WIKIPEDIA

Vector processor

In computing, a **vector processor** or **array processor** is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called *vectors*, compared to the scalar processors, whose instructions operate on single data items. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 1990s, notably the various Cray platforms. The rapid fall in the price-to-performance ratio of conventional microprocessor designs led to the vector supercomputer's demise in the later 1990s.

As of 2016 most commodity CPUs implement architectures that feature SIMD instructions for a form of vector processing on multiple (vectorized) data sets. Common examples include Intel x86's MMX, SSE and AVX instructions, AMD's 3DNow! extensions, Sparc's VIS extension, PowerPC's AltiVec and MIPS' MSA. Vector processing techniques also operate in video-game console hardware and in graphics accelerators. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor.

Other CPU designs include some multiple instructions for vector processing on multiple (vectorised) data sets, typically known as MIMD (**M**ultiple **I**nstruction, **M**ultiple **D**ata) and realized with VLIW (**V**ery **L**ong **I**nstruction **W**ord). The Fujitsu FR-V VLIW/*vector processor* combines both technologies.

Contents

History

Early work

Vector processing development began in the early 1960s at Westinghouse in their "Solomon" project. Solomon's goal was to dramatically increase math performance by using a large number of simple math co-processors under the control of a single master CPU. The CPU fed a single common instruction to all of the arithmetic logic units (ALUs), one per cycle, but with a different data point for each one to work on. This allowed the Solomon machine to apply a single algorithm to a large data set, fed in the form of an array.

In 1962, Westinghouse cancelled the project, but the effort was restarted at the University of Illinois as the ILLIAC IV. Their version of the design originally called for a 1 GFLOPS machine with 256 ALUs, but, when it was finally delivered in 1972, it had only 64 ALUs and could reach only 100 to 150 MFLOPS. Nevertheless, it showed that the basic concept was sound, and, when used on data-intensive applications, such as computational fluid dynamics, the ILLIAC was the fastest machine in the world. The ILLIAC approach of using separate ALUs for each data element is not common to later designs, and is often referred to under a separate category, massively parallel computing.

A computer for operations with functions was presented and developed by Kartsev in 1967.[1]

Supercomputers

The first successful implementation of vector processing occurred in 1966, when both the Control Data Corporation STAR-100 and the Texas Instruments Advanced Scientific Computer (ASC) were introduced.

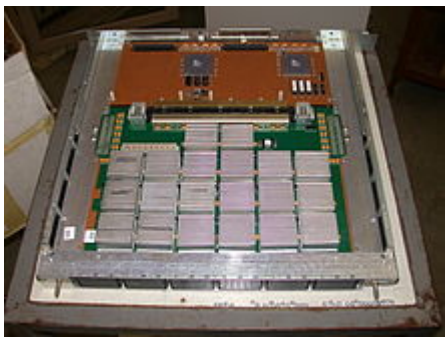
The basic ASC (i.e., "one pipe") ALU used a pipeline architecture that supported both scalar and vector computations, with peak performance reaching approximately 20 MFLOPS, readily achieved when processing long vectors. Expanded ALU configurations supported "two pipes" or "four pipes" with a corresponding 2X or 4X performance gain. Memory bandwidth was sufficient to support these expanded modes.

The STAR was otherwise slower than CDC's own supercomputers like the CDC 7600, but at data related tasks they could keep up while being much smaller and less expensive. However the machine also took considerable time decoding the vector instructions and getting ready to run the process, so it required very specific data sets to work on before it actually sped anything up.

The vector technique was first fully exploited in 1976 by the famous Cray-1. Instead of leaving the data in memory like the STAR and ASC, the Cray design had eight vector registers, which held sixty-four 64-bit words each. The vector instructions were applied between registers, which is much faster than talking to main memory. Whereas the STAR would apply a single operation across a long vector in memory and then move on to the next operation, the Cray design would load a smaller section of the vector into registers and then apply as many operations as it could to that data, thereby avoiding many of the much slower memory access operations.

The Cray design used pipeline parallelism to implement vector instructions rather than multiple ALUs. In addition, the design had completely separate pipelines for different instructions, for example, addition/subtraction was implemented in different hardware than multiplication. This allowed a batch of vector instructions to be pipelined into each of the ALU subunits, a technique they called *vector chaining*. The Cray-1 normally had a performance of about 80 MFLOPS, but with up to three chains running it could peak at 240 MFLOPS and averaged around 150 – far faster than any machine of the era.

Other examples followed. Control Data Corporation tried to re-enter the high-end market again with its ETA-10 machine, but it sold poorly and they took that as an opportunity to leave the supercomputing field entirely. In the early and mid-1980s Japanese companies (Fujitsu, Hitachi and Nippon Electric Corporation (NEC) introduced register-based vector machines similar to the Cray-1, typically being slightly faster and much smaller. Oregon-based Floating Point Systems (FPS) built add-on array processors for minicomputers, later building their own minisupercomputers.



