Algorithms and Computation in Signal Processing

special topic course 18-799B
spring 2005
21th lecture Mar. 29, 2005

Instructor: Markus Pueschel
Guest instructor: Franz Franchetti
TA: Srinivas Chellappa

Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - State-of-the-art floating-point SIMD extensions
  - History and related technologies
  - How to use it
- **Writing code for Intel's SSE**
  - Instructions
  - Common building blocks
  - Examples: WHT, matrix multiplication, FFT
- **Selected topics**
  - BlueGene/L
  - Complex arithmetic and instruction-level parallelism
  - Things that don't work as expected
- **Conclusion: How to write good vector code**

Organization

- **Overview**
    - Idea, benefits, reasons, restrictions
    - State-of-the-art floating-point SIMD extensions
    - History and related technologies
    - How to use it
- **Writing code for Intel's SSE**
    - Instructions
    - Common building blocks
    - Examples: WHT, matrix multiplication, FFT
- **Selected topics**
    - BlueGene/L
    - Complex arithmetic and instruction-level parallelism
    - Things that don't work as expected
- **Conclusion: How to write good vector code**

SIMD (Signal Instruction Multiple Data) vector instructions in a nutshell

- ## What are these instructions?
  - Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats

   +   **x**   **4-way**
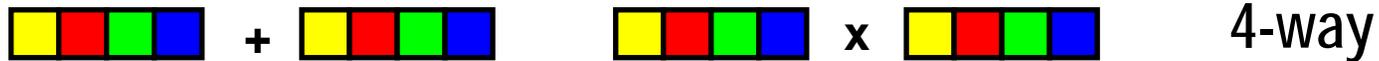
- ## Why are they here?
  - **Useful:** Many applications (e.g.,multi media) feature the required fine grain parallelism – code potentially faster
  - **Doable:** Chip designers have enough transistors available, easy to implement

Overview Vector ISAs

Vendor	Name	ν-way	Precision	Processor
Intel	SSE	4-way	single	Pentium III Pentium 4
Intel	SSE2	2-way	double	Pentium 4
Intel	SSE3	4-way 2-way	single double	Pentium 4
Intel	IPF	2-way	single	Itanium Itanium 2
AMD	3DNow!	2-way	single	K6
AMD	Enhanced 3DNow!	2-way	single	K7, Athlon XP Athlon MP
AMD	3DNow! Professional	4-way	single	Athlon XP Athlon MP
AMD	AMD64	2-way 4-way 2-way	single single double	Athlon 64 Opteron
Motorola	AltiVec	4-way	single	MPC 74xx G4
IBM	AltiVec	4-way	single	PowerPC 970 G5
IBM	Double FPU	2-way	double	PowerPC 440 FP2

Evolution of Intel Vector Instructions

- **MMX (1996, Pentium)**
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - MMX register = Float register
  - Lost importance due to SSE2 and modern graphics cards
- **SSE (1999, Pentium III)**
  - Superset of MMX
  - 4-way float operations, single precision
  - 8 new 128 Bit Register
  - 100+ instructions
- **SSE2 (2001, Pentium 4)**
  - Superset of SSE
  - "MMX" operating on SSE registers, 2 x 64
  - 2-way float ops, double-precision, same registers as 4-way single-precision
- **SSE3 (2004, Pentium 4E Prescott)**
  - Superset of SSE2
  - New 2-way and 4-way vector instructions for complex arithmetic

Related Technologies

- **Original SIMD machines (CM-2,…)**
  - Don't really have anything in common with SIMD vector extension

- **Vector Computers (NEC SX6, Earth simulator)**
  - Vector lengths of up to 128
  - High bandwidth memory, no memory hierarchy
  - Pipelined vector operations
  - Support strided memory access

- **Very long instruction word (VLIW) architectures (Itanium,…)**
  - Explicit parallelism
  - More flexible
  - No data reorganization necessary

- **Superscalar processors (x86, …)**
  - No explicit parallelism
  - Memory hierarchy

SIMD vector extensions borrow multiple concepts

How to use SIMD Vector Extensions?

- **Prerequisite: fine grain parallelism**

- **Helpful: regular algorithm structure**

- **Easiest way: use existing libraries**
  Intel MKL and IPP, Apple vDSP, AMD ACML,
  Atlas, FFTW, Spiral

- **Do it yourself:**
  - **Use compiler vectorization: write vectorizable code**
  - **Use language extensions to explicitly issue the instructions**
    Vector data types and intrinsic/builtin functions
    Intel C++ compiler, GNU C compiler, IBM VisualAge for BG/L,…
  - Implement kernels using assembly (inline or coding of full modules)

