

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (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.

Seymour Cray

Public lecture at Lawrence Livermore Laboratories on the introduction of the Cray-1 (1976)

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors F-34
- A Modern Vector Supercomputer: The Cray X1 F-40

SIMD: Single instruction stream, multiple data streams

- The same instruction is executed by multiple processors using different data streams.
- SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel.
- Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions.
- For applications that display significant data-level parallelism, the SIMD approach can be very efficient.
- Vector architectures, are the largest class of SIMD architectures.
- SIMD approaches have experienced a rebirth in the last few years with the growing importance of graphics performance, especially for the game market.
- SIMD approaches are the favored method for achieving the high performance needed to create realistic threedimensional, real-time virtual environments.

Why Vector Processors?

- We have seen how we could significantly increase the performance of a processor by issuing multiple instructions per clock cycle and by more deeply pipelining the execution units to allow greater exploitation of instruction-level parallelism.
- Unfortunately, we also saw that there are serious difficulties in exploiting ever larger degrees of ILP.
- The rapid increase in circuit complexity makes it difficult to build machines that can control large numbers of in-flight instructions, and hence limits practical issue widths and pipeline depths.

Vector processors

- *Vector processors* were successfully commercialized long before instruction-level parallel machines and take an alternative approach to controlling multiple functional units with deep pipelines.
- Vector processors provide high-level operations that work on *vectors* — linear arrays of numbers.
- A typical vector operation might add two 64-element, floating-point vectors to obtain a single 64-element vector result.
- The vector instruction is equivalent to an entire loop, with each iteration computing one of the 64 elements of the result, updating the indices, and branching back to the beginning.

Vector instructions

- Vector instructions have several important properties that solve many problems:
- A single vector instruction specifies a great deal of work — it is equivalent to executing an entire loop.
  - Each instruction represents tens or hundreds of operations, and so the instruction fetch and decode bandwidth needed to keep multiple deeply pipelined functional units busy is dramatically reduced.

Vector instructions

- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector and so hardware does not have to check for data hazards within a vector instruction.
  - The elements in the vector can be computed using an array of parallel functional units, or a single very deeply pipelined functional unit, or any intermediate configuration of parallel and pipelined functional units.

Vector instructions

- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors.
- That means the dependency checking logic required between two vector instructions is approximately the same as that required between two scalar instructions, but now many more elemental operations can be in flight for the same complexity of control logic.

Vector instructions

- Vector instructions that access memory have a known access pattern.
  - If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.

Vector Processors

- For the previously stated reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can use them frequently.
- Vector processors are particularly useful for large scientific and engineering applications, including car crash simulations and weather forecasting, for which a typical job might take dozens of hours of supercomputer time running over multi-gigabyte data sets.

Success of Vector Processors

- In 2001, exotic vector supercomputers appeared to be slowly fading from the supercomputing arena, to be replaced by systems built from large numbers of superscalar microprocessors.
- But in 2002, Japan unveiled the world's fastest supercomputer, the Earth Simulator, designed to create a "virtual planet" to analyze and predict the effect of environmental changes on the world's climate.
- The Earth Simulator was five times faster than the previous leader, and faster than the next 12 fastest machines combined.
- The Earth Simulator has fewer processors than competing microprocessor-based machines, but each node is a single-chip vector microprocessor with much greater efficiency

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors F-34
- A Modern Vector Supercomputer: The Cray X1 F-40

Types of vector architectures

- There are two primary types of architectures for vector processors: *vector-register processors* and *memory-memory vector processors*.
- In a vector-register processor, all vector operations—except load and store—are among the vector registers.
- These architectures are the vector counterpart of a load-store architecture.
- All major vector computers shipped since the late 1980s use a vector-register architecture, including the Cray Research processors (Cray-1, Cray-2, X-MP, YMP, C90, T90, SV1, and X1), the Japanese supercomputers (NEC SX/2 through
- SX/8, Fujitsu VP200 through VPP5000, and the Hitachi S820 and S-8300), and the minisupercomputers (Convex C-1 through C-4).

Types of vector architectures

- In a memory-memory vector processor, all vector operations are memory to memory. The first vector computers were of this type, as were CDC's vector computers.
- Here we will focus on vector-register architectures only.

