For some applications, data naturally comes in vector or matrix form.

For example, a vector of digitized samples representing an audio waveform over time, or a matrix of pixel colors in a 2D image from a camera.

When processing that data, it's common to perform the same sequence of operations on each data element.

The example code shown here is computing a vector sum, where each component of one vector is added to the corresponding component of another vector.

By replicating the datapath portion of our CPU, we can design special-purpose vector processors capable of performing the same operation on many data elements in parallel.

Here we see that the register file and ALU have been replicated and the control signals from decoding the current instruction are shared by all the datapaths.

Data is fetched from memory in big blocks (very much like fetching a cache line) and the specified register in each datapath is loaded with one of the words from the block.

Similarly each datapath can contribute a word to be stored as a contiguous block in main memory.

In such machines, the width of the data buses to and from main memory is many words wide, so a single memory access provides data for all the datapaths in parallel.

Executing a single instruction on a machine with N datapaths is equivalent to executing N instructions on a conventional machine with a single datapath.

The result achieves a lot of parallelism without the complexities of out-of-order superscalar execution.

Suppose we had a vector processor with 16 datapaths.

Let's compare its performance on a vector-sum operation to that of a conventional pipelined Beta processor.

Here's the Beta code, carefully organized to avoid any data hazards during execution.

There are 9 instructions in the loop, taking 10 cycles to execute if we count the NOP introduced into the pipeline when the BNE at the end of the loop is taken.

It takes 160 cycles to sum all 16 elements assuming no additional cycles are required due to cache misses.

And here's the corresponding code for a vector processor where we've assumed constant-sized 16-element vectors.

Note that "V" registers refer to a particular location in the register file associated with each datapath, while the "R" registers are the conventional Beta registers used for address computations, etc.

It would only take 4 cycles for the vector processor to complete the desired operations, a speed-up of 40.

This example shows the best-possible speed-up.

The key to a good speed-up is our ability to "vectorize" the code and take advantage of all the datapaths operating in parallel.

This isn't possible for every application, but for tasks like audio or video encoding and decoding, and all sorts of digital signal processing, vectorization is very doable.

Memory operations enjoy a similar performance improvement since the access overhead is amortized over large blocks of data.

You might wonder if it's possible to efficiently perform data-dependent operations on a vector processor.

Data-dependent operations appear as conditional statements on conventional machines, where the body of the statement is executed if the condition is true.

If testing and branching is under the control of the single-instruction execution engine, how can we take advantage of the parallel datapaths?

The trick is provide each datapath with a local predicate flag.

Use a vectorized compare instruction (CMPLT.V) to perform the a[i] b[i] comparisons in parallel and remember the result locally in each datapath's predicate flag.

Then extend the vector ISA to include "predicated instructions" which check the local predicate to see if they should execute or do nothing.

In this example, ADDC.V.iftrue only performs the ADDC on the local data if the local predicate flag is true.

Instruction predication is also used in many non-vector architectures to avoid the execution-time penalties associated with mis-predicted conditional branches.

They are particularly useful for simple arithmetic and boolean operations (i.e., very short instruction sequences) that should be executed only if a condition is met.

The x86 ISA includes a conditional move instruction, and in the 32-bit ARM ISA almost all instructions can be conditionally executed.

The power of vector processors comes from having 1 instruction initiate N parallel operations on N pairs of operands.

Most modern CPUs incorporate vector extensions that operate in parallel on 8-, 16-, 32- or 64-bit operands organized as blocks of 128-, 256-, or 512-bit data.

Often all that's needed is some simple additional logic on an ALU designed to process full-width operands.

The parallelism is baked into the vector program, not discovered on-the-fly by the instruction dispatch and execution machinery.

Writing the specialized vector programs is a worthwhile investment for certain library functions which see a lot use in processing today's information streams with their heavy use of images, and A/V material.

Perhaps the best example of architectures with many datapaths operating in parallel are the graphics processing units (GPUs) found in almost all computer graphics systems.

GPU datapaths are typically specialized for 32- and 64-bit floating point operations found in the algorithms needed to display in real-time a 3D scene represented as billions of triangular patches as a 2D image on the computer screen.

Coordinate transformation, pixel shading and antialiasing, texture mapping, etc., are examples of "embarrassingly parallel" computations where the parallelism comes from having to perform the same computation independently on millions of different data objects.

Similar problems can be found in the fields of bioinformatics, big data processing, neural net emulation used in deep machine learning, and so on.

Increasingly, GPUs are used in many interesting scientific and engineering calculations and not just as graphics engines.

Data-level parallelism provides significant performance improvements in a variety of useful situations.

So current and future ISAs will almost certainly include support for vector operations.