HPC VT 2013

Machine-dependent Optimization II
Overview of instruction pipelining & memory hierarchy

- Loop unrolling and Loop fusion
- Avoiding branches in a loop
- Use of blocking for cache-efficiency
Flynn's Taxonomy

- Single thread
- Vector processors, GPU, FPGA
- Fault tolerant systems (rare)
- Multicore, Cluster
Vector Operation

Source vector a

\[
\begin{align*}
a_3 & \quad a_2 & \quad a_1 & \quad a_0 \\
\end{align*}
\]

Source vector b

\[
\begin{align*}
b_3 & \quad b_2 & \quad b_1 & \quad b_0 \\
\end{align*}
\]

\( \text{op} \)

Destination vector

\[
\begin{align*}
a_3 \text{ op } b_3 & \quad a_2 \text{ op } b_2 & \quad a_1 \text{ op } b_1 & \quad a_0 \text{ op } b_0 \\
\end{align*}
\]
First vector-capable supercomputer: Cray 1 (1976)
Today GPUs and FPGAs
Vector Units on CPUs, supporting special instruction sets
- MMX
- SSE
- AVX
- ARM NEON
Vector units achieve parallelism from one instruction.

This is described in terms of their width:
- 64-bit unit operates on 1 dp or 2 sp numbers
- 128-bit unit operates on 2 dp and 4 sp numbers

Data is loaded into a vector register and then processed by vector instructions.
SSE Instructions

128 bit

2x double
4x float
16x byte
8x short
4x integer32
2x integer64
SSE Instructions

• Arithmetic:
  • Multiply, Add, Subtract, Divide, Square root

• Logic:
  • and, and-not, or, xor

• Other:
  • Min/Max

• Dedicate functionality:
  • MPSADBW (Fast Block Difference), DPPS (Dot Product)
• Vector Data Types:
  • __m128 for single precision.
  • __m128i for integers.
  • __m128d for double precision

• example intrinsics:
  • __m128_mm_add_ps(__m128_A, __m128_B);
  • __m128_mm_mul_ps(__m128_A, __m128_B);
  • __m128_mm_and_pd(__m128_A, __m128_B);
  • __m128_mm_cmpeq_pd(__m128_A, __m128_B);
Memory access

- Vector data are stored in XMM registers
- Addresses have to be 16-byte-aligned
- Data can be loaded
  - _m128 a = _mm_load_ps(&vec_a[0]);
- And stored back
  - _mm_store_ps( (_m128*) a, vec_a );
double pi()
{
    double pi = 0.0;
    double t;

    #pragma omp simd private(t) reduction (+:pi)
    for (i=0; i<count; i++) {
        t=(double) ((i+0.5) / count);
        pi += 4.0 / (1.0+t*t);
    }
    pi /=count;
    return pi;
}
Auto-vectorization

- GCC support auto-vectorization
  - -ftree-vectorize -ftree-vectorizer-verbose=2

- DEMO
Most frequent reason: Data dependencies
  - Simplified: Loop iterations must be independent

Many other potential reasons
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not “countable”
    - E.g. upper bound not a run time constant
  - Mixed data types (many cases now handled successfully)
  - Non-unit stride between elements
  - Loop body too complex (register pressure)
Help from compilers

- Example: icc vectorization report

```fortran
35: subroutine fd(y)
36: integer :: i
37: real, dimension(10), intent(inout) :: y
38: do i=2,10
39:   y(i) = y(i-1) + 1
40: end do
41: end subroutine fd
```

novec.f90(38): (col. 3) remark: loop was not vectorized: existence of vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_: loop was not vectorized: existence of vector dependence
How to use them?

- Fully automatic vectorization
- Auto vectorization hints (#pragma ivdep)
- SIMD feature (#pragma simd and simd function annotation)
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (_mm_add_ps())
- ASM code (addps)

Ease of use

Programmer control
How to use them?

Optimized numerical libraries (BLAS, Intel MKL,..)

- Fully automatic vectorization
- Auto vectorization hints (#pragma ivdep)
- SIMD feature (#pragma simd and simd function annotation)
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (_mm_add_ps())
- ASM code (addps)

Ease of use

Programmer control
Example: Intel Xeon Phi / MIC
Different Instruction set. LRBNI (Larrabee New Instructions)

512-bit wide SIMD vector ops
  8 double (or 16 float / integer) operations per cycle

Vectorized Fused-Multiply-Add (FMA)

Vector masks

Gather / scatter operations
Summary

- Vectorization adds 4-16x SP speedup and 2-8x DP speedup depending on HW generation
- Auto-vectorization helps but often needs help
- Writing intrinsics leads to more efficient code, but at the cost of readability & flexibility
Questions?