Components of the architecture

- *Vector registers* — Each vector register is a fixed-length bank holding a single vector. VMIPS has eight vector registers, and each vector register holds 64 elements.
- *Vector functional units* — Each unit is fully pipelined and can start a new operation on every clock cycle.
- *Vector load-store unit* — This is a vector memory unit that loads or stores a vector to or from memory. The VMIPS vector loads and stores are fully pipelined,
- *A set of scalar registers* — Scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load-store unit.
  - These are the normal 32 general-purpose registers and 32 floating-point registers of MIPS.

VMIPS

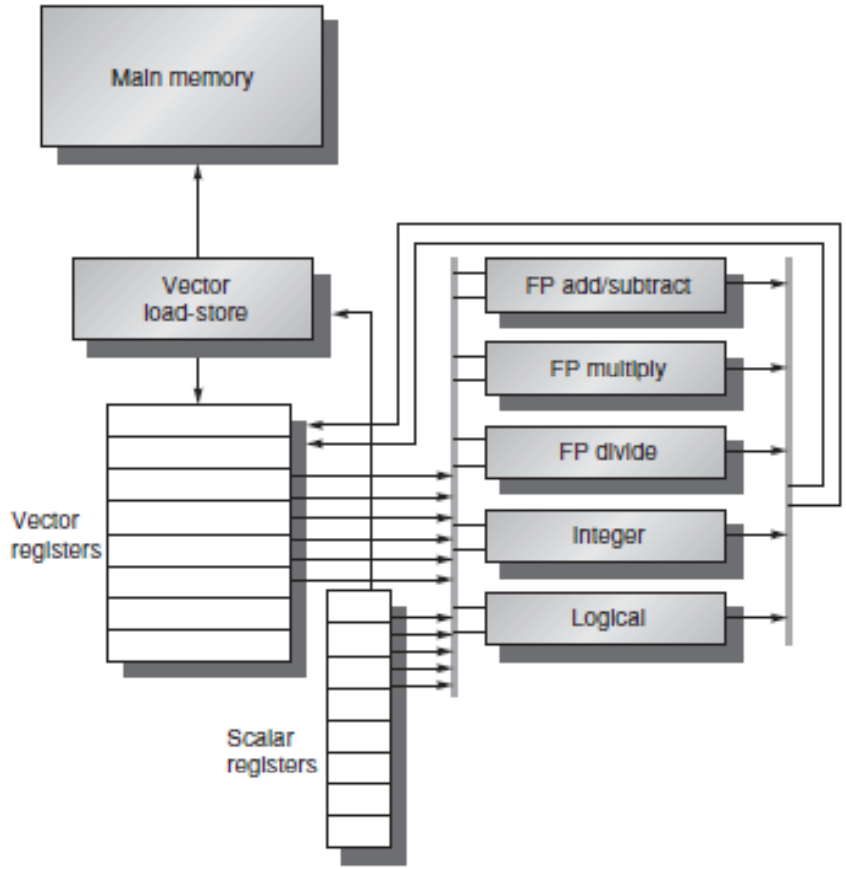

Figure F.1 The basic structure of a vector-register architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. Special vector instructions are defined both for arithmetic and for memory accesses. We show vector units for logical and integer operations. These are included so that VMIPS looks like a standard vector processor, which usually includes these units. However, we will not be discussing these units except in the exercises. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. These ports are connected to the inputs and outputs of the vector functional units by a set of crossbars (shown in thick gray lines). In Section F.4 we add chaining, which will require additional interconnect capability.

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
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998)	300	8	64 (MSP)	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
SV1ex (2001)	500					
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/shift, logical	1 load-store	1
NEC SX/6 (2001)	500	8 + 64	256	4: FP or integer add/shift, multiply, divide, logical	1	8
NEC SX/8 (2004)	2000	8 + 64	256	4: FP or integer add/shift, multiply, divide, logical	1	4
Cray X1 (2002)	800	32	64 256 (MSP)	3: FP or integer, add/logical, multiply/shift, divide/square root/logical	1 load 1 store	2 8 (MSP)
Cray XIE (2005)	1130					

VMIPS Vector Instructions

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.
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	VLR,R1	Move contents of R1 to the vector-length register.
MFC1	R1,VLR	Move the contents of the vector-length register to R1.
MVTM	VM,F0	Move contents of F0 to the vector-mask register.
MVFM	F0,VM	Move contents of vector-mask register to F0.

