Arquitecturas de Alto Desempenho

Data-Level Parallelism

António Rui Borges

Summary

- *Types of data-level parallelism*
- *Vector architectures*
- *Vector operations*
- *Improving performance*
  - *Multiple lanes: behind one element per clock cycle*
  - *Vector length registers: handling loops not equal to the register size*
  - *Vector mask registers: handling conditional statements in vector loops*
  - *Memory banks: supporting bandwidth for vector load / store units*
- *Widening the application area*
  - *Stride: handling multidimensional arrays*
  - *Gather-scatter: handling sparse matrices*
  - *Programming vector architectures*
- *SIMD instruction set extensions for multimedia*
  - *Programming multimedia SIMD architectures*
- *Suggested reading*

Types of data-level parallelism - 1

Data-level parallelism (DLP) arises when multiple data items are processed at the same time. Two different types of computer architectures can fulfill this aim: SIMD and MIMD. Since a single instruction can launch many data operations, SIMD is potentially more energy efficient than MIMD, where one instruction is fetched and executed per data operation. A further advantage of SIMD over MIMD is that the programmer may still keep thinking sequentially and, yet, get a parallel speed up by carrying out independent simultaneous data operations.

Types of data-level parallelism - 2

Three variations of SIMD will be discussed

- *vector architectures* – they represent essentially a pipelined execution of many data operations; they were traditionally targeted to high-end scientific applications where data are well-structured and the number of computations is very large; supercomputers of the past were built in this way
- *multimedia instruction set extensions* – they represent essentially a parallel execution of data operations and are found today in most instruction set architectures that support multimedia applications
- *graphic processing units* (GPUs) – they share many characteristics with vector architectures, but there is a key difference: typically, they act as coprocessors in computer systems which include a conventional processor and its associated memory in addition to the GPU and the graphic memory, giving rise to what is now called *heterogeneous computing*.

Vector architectures - 1

Vector architectures grab sets of data elements scattered about memory, transfer them into large, sequential register banks, operate on those register banks through matrix-oriented computations and disperse the results back into memory. Thus, a single instruction operates on data vectors, giving rise to dozens of register-register operations on independent data elements.

The large register banks act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth. Due to the fact that loads and stores are deep pipelined, the program minimizes memory latency to once per vector load and store, instead of once per element. Indeed, vectors programs strive to keep memory access continuous so that it can take place in parallel to the computations being carried out by the processor.

Vector architectures - 2

To enhance the discussion about vector processing, a processor loosely based on Cray-1 is described. The processor's instruction set architecture will be called VMIPS because its scalar portion is MIPS and its vector portion is the logical vector extension of MIPS.

The primary components of VMIPS instruction set architecture are

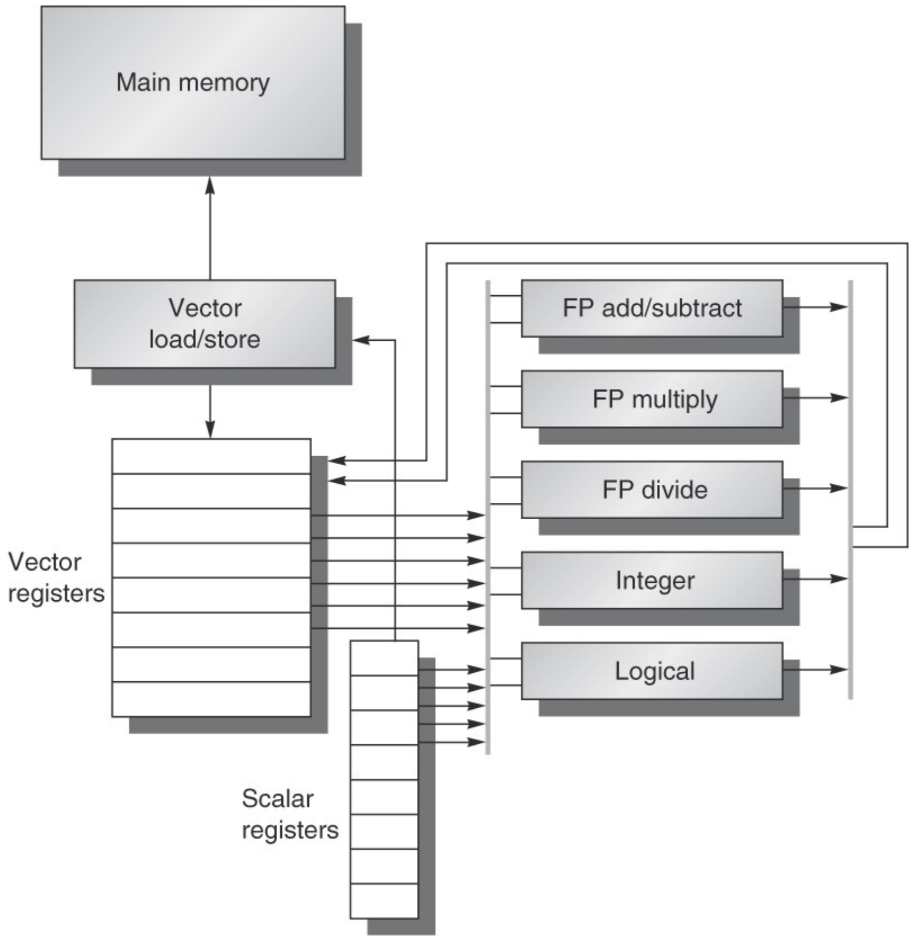
- *vector registers* – each vector register is in itself a fixed-length bank holding a single vector; VMIPS has eight vector registers, each holding 64 64-bit wide elements; the vector register bank provides enough input / output ports to feed all the vector functional units with a high degree of overlap; there are 16 read ports and 8 write ports which are connected to the functional units by a crossbar switch
- *vector functional units* – each unit is fully-pipelined and may start a new operation every clock cycle; one needs a control unit to detect hazards, both structural hazards for functional unit allocation and data hazards on register access

