Journal Of Harmonized Research (JOHR)

Journal Of Harmonized Research in Engineering
1(2), 2013, 161-166

ISSN 2347 – 7393

Review article

IMPORTANCE OF VECTOR PROCESSING

Naveen Malik, Pankaj Sharma, Naeem Akhtar, Rahul, Hardeep Rohilla

Dronacharya College of Engineering, Khentawas,
Farukhnagar, Gurgaon, India

Abstract
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*. This is in contrast to a scalar processor, 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 90s, notably the various Cray platforms. The rapid rise in the price-to-performance ratio of conventional microprocessor designs led to the vector supercomputer's demise in the later 1990s.
Today, most commodity CPUs implement architectures that feature instructions for a form vector processing on multiple (vectorized) data sets, typically known as SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata). Common examples include VIS, MMX, SSE, AltiVec and AVX. Vector processing techniques are also found in video game console hardware and graphics accelerators. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor, consisting of one scalar processor and eight vector processors, which found use in the Sony PlayStation 3 among other applications.
Other CPU designs may 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. Such designs are usually dedicated to a particular application and not commonly marketed for general purpose computing. In the Fujitsu FR-V VLIW/*vector processor* both technologies are combined.

For Correspondence:
malik2008ATgmail.com
Received on: November 2013
Accepted after revision: December 2013
Downloaded from: www.johronline.com

1. Introduction

Vector processing was once intimately associated with the concept of a "supercomputer". As with most architectural techniques for achieving high performance, it exploits regularities in the structure of computation, in this case, the fact that many codes contain loops that range over linear arrays of data performing symmetric operations. The origins of vector architecure lay in trying to address the problem of instruction bandwidth. By the end of the 1960's, it was possible to build multiple pipelined functional units, but the fetch and decode of instructions from memory was too slow to permit them to be fully exploited. Applying a single instruction to multiple data elements (SIMD) is one simple and logical way to leverage limited instruction bandwidth.

The most powerful computers of the 1970s and 1980s tended to be vector machines, from Cray, NEC, and Fujitsu, but with increasingly higher degrees of semiconductor integration, the mismatch between instruction bandwidth and operand bandwidth essentially went away. As of 2009, only 1 of the worlds top 500 supercomputers was still based on a vector architecture.The lessons of SIMD processing weren't entirely lost, however. While Cray-style vector units that perform a common operations across vector registers of hundreds or thousands of data elements have largely disappeared, the SIMD approach has been applied to the processing of 8 and 16-bit multimedia data by 32 and 64-bit processors and DSPs with great success. Under the names "MMX" and "SSE", SIMD processing can be found in essentially every modern personal computer, where it is exploited by image processing and audio applications. A Vector processor is a processor that can operate on an entire vector in one instruction. The operands to the instructions are complete vectors instead of one element. Vector processors reduce the fetch and decode bandwidth as the numbers of instructions fetched are less. They also exploit data parallelism in large scientific and multimedia applications. Based on how the operands are fetched, vector processors can be divided into two categories - in memory-memory architecture operands are directly streamed to the functional units from the memory and results are written back to memory as the vector operation proceeds. In vector-register architecture, operands are read into vector registers from which they are fed to the functional units and results of operations are written to vector registers. Many performance optimization schemes are used in vector processors. Memory banks are used to reduce load/store latency. Strip mining is used to generate code so that vector operation is possible for vector operands whose size is less than or greater than the size of vector registers. Vector chaining the equivalent of forwarding in vector processors - is used in case of data dependency among vector instructions. Special scatter and gather instructions are provided to efficiently operate on sparse matrices.

Instruction set has been designed with the property that all vector arithmetic instructions only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel lanes. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code.