Cray J90 processor module with four scalar/vector processors

Throughout, Cray continued to be the performance leader, continually beating the competition with a series of machines that led to the Cray-2, Cray X-MP and Cray Y-MP. Since then, the supercomputer market has focused much more on massively parallel processing rather than better

implementations of vector processors. However, recognising the benefits of vector processing IBM developed Virtual Vector Architecture for use in supercomputers coupling several scalar processors to act as a vector processor.

Although vector supercomputers resembling the Cray-1 are less popular these days, NEC has continued to make this type of computer up to the present day, with their SX series of computers. Most recently, the SX-Aurora TSUBASA places the processor and either 24 or 48 gigabytes of memory on an HBM 2 module within a card that physically resembles a graphics coprocessor, but instead of serving as a co-processor, it is the main computer with the PC-compatible computer into which it is plugged serving support functions.

GPU

Modern graphics processing units (GPUs) include an array of shader pipelines which may be driven by compute kernels, which can be considered vector processors (using a similar strategy for hiding memory latencies).

Description

In general terms, CPUs are able to manipulate one or two pieces of data at a time. For instance, most CPUs have an instruction that essentially says "add A to B and put the result in C". The data for A, B and C could be—in theory at least—encoded directly into the instruction. However, in efficient implementation things are rarely that simple. The data is rarely sent in raw form, and is instead "pointed to" by passing in an address to a memory location that holds the data. Decoding this address and getting the data out of the memory takes some time, during which the CPU traditionally would sit idle waiting for the requested data to show up. As CPU speeds have increased, this *memory latency* has historically become a large impediment to performance; see Memory wall.

In order to reduce the amount of time consumed by these steps, most modern CPUs use a technique known as instruction pipelining in which the instructions pass through several sub-units in turn. The first sub-unit reads the address and decodes it, the next "fetches" the values at those addresses, and the next does the math itself. With pipelining the "trick" is to start decoding the next instruction even before the first has left the CPU, in the fashion of an assembly line, so the address decoder is constantly in use. Any particular instruction takes the same amount of time to complete, a time known as the *latency*, but the CPU can process an entire batch of operations much faster and more efficiently than if it did so one at a time.

Vector processors take this concept one step further. Instead of pipelining just the instructions, they also pipeline the data itself. The processor is fed instructions that say not just to add A to B, but to add all of the numbers "from here to here" to all of the numbers "from there to there". Instead of constantly having to decode instructions and then fetch the data needed to complete them, the processor reads a single instruction from memory, and it is simply implied in the definition of the instruction *itself* that the instruction will operate again on another item of data, at an address one increment larger than the last. This allows for significant savings in decoding time.

To illustrate what a difference this can make, consider the simple task of adding two groups of 10 numbers together. In a normal programming language one would write a "loop" that picked up each of the pairs of numbers in turn, and then added them. To the CPU, this would look something like this:

```
; Hypothetical RISC machine
; add 10 numbers in a to 10 numbers in b, storing results in c
; assume a, b, and c are memory locations in their respective registers
  move  $10, count   ; count := 10
loop:
  load  r1, a
  load  r2, b
  add   r3, r1, r2   ; r3 := r1 + r2
  store r3, c
```

```
    add   a, a, $4      ; move on
    add   b, b, $4
    add   c, c, $4
    dec   count         ; decrement
    jnez  count, loop   ; loop back if count is not yet 0
    ret
```

But to a vector processor, this task looks considerably different:

```
; assume we have vector registers v1-v3 with size larger than 10
    move   $10, count    ; count = 10
    vload  v1, a, count
    vload  v2, b, count
    vadd   v3, v1, v2
    vstore v3, c, count
    ret
```

There are several savings inherent in this approach. For one, only two address translations are needed. Depending on the architecture, this can represent a significant savings by itself. Another saving is fetching and decoding the instruction itself, which has to be done only one time instead of ten. The code itself is also smaller, which can lead to more efficient memory use.

But more than that, a vector processor may have multiple functional units adding those numbers in parallel. The checking of dependencies between those numbers is not required as a vector instruction specifies multiple independent operations. This simplifies the control logic required, and can improve performance by avoiding stalls. The math operations thus completed far faster overall, the limiting factor being the time required to fetch the data from memory.

Not all problems can be attacked with this sort of solution. Including these types of instructions necessarily adds complexity to the core CPU. That complexity typically makes *other* instructions run slower—i.e., whenever it is **not** adding up many numbers in a row. The more complex instructions also add to the complexity of the decoders, which might slow down the decoding of the more common instructions such as normal adding.

In fact, vector processors work best only when there are large amounts of data to be worked on. For this reason, these sorts of CPUs were found primarily in supercomputers, as the supercomputers themselves were, in general, found in places such as weather prediction centers and physics labs, where huge amounts of data are "crunched".

Vector instructions

The vector pseudocode example above comes with a big assumption that the vector computer can process more than ten numbers in one batch. For a greater number of numbers, it becomes unfeasible for the computer to have a register that large. As a result, the vector processor either gains the ability to perform loops itself, or exposes some sort of vector register to the programmer.

