Lecture 8 notes

We discuss vector processors. The major property of these processors is SIMD (Single Instruction Multiple Data) execution. Vector processors execute instructions representing one operation (e.g. add) on pairs of data vectors, storing the result in a vector as well.

Vector processors appeared first as “supercomputers”. They operated on some of the fastest machines of the planet (at the time). Supercomputers then and now tend to emphasize high speed of computation on data residing in on-board, on-chip memories, allocating most of their budget to extremely powerful processors, or a very large number of processors. Typically, they are very expensive (millions of euros) and they are used primarily in science and engineering applications that are extremely demanding in terms of computation (e.g. climate modeling, industrial design, material science, etc.). In the 1970s and 1980s supercomputer architecture was dominated by vector processors.

We introduce vector processors from a different perspective. We have seen several examples of exploiting ILP using dynamic scheduling, VLIW, etc. Practically all examples were showing loops operating on arrays or vectors (e.g. multiplying a vector by a scalar, adding two vectors etc.). Vector processors introduce instructions that load vectors, store vectors and perform arithmetic on vectors. These instructions are “parallel” in the sense that each instructions applies an operation to a vector of data simultaneously. A vector instruction encodes multiple operations proceedings in parallel. The implementation of this operations can itself be pipelined. Note that like VLIW, vector instructions guarantee that instructions are independent and can be executed in parallel. Note that unlike VLIW, where a long instructions may contain individual micro-instructions of different types (e.g. memory and arithmetic), a vector instruction encodes one type of instruction applied to multiple data.

Vector processors share some common components with conventional superscalar and VLIW processors. There are multiple execution units, and the execution of instructions is typically pipelined. Vector processors are also “load/store” architectures, however individual operations use vector registers (registers packing multiple words, typically 2-16). Vector processors often do not have data caches, using instead the wide registers as a form of very fast on-chip data cache. This is not a rule and there are important exceptions of vector processors that use scratchpad memories as fast data caches (e.g. SPE cores on the Cell/BE). Vector processors often also avoid entirely virtual memory and translation, as this can slow down dramatically vector memory accesses. On the other hand, vector processors often provide a highly parallel multi-bank and pipeline memory system, so that vector memory operations can read and write data from/to memory faster than conventional processors. This assumes that the vector processor is connected to a bus/network with enough bandwidth to match the supply with the demand for data.

Vector processors have typically simpler and more comprehensive instruction sets than conventional processors. The instruction set is compact, since each instruction represents multiple operations. The instruction set is also expressive, in the sense that each vector instruction guarantees independence (therefore there is no need for hazard control), distributes naturally the data in registers, and accesses data in a regular pattern (e.g. adjacent data in memory, or data separated in memory by fixed strides). Further, the whole vector execution pattern is scalable, in the sense that we can keep throwing more hardware at the processor (more execution units, wider registers) and scale up the performance of vector instructions by a multiplicative factor (at least in theory).

We discuss the basic principles of vector processors, including:

- vector code structure (in particular, how compact is vector code compared to scalar code for loops)
- the concept of lanes (i.e. multiple pipelined functional units supporting execution on data located in vector registers). We discuss lane parallelism in relation to vector length and how a combination of parallelism and pipelining achieves higher performance in vector processors
- the concept of stripmining (i.e. how we transform an innermost parallel loop to a loop using vector instructions, regardless of the number of iterations in the loop). Stripmining is a useful transformation for organizing a loop in "blocks of iterations". These blocks can be executed using vector instructions. On shared-memory multiprocessors, the blocks can be distributed between processors.
- the concept of chaining (the equivalent of forwarding on vector processors). We allow a vector instruction A to forward data between to another (dependent) vector instruction B, without waiting for A to writeback all the produces results to instruction B.
- vector gather/scatter operations: the gather operation "collects" words that are located in sparse memory locations and "compress" them into a vector register. The scatter operation "scatters" (distributes) words from a vector register to various memory locations. Both operations use an indirection vector to find the memory locations to collect form or scatter to.
- Vector conditional execution: we can "select" specific words from vector registers while performing arithmetic operations using a mask register. We discuss two implementations, one using the mask register after the operation and a second using the mask register before the operation. We have discussed some performance and power trade-off's of the two operations.

A special case of vector-like execution support for other (e.g. superscalar) processors is popularly called "multimedia extensions" (MMX, SSE, AltiVec, etc.). These extensions masquerade general purpose registers as vector registers (e.g. by "packing" virtually 8 8-bit operands in a 64-bit registers and applying vector operations on them). Multimedia extensions are typically less powerful than vector instruction sets (they usually lack scatter/gather and mask operations), the vector

register length is limited (no more than 128 bits with current technologies), and the hardware or compiler still need to apply renaming and loop unrolling to keep execution units busy.

How do vector processors fare against superscalar, VLIW and other single-thread ILP processors?

The answer depends on the assumptions. If both processors have the same complexity (e.g. transistors), it is likely that the vector processor can exploit more instruction-level parallelism in codes with high degrees of inherent ILP, therefore higher performance. Furthermore, the instruction issue and the overall control logic of the vector processor can be much simpler than the issue and control logic of the superscalar processor. On the other hand, the vector processor assumes that a compiler is capable enough to extract and exploit instruction-level parallelism using instructions. Automatic compiler-based vectorization is a hard task and has not been perfected even in the best commercial compilers available in 2008.