Figure F.3 The VMIPS vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in Section F.3) and VM (discussed in Section F.4). These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride are explained in Section F.3, and the uses of the index creation and indexed load-store operations are explained in Section F.4.

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors F-34
- A Modern Vector Supercomputer: The Cray X1 F-40

Vector-Length Control

- A vector-register processor has a natural vector length determined by the number of elements in each vector register.
- This length, which is 64 for VMIPS, is unlikely to match the real vector length in a program.
- Moreover, in a real program the length of a particular vector operation is often unknown at compile time. In fact, a single piece of code may require different vector lengths.
- The solution to these problems is to create a *vector-length register* (VLR). The VLR controls the length of any vector operation, including a vector load or store.
- The value in the VLR, however, cannot be greater than the length of the vector registers. This solves our problem as long as the real length is less than or equal to the *maximum vector length* (MVL) defined by the processor.

Strip Mining

- What if the value of *n* is not known at compile time, and thus may be greater than MVL?
- To tackle this problem where the vector is longer than the maximum length, a technique called *strip mining* is used.
- Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the MVL.
- We could strip-mine the loop in the same manner that we unrolled loops: create one loop that handles any number of iterations that is a multiple of MVL and another loop that handles any remaining iterations, which must be less than MVL.

Vector Stride

- The second problem this section addresses is that the position in memory of adjacent elements in a vector may not be sequential.
- This distance separating elements that are to be gathered into a single register is called the *stride*.
- Once a vector is loaded into a vector register it acts as if it had logically adjacent elements.
  - Thus a vector-register processor can handle strides greater than one, called *nonunit strides*, using only vector-load and vector-store operations with stride capability.
  - This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector processor over a cache-based processor.

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors F-34
- A Modern Vector Supercomputer: The Cray X1 F-40

Chaining: the Concept of Forwarding Extended to Vector Registers

- Consider the simple vector sequence

  MULV.D V1,V2,V3

  ADDV.D V4,V1,V5

- In VMIPS, as it currently stands, these two instructions must be put into two separate convoys, since the instructions are dependent.
- On the other hand, if the vector register, V1 in this case, is treated not as a single entity but as a group of individual registers, then the ideas of forwarding can be conceptually extended to work on individual elements of a vector.
- This insight, which will allow the ADDV.D to start earlier in this example, is called *chaining*.
- Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available: The results from the first functional unit in the chain are "forwarded" to the second functional unit.

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors F-34
- A Modern Vector Supercomputer: The Cray X1 F-40

Effectiveness of Compiler Vectorization

- Two factors affect the success with which a program can be run in vector mode.
- The first factor is the structure of the program itself:
  - Do the loops have true data dependences, or can they be restructured so as not to have such dependences?
  - This factor is influenced by the algorithms chosen and, to some extent, by how they are coded.
- The second factor is the capability of the compiler. While no compiler can vectorize a loop where no parallelism among the loop iterations exists, there is tremendous variation in the ability of compilers to determine whether a loop can be vectorized.

Level of Vectorization

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

Figure F.14 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991]. The first column shows the vectorization level obtained with the compiler, while the second column shows the results after the codes have been hand-optimized by a team of Cray Research programmers. Speedup numbers are not available for FLO52 and DYFESM, as the hand-optimized runs used larger data sets than the compiler-optimized runs.

Applying Vectorizing Compilers

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure F.15 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors
- A Modern Vector Supercomputer: The Cray X1 F-40

Performance of Vector Processors

- The simplest and best way to report the performance of a vector processor on a loop is to give the execution time of the vector loop.
- For vector loops people often give the MFLOPS (millions of floating-point operations per second) rating rather than execution time.
- We use the notation R_n for the MFLOPS rating on a vector of length n.
- Using the measurements T_n (time) or R_n (rate) is equivalent if the number of FLOPS is agreed upon.