Vector architectures - 3

- *vector load / store unit* – the vector load / store unit loads from or stores to memory a data vector; VMIPS unit is fully-pipelined so that words are moved between the vector registers and memory with a bandwidth of one word per clock cycle, after the initial latency; this unit will also handle scalar loads and stores
- *scalar registers* – scalar registers can also provide input to the vector functional units; they are intended to hold the addresses to pass to the vector load / store unit as well; VMIPS has the usual 32 general purpose registers and the 32 floating point registers of MIPS, all 64-bit wide.

Vector architectures - 4

Basic organization of VMIPS architecture
Source: Computer Architecture: A Quantitative Approach



Vector architectures - 5

VMIPS vector instructions (only double precision floating point operations are shown)

instruction	***operands***	***comment***
`ADDVV.D`	`V1,V2,V3`	V1[k] = V2[k] + V3[k] (vector-vector addition)
`ADDVS.D`	`V1,V2,F3`	V1[k] = V2[k] + F3 (vector-scalar addition)
`SUBVV.D`	`V1,V2,V3`	V1[k] = V2[k] – V3[k] (vector-vector subtraction)
`SUBSV.D`	`V1,F2,V3`	V1[k] = F2 – V3[k] (scalar-vector subtraction)
`SUBVS.D`	`V1,V2,F3`	V1[k] = V2[k] - F3 (vector-scalar subtraction)
`MULVV.D`	`V1,V2,V3`	V1[k] = V2[k] x V3[k] (vector-vector multiplication)
`MULVS.D`	`V1,V2,F3`	V1[k] = V2[k] x F3 (vector-scalar multiplication)
`DIVVV.D`	`V1,V2,V3`	V1[k] = V2[k] / V3[k] (vector-vector division)
`DIVSV.D`	`V1,F2,V3`	V1[k] = F2 / V3[k] (scalar-vector division)
`DIVVS.D`	`V1,V2,F3`	V1[k] = V2[k] / F3 (vector-scalar division)
`LV`	`V1,R1`	V1[k] = mem[R1 + k] (vector elements are adjacent)
`SV`	`R1,V1`	mem[R1 + k] = V1[k] (vector elements are adjacent)
`LVWS`	`V1,(R1,R2)`	V1[k] = mem[R1 + k x R2] (vector elements are separated by the fixed value in R2)
`SVWS`	`(R1,R2),V1`	mem[R1 + k x R2] = V1[k] (vector elements are separated by the fixed value in R2)
`LVI`	`V1,(R1+V2)`	V1[k] = mem[R1 + V2[k]] (vector elements are random separated by values in V2)
`SVI`	`(R1+V2),V1`	mem[R1 + V2[k]] = V1[k] (vector elements are random separated by values in V2)
`CVI`	`V1,R1`	create the index vector V1[k] = k x R1
`S__VV.D`	`V1,V2`	compare the elements V1[k], V2[k] (__ = EQ, NE, GT, LT, GE, LE) VM[k] = 1 (condition true) - 0 (otherwise)
`S__VF.D`	`V1,F2`	compare the elements V1[k], F2 (__ = EQ, NE, GT, LT, GE, LE) VM[k] = 1 (condition true) - 0 (otherwise)
`POP`	`R1,VMR`	R1 = number of 1s in *vector mask register* VM
`CVM`		set *vector mask register* contents to all 1s
`MTC1`	`VLR,R1`	move contents of R1 to *vector length register* VL
`MFC1`	`R1,VLR`	move contents of *vector length register* VL to R1
`MVTM`	`VMR,F1`	move contents of F1 to *vector mask register* VM
`MVFM`	`F1,VMR`	move contents of *vector mask register* VM to F1

Vector architectures - 6

The power wall leads to value architectures that can deliver high performance without the energy consumption and the design complexity of out-of-order superscalar processors. Vector instructions are a natural match to this trend, since they can be used to increase the performance of simple in-order scalar processors without greatly increasing energy demands and design complexity. In practice, developers can express many of the programs that ran well in complex out-of-order designs more efficiently as data-level parallelism in the form of vector instructions.

