Contents

Vector Parallel Computing

- Vector computers
  - One of the more powerful early supercomputers
  - SIMD model
- Mathematicians call a 1D array a vector
  - Vector instructions operate on 1D array
  - Usually a short subsequence at a time
  - Operations on 2D arrays built from 1D primitives

Vector Extensions

- Many processor designers have added a vector unit to their existing processors

  - Intel
    * MMX
    * SSE
    * AVX
  - ARM
    * NEON
  - PowerPC
    * AltiVec

Programming Vector Machines

- Program in plain C
  - Hope the compiler will vectorise the code
  - Two big problems for compiler automatic vectorisation
    * Pointers
      · Compiler has no idea whether points point to the same memory
      · Restrict pointers guarantee not to point to same memory as another restrict pointer
    * Data layouts
      · `struct point { floats x, y, z; };`
      · `struct points { float x[1024], y[1024], z[1024]; };`
- Domain specific languages
- Assembly
  - Complete control over the machine
  - Tedious, error prone, hard to maintain
- Compiler "intrinsics"
  - Functions that are built into the compiler
  - Vector machine instructions are expressed as C function calls
  - Middle ground between assembly and pure C

Compiler Intrinsics

- We will do vector programming using compiler intrinsics
- It's sort of low-level, but we want to understand the architectures

Operating Across Lanes

- `_mm_hadd_ps(_m128 a, m_128 b);`
  - `a = [a3, a2, a1, a0];`
  - `b = [b3, b2, b1, b0];`
  - `[a3+a2, a1+a0, b3+b2, b1+b0];`
- Vector "swizzles" can also operate across lanes
  - Permute: re-orders lanes within a vector
  - Blend: Select corresponding lanes from two vectors
  - Shuffle: A combination of permutes and blends

Vector Swizzles

- There are *lots* of swizzle operations in most vector instruction sets
  - Lots of restrictions and special cases
- This may seem odd
  - Its much easier for programmers and compiler writers to use a small number of general instructions

Vector Permutation

- Arbitrary permute instructions are expensive to implement
  - E.g. vector with four lanes
  - Each of the four result lanes can be any of the four input lanes
  - 4^4 - 256 possible permutations
- To implement this instruction we need
  - A circuit that can map any input lane to any output lane
  - An operand that specifies which of the possible 256 permutations should be implemented in this case
- Permutation Circuit
  - For each output lane
    - Select from any of the input lanes
    - #lanes == N
    - bit-width of lane == B
    - Circuit of at last O(BN) gates per output lane
      - Using "log shifter" circuits
    - Total circuit at least $O(BN^2\log_2(N))$ gates

        * Bigger circuits are slower and may need pipelining
        * Lots of cross wirse interconnection
    - For #lanes == 4, this is not a problem
    - For #lanes == 32, this is a big problem
- Specifying which permutation
    - For #lanes == N
    - Number of possible permutations is N^N
        * E.g. N == 4, N^N == 256 == 2^8
            · Requires 1 byte to specify which permutation
        * E.g. N == 8, N^N == 16,777,216 == 2^{24}
            · Requires 3 byte to specify which permutation
        * E.g. N == 16, N^N == 1.845 x 10^{19} == 2^{24}
            · Requires 8 byte to specify which permutation

Vector Swizzling

- Several vector permutation instructions in real instruction sets
    - But often restrictions to reduce
        * Circuit complexity
        * Instruction length
- `_mm_shuffle_ps(a, b, _MM_SHUFFLE(1, 0, 3, 2));`
    - `a = [a3, a2, a1, a0];`
    - `b = [b3, b2, b1, b0];`
    - `[a1, a0, b3, b2];`

Accessing Individual Lanes

- Most vector instruction sets have some mechanism to insert/extract a value to/from an individual lane within a vector
- In Intel SSE, only works with integer register
- In practice, the lack of floating point insert/extract instructions means we can efficiently load/store floating point values in memory

Bitwise Operations

- Like most vector instruction sets, SSE provides a variety of bitwise instructions
- Bitwise vector operations allow a lot of bitwise parallelism
    - e.g. union/intersection of bit vector sets

Comparison Operations

- We can compare corresponding lanes of pairs of vectors
- The result of a comparison is always a mask of values
  - Integer 0 in false lanes (`0000000000000`)
  - Integer -1 in true lanes (`1111111111111`)
- Masking can be used to vectorise many loops containing if statements
- SSE also allows us to convert a full-width vector mask to a bit-by-bit integer mask
  - `int _mm_movemask_ps(mask);`
  - Integer can be stored to summarize comparison
  - Integer can be tested with if or switch statement

Max/Min

- Finding the minimum or maximum of pairs of numbers is so common that most vector instruction sets provide instructions