Lecture Notes: SIMD Processing (02/26/09)

SIMD

The term SIMD (Single Instruction Multiple Data) was coined by Mike Flynn of Stanford in 1960's.
It generally refers to a class of processor architectures, in which multiple execution units proceed in lockstep executing the same instruction sequence, while processing distinct data items at the same time.

High Level Taxonomy of Processor Architectures

If we classify processor architectures based on the instructions executed and data elements processed in single step, we get following classification

	Data	
Instructions	Single	Multiple
Single	SISD Simple Processors, which execute one instruction at a time, while processing single data item	SIMD Multiple execution units, proceed in lock step, executing the same instruction while processing distinct data items
Multiple	MISD No one has come up with practical processors which use this paradigm yet!	MIMD Generic multiprocessors - e.g. SMP/ NUMA systems

For a generic processor, non trivial branches is the main hurdle for achieving performance. It causes cache misses and pipeline stalls. On the other hand, if the program control is predictable and the application is massively data parallel, SIMD is a good option to exploit the parallelism.

Examples of SIMD Processors

- Vector Processors: Cray
- Array Processors: CM
- GPU's: Nvidia, Cell BE Engine
- Processor Extensions: MMX, Altvec

Principle of Execution

SIMD processors (slaves) are attached to main CPU (master). The master handles complex control in the program and offloads data processing work to slaves. The slaves process distinct data elements in parallel, effectively achieving a much higher IPC compared to a simple SISD processor.

Instruction Set Considerations for SIMD

1. Consider following scalar operation:

a <-- b + c

The equivalent SISD instruction is

ADD a, b, c.

where a, b, c are processor registers.

Now consider a vector instruction for addition just like one above

VADD A, B, C

Here although A, B and C are still processor registers, they do not hold a single value. For example consider a 4-way SIMD processor register file as one below:

Reg-A, Loc-0	Loc-1	Loc-2	Loc-3
Reg-B, Loc-0	Loc-1	Loc-2	Loc-3
Reg-C, Loc-0	Loc-1	Loc-2	Loc-3

All the operation in SIMD like VADD are vector operations, which manipulate multiple values at a time. The VADD instruction, on a 4-way SIMD will add 4 values at a time, producing 4 results.

2. Operations exclusive to SIMD

Few instructions are only possible on SIMD architectures, due to their array processing capabilities.

e.g. consider the dot production operation: s = sum a_i * b_i ------ (0 <= i < N)

This operation is a special instruction exclusive to SIMD processors. It produces a scalar result by multiplying corresponding elements of two vectors and adding them together. Clearly, the multiplication of individual vector elements can be done in parallel, and so in constant time. But what about final addition?

Parallel add can be done in O(log n) time, using divide and conquer algorithm. However, for any SIMD architecture, N is an architectural constant (e.g. in 4-Way SIMD, N=4) and so dot product as instruction, can always be achieved in fixed number of cycles.

Generally, reductions such as min, max, etc. which can take advantage of associativity of operation can be implemented as instruction on SIMD.

Other interesting instructions possible on SIMD include scatter, gather, rotate, etc.

<u>Other considerations in SIMD design</u>

Why not branches?

Consider following C code:

z[i] = x[i] ? y[i] : y[i] + 1

How can the code execute in SIMD model? It obviously cannot, because depending on individual values of x[i], each SIMD unit will decide its execution path, and so they cannot proceed in lock step. Whenever there are data dependent branches in the code, the SIMD paradigm cannot be used. The way around this problem is to structure the code in a way to avoid them.

Where does the data come from?

Load/Store instructions in SIMD need to be designed with a consideration to vector operations.
Consider a 4-way SIMD system. To keep the CPU busy, the V-LOAD V-STORE instructions should process 4 elements at a time. There are different ways of designing memory instructions.

1. Specify all memory addresses to be used for load/store: This is a bad choice because it makes the instruction size larger, is more expensive to execute and may result in cache misses.
2. Specify a start address: This technique overcomes all the above disadvantages, but it is inflexible.

Generally, all SIMD architectures follow 2nd alternative. Due to this, SIMD programs are constrained.

SIMD Programming Constraints:

1. Data alignment: Data needs to be aligned at boundaries consistent with SIMD memory subsystem architecture
2. Data arrangement: Arrangement needs to be in such a way that the access pattern of code execution does not cause excessive cache misses.

Memory systems are built into banks. This allows multiple data paths for load/store of vectors.