With a vector expression, the system can perform the operations on the vector data elements in many different ways, including operating simultaneously on these elements. This flexibility lets vector designs use slow, but wide, execution units to achieve high performance at low power. Furthermore, the independence of elements within a vector instruction set allows scaling the functional units without carrying out additional dependency checks as superscalar require.

Vectors naturally accomodate varying data types: one view of a vector register size is 64 64-bit wide elements, but 128 32-bit wide elements, 256 16-bit wide elements, or even 512 8-bit wide elements, are equally valid views. Such a multiplicity makes a vector architecture useful for multimedia as well as scientific applications.

Vector operations - 1

Consider the following vector problem

$$Y = a \times X + Y$$

where `X` and `Y` are vectors, initially resident in memory, and `a` is a scalar.

This problem is called SAXPY or DAXPY, according to the operands are single or double precision, and forms the inner loop of the *Linpack benchmark*. *Linpack* is a library of linear algebra subroutines and the *Linpack benchmark* consists of a program for performing Gaussian elimination.

It is assumed for now that the vectors length is 64 (the length of VMIPS vector registers) and one wants to compare the code for the DAXPY loop written both in MIPS and VMIPS.

The initial addresses of the memory locations where vectors `X` and `Y` are stored, are assumed to be in registers `Rx` and `Ry`, respectively.

Vector operations - 2

MIPS code for DAXPY

```
        L.D     F0,a
        DADDIU  R4,Rx,512
Loop:   L.D     F2,0(Rx)
        MUL.D   F2,F2,F0
        L.D     F4,0(Ry)
        ADD.D   F4,F4,F2
        S.D     F4,0(Ry)
        DADDIU  Rx,Rx,8
        DADDIU  Ry,Ry,8
        DSUBU   R20,R4,Rx
        BNEZ    R20,Loop
```

VMIPS code for DAXPY

```
L.D      F0,a
LV       V1,Rx
MULVS.D  V2,V1,F0
LV       V3,Ry
ADDVV.D  V4,V2,V3
SV       Ry,V4
```

Vector operations - 3

The most striking difference between the two programs is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus 578 for MIPS. This reduction is due to the fact that the vector operations work directly on the 64 elements, as a whole, and that the overhead instructions, which represent nearly half the loop on MIPS, are not present here.

When the compiler generates vector instructions for a sequence and the resulting code spends much of its time running in vector mode, the code is said to be *vectorized* or *vectorizable*. Loops can be vectorized only if they do not have dependences between loop iterations, or *loop-carried dependences*, as they are also called.

Another important difference is the frequency of pipeline interlocks. In the MIPS code, they occur for every loop iteration, while for the vector processor, each vector instruction will only stall for the first element in each vector, subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *instruction*, rather than once per vector *element*. Vector architects call forwarding of element-dependent operations *chaining*, because the dependent operations are chained together.

Vector operations - 4

The execution time of a sequence of vector operations depends primarily on three factors

- the length of the operand vectors
- the number of structural hazards, and their kind, that the operations underlie
- the data dependences among successive operations.

Given the vector length and the initiation rate, which is the rate a specific vector unit consumes new operands and produces new results, the time for a single vector instruction can be computed. For simplicity, it will be assumed that VMIPS implementation has all functional units having a single pipeline, or *lane*, with an initiation rate of one element per clock cycle for individual operations. Thus, the execution time for a single vector operation is approximately the vector length.

Vector operations - 5

The notion of convoy will be also introduced to simplify the discussion of vector execution and vector performance. A *convoy* is to be understood as the set of vector instructions that can potentially be executed together. The instructions in a convoy *must not* contain any structural hazards. If such hazards were present, they would need to be serialized and initiated in different convoys. Again, to make analysis simple, it will be assumed that a convoy of instructions must complete execution before any other instructions, scalar or vector, may begin execution.

It might appear that besides vector instruction sequences with structural hazards, sequences with RAW hazards should also be in separate convoys, but one has to remember that chaining allows them to be executed together. Through its application, a vector operation may start as soon as the individual elements of their vector source operands become available: the results from the first functional unit in the chain are forwarded to the other functional units. In practice, this is implemented by allowing the processor to read and write a particular vector register at the same time, provided that different elements are accessed. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, provided that a structural hazard is not generated.

Vector operations - 6

Consider the following VMIPS code sequence

```
LV       V1,Rx
MULVS.D V2,V1,F0
LV       V3,Ry
ADDVV.D V4,V2,V3
SV       Ry,V4
```

A minimum of three convoys are needed to execute it.

The first convoy starts with the first `LV` instruction. The `MULVS.D` is dependent on it, but chaining allows it to be included in the same convoy. The second convoy starts with the second `LV` instruction, a structural hazard would be induced if placed in the first convoy (why?). The `ADDVV.D` is dependent on it, but it can again be placed in the same convoy via chaining. Finally, the `SV` instruction induces another structural hazard, if placed in the second convoy, so it must go in a third convoy.