2. Brief History:-

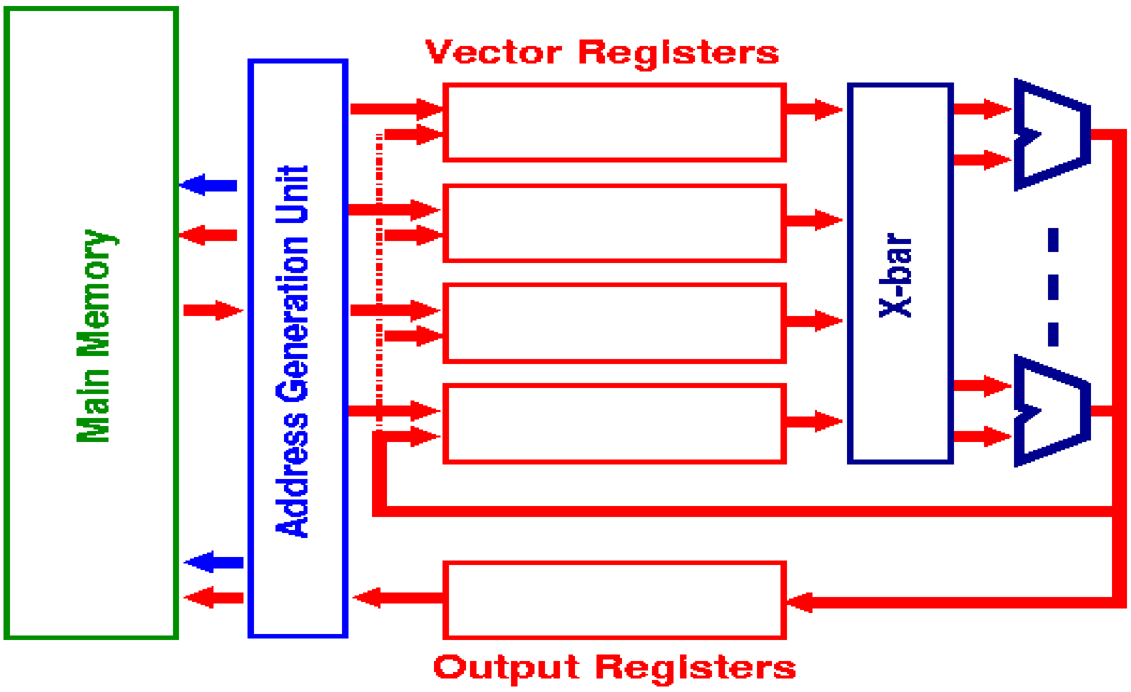
Vector processing development began in the early 1960s at Westinghouse in their **Solomon** project. Solomon's goal was to dramatically increase math performance by using a large number of simple math co-processors under the control of a single master CPU. The CPU fed a single common instruction to all 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 "failed" 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

3. Structure of Vector processors:

Commonly called **supercomputers**, the vector processors are machines built primarily to handle large scientific and engineering calculations. Their performance derives from a heavily pipelined architecture which operations on vectors and matrices can efficiently exploit.

Vector Registers



Anatomy of a typical vector processor showing the vector registers and multiple floating point ALUs.

The "conventional" scalar processing units are not shown.

Data is read into the **vector registers** which are FIFO queues capable of holding 50-100 floating point values. A machine will be provided with several vector registers, V_a, V_b, *etc*. The instruction set will contain instruction which:

- load a vector register from a location in memory,
- perform operations on elements in the vector registers and
- store data back into memory from the vector registers.

Thus a program to calculate the dot-product of two vectors might look like this:

V_load V_a, add_A
V_load V_b, add_B
V_multiply V_c, V_a, V_b
V_sum R_1,V_c

1) where the last operation sums the elements in vector register C and stores the result in a scalar register, R_1.

4. Implematation:-

4.1. Supercomputers:-

The first *successful* implementation of vector processing appears to be the Control Data Corporation STAR-100 and the Texas Instruments Advanced Scientific Computer (ASC). 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. 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 themselves to be pipelined, 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 – a respectable number even as of 2002.

Cray J90 processor module with four scalar/vector processors

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. However 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.

4.2. SIMD

Vector processing techniques have since been added to almost all modern CPU designs, although they are typically referred to as SIMD. In these implementations, the vector unit runs beside the main scalar CPU, and is fed data from vector instruction aware programs

4.3. 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:

```
execute this loop 10 times
  read the next instruction and decode it
  fetch this number
  fetch that number
  add them
  put the result here
end loop
```

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

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
add them
put the results here
```

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. As mentioned earlier, the Cray implementations took this a step further, allowing several different types of operations to be carried out at the same time. Consider code that adds two numbers and then multiplies by a third; in the Cray, these would all be fetched at once, and both added and multiplied in a single operation. Using the pseudocode above, the Cray did:

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
fetch another 10 numbers
add and multiply them
put the results here
```

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. Adding these sorts 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 centres and physics labs, where huge amounts of data are "crunched"

5.Conclution:-

Vector supercomputers are not viable due to cost reason, but vector instruction set architecture is still useful. Vector supercomputers are adapting commodity technology like SMT to improve their price-performance. Superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems (Ex - Intel MMX extension). Vector processors are useful for embedded and multimedia applications which require low power, small code size and high performance.

6.References

[1]. http://www.wikipedia.org

[2]. Roger Espasa, Mateo Valero, James E. Smith, ” Vector Architectures: Past, Present and Future”, in Supercomputing, 1998.

[3]. Hennessy/Patterson Appendix G: Vector Processing Appendix G

[4]. C. Kozyrakis, D. Patterson, ” Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks”, in MICRO, 2002.