**Vector Processors**

A vector processor is a processor that can operate on entire vectors with one instruction, i.e. the operands of some instructions specify complete vectors. For example, consider the following add instruction:

C = A + B

In both scalar and vector machines this means ``add the contents of A to the contents of B and put the sum in C.'' In a scalar machine the operands are numbers, but in vector processors the operands are vectors and the instruction directs the machine to compute the pairwise sum of each pair of vector elements. A processor register, usually called the vector length register, tells the processor how many individual additions to perform when it adds the vectors.

A vectorizing compiler is a compiler that will try to recognize when loops can be transformed into single vector instructions. For example, the following loop can be executed by a single instruction on a vector processor:

 DO 10 I=1,N

 A(I) = B(I) + C(I)

10 CONTINUE

This code would be translated into an instruction that would set the vector length to N followed by a vector add instruction.

The use of vector instructions pays off in two different ways. First, the machine has to fetch and decode far fewer instructions, so the control unit overhead is greatly reduced and the memory bandwidth necessary to perform this sequence of operations is reduced a corresponding amount. The second payoff, equally important, is that the instruction provides the processor with a regular source of data. When the vector instruction is initiated, the machine knows it will have to fetch  pairs of operands which are arranged in a regular pattern in memory. Thus the processor can tell the memory system to start sending those pairs. With an interleaved memory, the pairs will arrive at a rate of one per cycle, at which point they can be routed directly to a pipelined data unit for processing. Without an interleaved memory or some other way of providing operands at a high rate the advantages of processing an entire vector with a single instruction would be greatly reduced.

A key division of vector processors arises from the way the instructions access their operands. In the *memory to memory* organization the operands are fetched from memory and routed directly to the functional unit. Results are streamed back out to memory as the operation proceeds. In the *register to register* organization operands are first loaded into a set of *vector registers*, each of which can hold a segment of a register, for example 64 elements. The vector operation then proceeds by fetching the operands from the vector registers and returning the results to a vector register.

The advantage of memory to memory machines is the ability to process very long vectors, whereas register to register machines must break long vectors into fixed length segments. Unfortunately, this flexibility is offset by a relatively large overhead known as the *startup time*, which is the time between the initialization of the instruction and the time the first result emerges from the pipeline. The long startup time on a memory to memory machine is a function of memory latency, which is longer than the time it takes to access a value in an internal register. Once the pipeline is full, however, a result is produced every cycle or perhaps every other cycle. Thus a performance model for a vector processor is of the form



where  is the startup time,  is the length of the vector and  is an instruction dependent constant, usually , 1 or 2.

Examples of this type of architecture include the Texas Instruments Inc. Advanced Scientific Computer and a family of machines built by Control Data Corp. known first as the Cyber 200 series and later the ETA-10 when Control Data Corp. founded a separate company known as ETA Systems Inc. These machines appeared in the mid 1970s after a long development cycle that left them with dated technology and disappeared in the mid 1980s. For a thorough discussion of their characteristics, see Hockney and Jesshope. One of the major reasons for their demise was the large startup time, which was on the order of 100 processor cycles. This meant that short vector operations were very inefficient, and even for vectors of length 100 the machines were delivering only about half their maximum performance. In a later section we will see how this vector length that yields half of peak performance is used to characterize vector computers.

In the register to register machines the vectors have a relatively short length, 64 in the case of the Cray family, but the startup time is far less than on the memory to memory machines. Thus these machines are much more efficient for operations involving short vectors, but for long vector operations the vector registers must loaded with each segment before the operation can continue. Register to register machines now dominate the vector computer market, with a number of offerings from Cray Research Inc., including the Y-MP and the C-90. The approach is also the basis for machines from Fujitsu, Hitachi and NEC. Clock cycles on modern vector processors range from 2.5ns (NEC SX-3) to 4.2ns (Cray C90), and single processor performance on LINPACK benchmarks is in the range of 1000 to 2000 MFLOPS (1 to 2 GFLOPS).

The basic processor architecture of the Cray supercomputers has changed little since the Cray-1 was introduced in 1976. There are 8 vector registers, named V0 through V7, which each hold 64 64-bit words. There are also 8 scalar registers, which hold single 64-bit words, and 8 address registers (for pointers) that have 20-bit words. Instead of a cache, these machines have a set of backup registers for the scalar and address registers; transfer to and from the backup registers is done under program control, rather than by lower level hardware using dynamic memory referencing patterns.

The original Cray-1 had 12 pipelined data processing units; newer Cray systems have 14. There are separate pipelines for addition, multiplication, computing reciprocals (to divide  by , a Cray computes ), and logical operations. The cycle time of the data processing pipelines is carefully matched to the memory cycle times. The memory system delivers one value per clock cycle through the use of 4-way interleaved memory.

An interesting feature introduced in the Cray computers is the notion of *vector chaining*. Consider the following two vector instructions:

 V2 = V0 \* V1

 V4 = V2 + V3

The output of the first instruction is one of the operands of the second instruction. Recall that since these are vector instructions, the first instruction will route up to 64 pairs of numbers to a pipelined multiplier. About midway through the execution of this instruction, the machine will be in an interesting state: the first few elements ofV2 will contain recently computed products; the products that will eventually go into the next elements of V2 are still in the multiplier pipeline; and the remainder of the operands are still in V0 and V1, waiting to be fetched and routed to the pipeline. This situation is shown in Figure 16, where the operands from V0 and V1 that are currently in the multiplier pipeline are indicated by gray cells. At this point, the system is fetching V0[k] and V1[k] to route them to the first stage of the pipeline andV2[j] is just leaving the pipeline. Vector chaining relies on the path marked with an asterisk. While V2[j] is being stored in the vector register, it is also routed directly to the pipelined adder, where it is matched with V3[j]. As the figure shows, the second instruction can begin even before the first finished, and while both are executing the machine is producing two results per cycle (V4[i] and V2[j]) instead of just one.

Without vector chaining, the peak performance of the Cray-1 would have been 80 MFLOPS (one full pipeline producing a result every 12.5ns, or 80,000,000 results per second). With three pipelines chained together, there is a very short burst of time where all three are producing results, for a theoretical peak performance of 240 MFLOPS. In principle vector chaining could be implemented in a memory-to-memory vector processor, but it would require much higher memory bandwidth to do so. Without chaining, three ``channels'' must be used to fetch two input operand streams and store one result stream; with chaining, five channels would be needed for three inputs and two outputs. Thus the ability to chain operations together to double performance gave register- to-register designs another competitive edge over memory-to- memory designs.