Vector operations - 7

To turn convoys into execution time, a timing metric is needed to estimate the execution time of a convoy. This metric is called a *chime*. Therefore, a vector code sequence consisting of *m* convoys executes in *m* chimes: for vectors of size *n*, this would mean an execution time of approximately $m \cdot n$ clock cycles for VMIPS.

The chimes approximation ignore some processor specific overheads, many of which are independent on the vector length. Hence, measuring time in chimes is a better approximation for longer vectors than for short ones.

One ignored source of overhead in measuring chimes is the possible limitation on initiating multiple vector instructions in the same clock cycle. If only one vector instruction can be initiated per clock cycle, as it happens in most vector processors, the chime count will underestimate the actual execution time of a convoy.

The most important source of overhead ignored by the chime model is *vector start up time*. The start up time is chiefly determined by the pipelining latency of the vector functional units. The same pipeline depths, as the ones present in Cray-1, will be used for VMIPS although latencies in modern processors tend to increase, specially for vector load operations. The pipeline depths are 6 clock cycles for FP add, 7 for FP multiply, 20 for FP divide and 12 for vector load.

Improving performance

The following questions must have adequate answers if one aims to improve the performance of a vector processor

- how can a vector processor execute a single vector more than one element per clock cycle?
- how does a vector processor handle programs where the vector lengths are not the same as the length of the vector registers?
- what happens when there are conditional statements inside the code to be vectorized?
- what does a vector processor need from the memory system?

Multiple lanes: beyond one element per clock cycle - 1

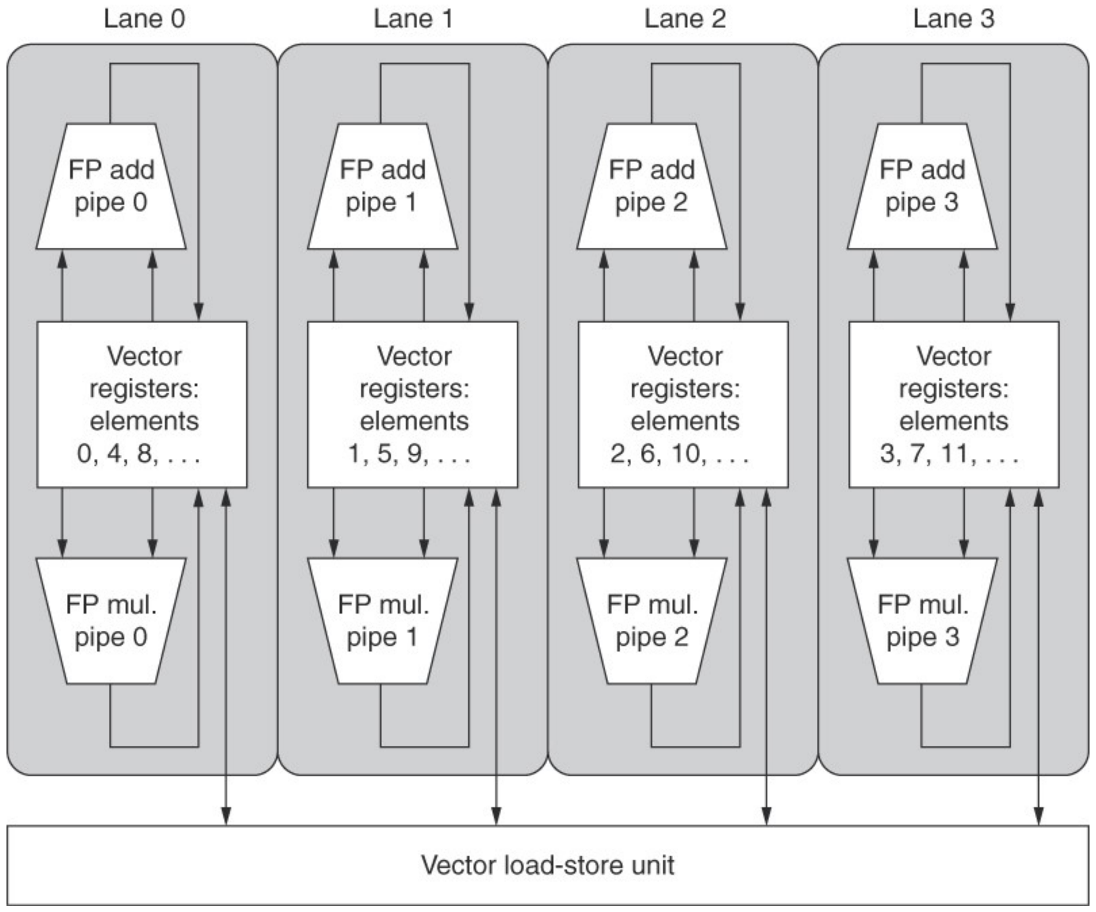
A critical advantage of a vector instruction set is that it allows the program to pass a large amount of parallel work to the hardware using only a single short instruction. This instruction includes scores of independent operations, yet encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allows an implementation to execute these elemental operations through either a deeply pipelined functional unit, as VMIPS, or an array of parallel functional units, or a combination of both.