The self-repeating instructions are found in early vector computers like the STAR, where the above action would be described in a single instruction (somewhat like `vadd c, a, b, $10`). They are also found in the x86 architecture as the `REP` perfix. However, only very simple calculations can be done effectively in hardware without a very large cost increase. Since all operands has to be in-memory, the latency caused by access becomes huge too.

The Cray-1 introduced the idea of using processor registers to hold vector data in batches. This way, a lot more work can be done in each batch, at the cost of requiring the programmer to manually load/store data from/to the memory for each batch. Modern SIMD computers improve on Cray by directly using multiple ALUs, for a higher degree of parallelism compared to only using the normal scalar pipeline. Masks can be used to selectively load or store memory locations for a version of parallel logic.

GPUs, which have many small compute units, use a variant of SIMD called Single Instruction Multiple Threads (SIMT). This is similar to modern SIMD, with the excption that the "vector registers" are very wide and the pipelines tend to be long. The "threading" part affects the way data are swapped between the compute units. In addition, GPUs and other external vector processors like the NEC SX-Aurora TSUBASA may use fewer vector units than the width implies: instead of having 64 units for a 64-number-wide register, the hardware might instead do a pipelined loop over 16 units for a hybrid approach.

The difference between a traditional vector processor and a modern SIMD one can be illustrated with this variant of the "DAXPY" function:

```
void iaxpy(size_t n, int a, const int x[], int y[]) {
    for (size_t i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

The STAR-like code remains concise, but we now require an extra slot of memory to process the information. Two times the latency is also needed due to the extra requirement of memory access.

```
  ; Assume tmp is pre-allocated
  vmul tmp, a, x, n ; tmp[i] = a * x[i]
  vadd y, y, tmp, n ; y[i] = y[i] + tmp[i]
  ret
```

This modern SIMD machine can do most of the operation in batches. The code is mostly similar to the scalar version. We are assuming that both x and y are properly aligned here and that n is a multiple of 4, as otherwise some setup code would be needed to calculate a mask or to run a scalar version. The time taken would be basically the same as a vector implementation of `c = a + b` described above.

```
vloop:
  load32x4  v1, x
  load32x4  v2, y
  mul32x4   v1, a, v1    ; v1 := v1 * a
  add32x4   v3, v1, v2   ; v3 := v1 + v2
  store32x4 v3, y
  addl      x, x, $16    ; a := a + 16
  addl      y, y, $16
  subl      n, n, $4     ; n := n - 4
  jgz       n, vloop     ; loop back if n > 0
out:
  ret
```

Performance and speed up

Let r be the vector speed ratio and f be the vectorization ratio. If the time taken for the vector unit to add an array of 64 numbers is 10 times faster than its equivalent scalar counterpart, r = 10. Also, if the total number of operations in a program is 100, out of which only 10 are scalar (after vectorization), then f = 0.9, i.e., 90% of the work is done by the vector unit. It follows the achievable speed up of:

$$r/[(1-f)*r+f]$$

So, even if the performance of the vector unit is very high ($r=\infty$) we get a speedup less than $1/(1-f)$, which suggests that the ratio f is crucial to the performance. This ratio depends on the efficiency of the compilation like adjacency of the elements in memory.

Programming heterogeneous computing architectures

Various machines were designed to include both traditional processors and vector processors, such as the Fujitsu AP1000 and AP3000. Programming such heterogeneous machines can be difficult since developing programs that make best use of characteristics of different processors increases the programmer's burden. It increases code complexity and decreases portability of the code by requiring hardware specific code to be interleaved throughout application code.[2] Balancing the application workload across processors can be problematic, especially given that they typically have different performance characteristics. There are different conceptual models to deal with the problem, for example using a coordination language and program building blocks (programming libraries or higher order functions). Each block can have a different native implementation for each processor type. Users simply program using these abstractions and an intelligent compiler chooses the best implementation based on the context.[3]

See also

- SX architecture
- GPGPU
- Compute kernel
- Stream processing
- SIMD
- Automatic vectorization
- Chaining (vector processing)
- Computer for operations with functions
- RISC-V, an open ISA standard with an associated variable width vector extension.
- Barrel processor
- Tensor processing unit

References

1. Malinovsky, B.N. (1995). *The history of computer technology in their faces (in Russian)*. Kiew: Firm "KIT". ISBN 5-7707-6131-8.
2. Kunzman, D. M.; Kale, L. V. (2011). "Programming Heterogeneous Systems". *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. p. 2061. doi:10.1109/IPDPS.2011.377 (https://doi.org/10.1109%2FIPDPS.2011.377). ISBN 978-1-61284-425-1.
3. John Darlinton; Moustafa Ghanem; Yike Guo; Hing Wing To (1996), "Guided Resource Organisation in Heterogeneous Parallel Computing", *Journal of High Performance Computing*, **4** (1): 13–23, CiteSeerX 10.1.1.37.4309 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.4309)

External links

- The History of the Development of Parallel Computing (http://ei.cs.vt.edu/~history/Parallel.html) (from 1955 to 1993)