Length Related Measures

Three of the most important length-related measures are

- R_∞—The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems do not have unlimited vector lengths, and the overhead penalties encountered in real problems will be larger.
- $N_{1/2}$ — The vector length needed to reach one-half of R_∞. This is a good measure of the impact of overhead.
- N_v — The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

Example

- What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

- Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS.
- We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 500$$

Example

- Simplifying this and then assuming $N_{1/2} < 64$, so that $T_{N1/2} < 64 = 64 + 3 \times n$, yields:

$$
\begin{aligned}
T_{N_{1/2}} &= 8 \times N_{1/2} \\
64 + 3 \times N_{1/2} &= 8 \times N_{1/2} \\
5 \times N_{1/2} &= 64 \\
N_{1/2} &= 12.8
\end{aligned}
$$

- So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.

Outline

- Why Vector Processors?
- Basic Vector Architecture
- Two Real-World Issues: Vector Length and Stride
- Enhancing Vector Performance
- Effectiveness of Compiler Vectorization
- Putting It All Together: Performance of Vector Processors
- A Modern Vector Supercomputer: The Cray X1

Cray X1

- The Cray X1 was introduced in 2002, and, together with the NEC SX/8, represents the state of the art in modern vector supercomputers.
- The X1 system architecture supports thousands of powerful vector processors sharing a single global memory.
- The Cray X1 has an unusual processor architecture,
  - A large Multi-Streaming Processor (MSP) is formed by ganging together four Single-Streaming Processors (SSPs).
  - Each SSP is a complete single-chip vector microprocessor, containing a scalar unit, scalar caches, and a two-lane vector unit.

Cray X1

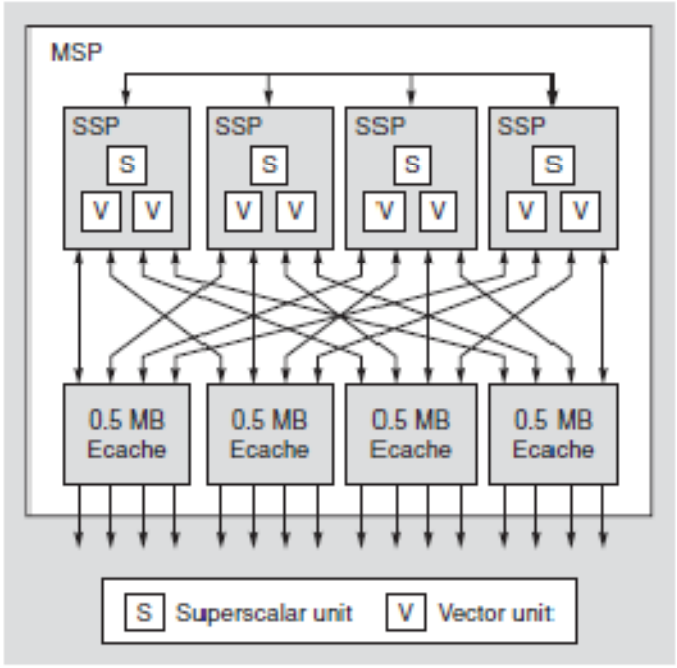

Figure F.17 Cray MSP module. (From Dunnigan et al. [2005].)

Each lane can perform a 64-bit floating-point add and a 64-bit floating-point multiply each cycle, leading to a peak performance of 12.8 GFLOPS per MSP.

Multi-Streaming Processors

- The Multi-Streaming concept was first introduced by Cray in the SV1, but has been considerably enhanced in the X1.
- The four SSPs within an MSP share Ecache, and there is hardware support for barrier synchronization across the four SSPs within an MSP.
- Each X1 SSP has a two-lane vector unit with 32 vector registers each holding 64 elements. The compiler has several choices as to how to use the SSPs within an MSP.

Cray X1E

- In 2004, Cray announced an upgrade to the original Cray X1 design.
- The X1E uses newer fabrication technology that allows two SSPs to be placed on a single chip, making the X1E the first multicore vector microprocessor.
- Each physical node now contains eight MSPs, but these are organized as two logical nodes of four MSPs each to retain the same programming model as the X1.
- In addition, the clock rates were raised from 400 MHz scalar and 800 MHz vector to 565 MHz scalar and 1130 MHz vector, giving an improved peak performance of 18 GFLOPS.

End of Lecture

- Readings
  - Book: Appendix F.