The VMIPS instruction set has the property that all vector arithmetic instructions only allow element *k* of one vector register to take part in operations with element *k* of another vector register. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as having multiple parallel lanes.

Multiple lanes: beyond one element per clock cycle - 2

Structure of a vector unit organized in four lanes

Source: Computer Architecture: A Quantitative Approach


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Multiple lanes: beyond one element per clock cycle - 3

Going from one to four lanes reduces the number of clock cycles for a chime from 64 to 16. However, for a multilane implementation to be effective, both the applications and the architecture must support long vectors; otherwise the execution will be so fast that there is a risk of running out of instruction bandwidth and, thus, requiring ILP techniques to feed enough vector instructions.

Each lane contains a portion of the vector register bank and one execution pipeline from each vector functional unit. The vector functional units, thus, execute vector instructions at the rate of one *group of elements* per clock cycle. Avoiding interlane communication reduces both the wiring cost and the number of register bank ports needed to build a highly parallel execution unit.

Adding multiple lanes is a popular technique to improve vector performance, as it requires little increase in control complexity and does not imply changes to existing machine code. It also allows designers to trade off die area, clock rate and energy without sacrificing peak performance (if, for instance, the clock rate is halved, doubling the number of lanes will retain the same potential performance).

Vector length registers: handling loops not equal to the register size - 1

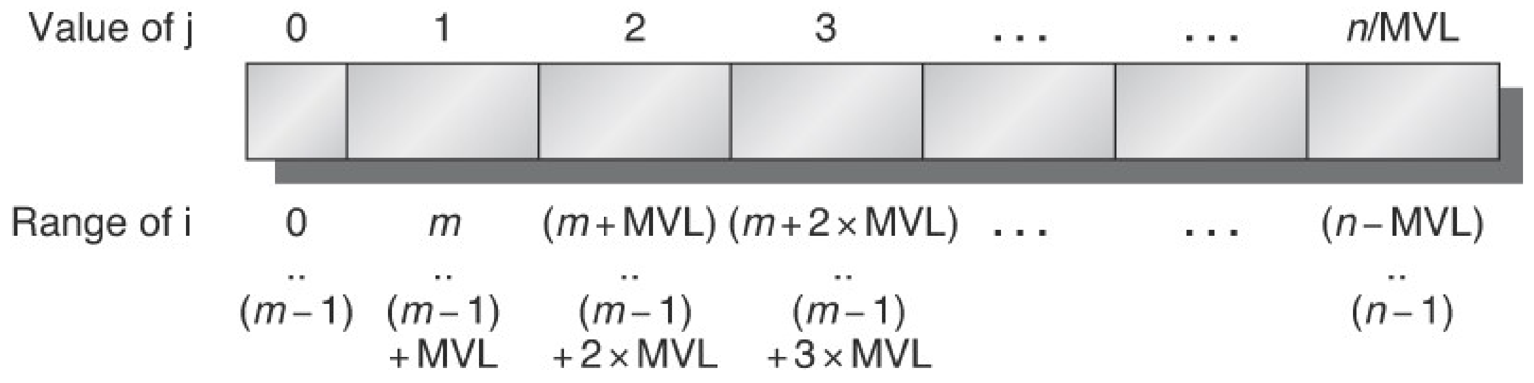
A vector processor has a natural vector length determined by the number of elements in each vector register. This length is unlikely to match the real vector length in a program. Moreover, the length of a particular vector operation in a real program is often unknown at compile time. In fact, a single piece of code may even require different vector lengths, as it happens if it is included in a procedure which has as parameter precisely the number of loop iterations.

The solution to these problems is to create a *vector length register* (VL). The VL controls the length of any vector operation, including loads and stores. The value stored in VL , however, can not be greater than the length of the vector registers, so solutions based on VL only work when the real vector length is less or equal to the *maximum vector length* (MVL), an architecture-based parameter that specifies the number of data elements in the vector registers for the current implementation.

Vector length registers: handling loops not equal to the register size - 2

Application of strip mining to a vector of arbitrary length

Source: Computer Architecture: A Quantitative Approach



When the real vector length is greater than the *maximum vector length* (MVL), a technique known as *strip mining* is applied.

```
b = 0;
VL = n % MVL;
for (j = 0; j <= n/MVL; j++)
{ for (i = b; i < b+VL; i++)     // vector operation
    Y[i] = a * X[i] + Y[i];
  b += VL;
  VL = MVL;
}
```

Vector mask registers: handling conditional statements in vector loops - 1

According to Amdahl's law, the speed up of programs with low to moderate levels of parallelization, in this case vectorization, is very limited. The presence of conditionals, like **if** statements, inside loops is a critical reason for low levels of vectorization because of the introduction of control dependences.

Consider the code

```
for (i = 0; i < VL; i++)
  if (X[i] != 0.0) X[i] = X[i] - Y[i];
```

The loop can not be vectorized in a straightforward manner due to the conditional execution of the body. However, if it can be worked out that the loop is run only for the iterations where `X[i]` $\neq$ `0`, then the subtraction operation could be vectorized.

Vector mask registers: handling conditional statements in vector loops - 2

The common extension for this capability is called *common mask control*. Mask registers provide conditional execution of each elemental operation in a vector operation. The *vector mask control* uses a boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a boolean condition to assert whether a scalar instruction should be executed.

When the *vector mask register* (VM) is enabled, any vector instructions operate only on the vector elements whose corresponding entries in VM are one. The vector elements whose corresponding entries in VM are zero, remain unaffected. Clearing VM, that is, setting all its bits to ones, makes subsequent vector instructions to operate on all vector elements.

Vector mask registers: handling conditional statements in vector loops - 3

Assuming that the start addresses of the memory locations where vectors `X` and `Y` are stored, are in registers `Rx` and `Ry`, the previous loop may be coded as

```
LV         V1,Rx
LV         V2,Ry
L.D        F0,0
SNEVS.D    V1,F0      // sets VM[i] to 1, if V1[i] ≠ 0
SUBVV.D    V1,V1,V2
SV         Rx,V1
```

Compiler writers call *if conversion* the transformation that changes an **if** statement into a straight line code sequence using conditional execution.

Conditional execution does have some overhead, exposed by the need to execute the `S__VS` .instruction. However, even with a significative number of zeros in VM, using *vector mask control* may still be faster than using the scalar mode.

Memory banks: supporting bandwidth for vector load / store units - 1

The behavior of a load / store vector unit is a lot more complicated than that of an arithmetic functional unit. The *start up time* for a load or store operation is the time required to get the first word from memory into a register or from a register into memory. If the remainder of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be one clock cycle because memory bank stalls can reduce the effective throughput.

Typically, penalties for start ups on load / store units are much higher than those for the arithmetic units, over 100 clock cycles on many processors. For VMIPS, a start up time of 12 clock cycles is assumed (the same as for Cray-1). Caching is being used by the most recent vector processors to bring down the latency time of vector loads and stores.

Memory banks: supporting bandwidth for vector load / store units - 2

To maintain an initiation rate of one word fetched or stored per clock cycle, the memory system must be capable of producing or consuming this amount of data. Spreading accesses across multiple independent memory banks usually delivers the desired rate.

This spreading, rather than simple memory interleaving, is relevant because

- most vector computer systems contain multiple vector processors that share the same memory system, meaning that each processor will be generating its own address stream
- the ability to load and store data words whose addresses are not sequential, a feature supported by most vector processors, requires independent bank addressing rather than interleaving
- simultaneous access by multiple load / store units to the memory system requires multiple banks and the capability of independent address control to them.

Memory banks: supporting bandwidth for vector load / store units - 3

The above mentioned reasons, taken together, lead to a large number of independent memory banks.

The largest configuration of Cray T90 (Cray T932) has 32 vector processors, each capable of generating 4 loads and 2 stores per clock cycle. The processors clock cycle is 2.167 ns, while the SRAMs cycle time used in the memory system is 15 ns.

Calculate the minimum number of memory banks required to allow all processors to run at full memory bandwidth.

The maximum number of memory references per clock cycle is equal to 192 (32 processors times 6 references per processor). On the other hand, each SRAM bank is busy for $15 / 2.167 \approx 7$ clock cycles. Hence, a minimum of $192 \times 7 = 1344$ memory banks are needed!

The Cray T932 actually has 1024 memory banks, not sustaining full bandwidth for all processors in simultaneous. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time and thereby providing sufficient bandwidth.

Widening the application area

The following questions must have adequate answers if one aims to widen the type of problems that can be solved efficiently by a vector processor

- how does a vector processor handle multidimensional matrices?
- how does a vector processor handle sparse matrices?
- how does one program a vector computer?

Stride: handling multidimensional arrays - 1

Successive elements of a vector may not be stored in adjacent memory locations.

Consider the code below for matrix multiplication of matrices of size N × N

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
  { A[i][j] = 0.0;
    for (k = 0; k < N; k++)
      A[i][j] += B[i][k] * D[k][j];
  }
```

To make it run efficiently on a vector processor, the multiplication of each row of `B` by each column of `D` can be vectorized and the inner loop strip-mined with *k* as the index variable.

To do so, a way must be found on how to address successive elements of `D`. One should note that when an array is stored in memory within a C environment, it is linearized and laid out in row-major order. This means that successive elements of `D`, accessed by the iterations in the inner loop, are separated by N. Thus, without reordering the loops, the compiler can not hide the distance between successive elements of `D`.

Stride: handling multidimensional arrays - 2

The distance separating elements to be gathered in a single register is called *stride*. Once a vector is loaded into a vector register, it acts as if its successive elements are adjacent, that is, a vector processor can handle strides greater than one provided that there are load and store instructions with stride capability. This ability of accessing non sequential memory locations and of reshaping their content into a dense structure is one of the major advantages of a vector processor.

The vector stride, as the vector starting address, can be loaded in a general purpose register and used in special load and store instructions. In VMIPS, the instruction `LVWS` (load vector with stride) fetches the vector elements, separated in memory by a stride greater than one, into a vector register. Likewise, the instruction `SVWS` (store vector with stride) does the same in the reverse direction.

Stride: handling multidimensional arrays - 3

Supporting strides greater than one complicates the memory system. Once non-unit strides are introduced, it becomes possible to request accesses from the same memory bank frequently. When multiple accesses contend for a particular bank, a memory bank conflict occurs, therefore, stalling all but one of the accesses.

A bank conflict and, hence, a stall will occur if

$$\frac{\text{number of banks}}{\text{g.c.d.(stride, number of banks)}} < \text{bank busy time} \;.$$

Suppose there are 8 memory banks with a bank busy time of 6 clock cycles and that the memory latency is 12 clock cycles. How long will it take to complete a 64-element vector load with a stride of 1 and with a stride of 32?

stride of 1: $12 + 64 = 76$ clock cycles or 1.2 clock cycles per element

stride of 32: $12 + 1 + 6 \times 63 = 391$ clock cycles or 6.1 clock cycles per element.

Notice that, in the second case, all the accesses are performed in the same memory bank.

Gather-scatter: handling sparse matrices - 1

Sparse matrices are [usually very large] matrices where most of its elements are zero. When operating with sparse matrices, it is important to have some means to describe in a compact form the location of its non zero elements so that operations can be concentrated on them and the computation be made run fast.

Assuming a simplified sparse structure, the code below computes the sum of two sparse matrices A and C, where the location of their corresponding non zero elements, *n* of them, is described by the index vector K

```
for (i = 0; i < N; i++)
  A[K[i]] = A[K[i]] + C[K[i]];
```

Gather-scatter: handling sparse matrices - 2

The primary mechanism in vector processors for supporting sparse matrices is the implementation of *gather-scatter* operations using index vectors. The goal of such operations is to enable moving back and forth between a compressed representation (zero elements are not included) and a normal representation (zero elements are included).

A *gather* operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets stored in the index vector. The outcome is a dense vector stored in a vector register. The elements are then operated in this dense form and the result is sent back to memory by the *scatter* operation using the same index vector.

Hardware support for these operations is present in nearly all modern vector processors. In VMIPS, for instance, the instructions `LVI` (load vector indexed or *gather*) and `SVI` (store vector indexed or *scatter*) play this role.

Gather-scatter: handling sparse matrices - 3

Assuming that the start addresses of the memory locations where matrices `A` and `C` are stored in normal representation, are in registers `Ra` and `Rc` and that register `Rk` contains the offsets to the corresponding non zero elements of these matrices, the previous loop may be coded as

```
LV          Vk,Rk
LVI         Va,(Ra+Vk)
LVI         Vc,(Rc+Vk)
ADDVV.D     Va,Va,Vc
SVI          (Ra+Vk),Va
```

This technique allows code with sparse matrices to run in vector mode. However, a programmer's directive is required to tell the compiler it is safe to code the loop in this way.

Although indexed loads and stores can be pipelined, they will run much slower than the corresponding non-indexed ones due to the bank conflicts that may occur throughout the memory system.

Programming vector architectures

An advantage of vector architectures is that compilers can inform programmers at compile time whether a specific code portion will vectorize or not, often providing hints as to why it did not vectorize it. This straightforward execution model allows experts in other domains to learn how to improve performance by revising the code or by advising the compiler when it is OK to assume independent loop operations, such as in the gather-scatter case for data transfers. It is precisely this interaction between the compiler and the programmer that simplifies programming of vector computers.

Nowadays, the main factor that affects the success with which a program runs in vector mode is the structure of the program itself. Do particular loops have true data dependences, or some restructuring is possible to get rid of such dependences, are important questions to be dealt with. The answer to them greatly influences the type of algorithms that are chosen and, to some extent, how they are coded.

SIMD instruction set extensions for multimedia - 1

SIMD multimedia extensions appeared following the observation that many media applications operate on narrower data types than the 32-bit processors were optimized for. Many graphics systems used 8 bits to represent each of the three primary colors plus 8 bits for transparency. Depending on the application, audio samples are usually represented in 8 or 16 bits. By partitioning the carry chains within a 256-bit adder, a processor could perform simultaneous operations on short vectors of 32 8-bit operands, 16 16-bit operands, 8 32-bit operands or 4 64-bit operands. The additional cost of such partitioned adders was small.

Summary of typical SIMD multimedia support for 256-bit wide operations

Source: Computer Architecture: A Quantitative Approach

instruction category	***operands***
unsigned add / subtract	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
maximum / minimum	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
average	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
shift right / left	32 8-bit, 16 16-bit, 8 32-bit, 4 64 bit
floating point	16 16-bit, 8 32-bit, 4 64 bit, 2 128-bit

SIMD instruction set extensions for multimedia - 2

Like vector instructions, a SIMD instruction specifies the same operation on data vectors. Unlike vector machines with large register banks, SIMD instructions tend to specify fewer operands and, therefore, use much smaller register banks.

In contrast to vector architectures, which offer an elegant instruction set intended to be the target of a vectorizing compiler, SIMD instructions present three major weaknesses

- they fix the number of data operands in the opcode, which has led to the addition of hundreds of instructions in the MMX, SSE and AVX extensions of the x86 architecture
- they do not offer the more sophisticated addressing modes of vector architectures, namely strided and gather-scatter accesses
- they do not usually offer the mask registers to support elemental conditional execution.

These weaknesses make it harder for the compiler to generate SIMD code and increase the difficulty of programming in assembly language.

SIMD instruction set extensions for multimedia - 3

For the x86 architecture, the MMX instructions, added in 1996, have reused the 64-bit floating point registers, so that the basic instructions could perform 8 8-bit operations or 4 16-bit operations simultaneously. These were joined by parallel MAX and MIN operations, a wide variety of masking and conditional instructions, operations typically found in digital signal processors (DSPs) and *ad hoc* instructions, believed to be useful in important media libraries. MMX also reused the floating point data transfer instructions to access memory.

The Streaming SIMD Extensions (SSE), its successor in 1999, added separate registers that were 128-bit wide, so now instructions could perform simultaneously 16 8-bit operations, 8 16-bit operations or 4 32-bit operations. It also performed parallel floating point arithmetic. Since SSE had separate registers, it needed separate data transfer instructions. Intel soon added double precision SIMD floating point data types via SSE2 in 2001, SSE3 in 2004 and SSE4 in 2007. Instructions with 4 single precision floating point operations or 2 double precision floating point operations increased the peak performance of x86 processors, as long as programmers placed the operands side by side. With each generation, new *ad hoc* instructions whose aim was to accelerate specific multimedia functions, were also introduced.

SIMD instruction set extensions for multimedia - 4

The Advanced Vector Extensions (AVX), added in 2010, doubled again the width of the registers to 256 bits and thereby offered instructions that doubled the number of operations on all narrower data types. AVX includes means to extend the registers width to 512 bits and 1024 bits in future generations of the architecture.

The goal of these extensions has been in general to accelerate carefully written libraries rather than for the compiler to produce them. Recent x86 compilers, however, are trying to generate such a code, particularly for floating point intensive applications.

Given these weaknesses, why are Multimedia SIMD Extensions so popular? There are several reasons for it

- it was not too costly to add them to standard arithmetic and they could be easily implemented
- they require little extra state compared to vector architectures, which is important in context switching
- they do not require a large memory bandwidth.

SIMD instruction set extensions for multimedia - 5

AVX instructions for x86 architecture useful in double precision FP programs

Source: Computer Architecture: A Quantitative Approach

AVX instruction	*description*
`VADDPD`	add 4 packed double precision operands
`VSUBPD`	subtract 4 packed double precision operands
`VMULPD`	multiply 4 packed double precision operands
`VDIVPD`	divide 4 packed double precision operands
`VFMADDPD`	multiply and add 4 packed double precision operands
`VFMSUBPD`	multiply and subtract 4 packed double precision operands
`VCMP__`	compare 4 packed double precision operands for EQ, NEQ, GT, LT, GE, LE
`VMOVAPD`	move aligned 4 packed double precision operands
`VBROAADCASTSD`	broadcast one double precision operands to 4 locations in a 256-bit register

SIMD instruction set extensions for multimedia - 6

MIPS SIMD code for DAXPY

```
        L.D     F0,a
        MOV     F1,F0       // copy a to F1 for SIMD MUL
        MOV     F2,F0       // copy a to F2 for SIMD MUL
        MOV     F3,F0       // copy a to F3 for SIMD MUL
        DADDIU  R4,Rx,512
Loop:   L.4D    F4,0(Rx)    // F0 = X[i], ... , F3 = X[i+3]
        MUL.4D  F4,F4,F0    // F4 = a*X[i], ... , F7 = a*X[i+3]
        L.4D    F8,0(Ry)    // F8 = Y[i], ... , F11 = Y[i+3]
        ADD.4D  F8,F8,F4    // F8 = a*X[i]+Y[i], ... ,
                            //          F11 = a*X[i+3]+Y[i+3]
        S.4D    F8,0(Ry)    // Y[i] = F8, ... , Y[i+3] = F11
        DADDIU  Rx,Rx,32
        DADDIU  Ry,Ry,32
        DSUBU   R20,R4,Rx
        BNEZ    R20,Loop
```

Programming multimedia SIMD architectures

Given the *ad hoc* nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries, or by writing directly the code in assembly language.

Recent extensions have become more regular, presenting the compiler with a more reasonable target. By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically. However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
  - Chapter 4: *Data-Level Parallelism in Vector, SIMD and GPU Architectures* – Sections 1 to 5
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
  - Chapter 19: *General Purpose Graphic Processing Units* – Sections 1 to 3