Characterization of Available Methods

- **Interface used**
    - Portable high-level language (possibly with pragmas)
    - Proprietary language extension (builtin functions and data types)
    - Assembly language
- **Who vectorizes**
    - Programmer or code generator expresses parallelism
    - Vectorizing compiler extracts parallelism
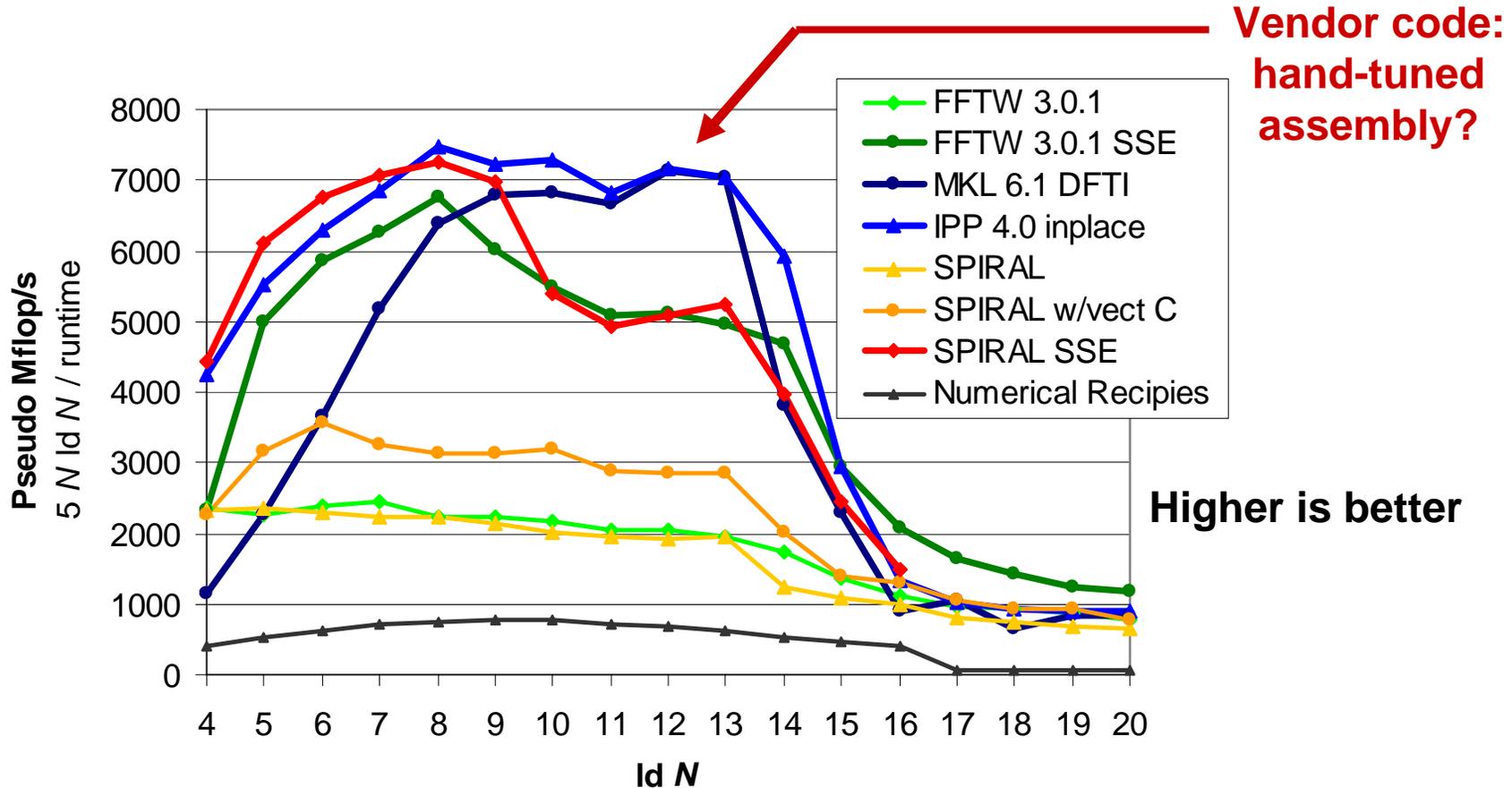- **Structures vectorized**
    - Vectorization of independent loops
    - Instruction-level parallelism extraction
- **Generality of approach**
    - General purpose (e.g., for complex code or for loops)
    - Problem specific (for FFTs or for matrix products)

Benchmark: DFT, 2-powers

P4, 3.0 GHz, icc 8.0

Vendor code: hand-tuned assembly?

Higher is better

Legend:
- FFTW 3.0.1
- FFTW 3.0.1 SSE
- MKL 6.1 DFTI
- IPP 4.0 inplace
- SPIRAL
- SPIRAL w/vect C
- SPIRAL SSE
- Numerical Recipies

Y-axis: **Pseudo Mflop/s** $5\,N\,\mathrm{ld}\,N\,/\,\text{runtime}$

X-axis: **ld N**

Single precision code

- **limitations of compiler vectorization**
- **Spiral code competitive with the best**

Problems

- Correct data alignment paramount

- Reordering data kills runtime

- Algorithms must be adapted to suit machine needs

- Adaptation and optimization is machine/extension dependent

- Thorough understanding of ISA + Micro architecture required

One can easily slow down a program by vectorizing it

Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - State-of-the-art floating-point SIMD extensions
  - History and related technologies
  - How to use it
- **Writing code for Intel's SSE**
  - Instructions
  - Common building blocks
  - Examples: WHT, matrix multiplication, FFT
- **Selected topics**
  - BlueGene/L
  - Complex arithmetic and instruction-level parallelism
  - Things that don't work as expected
- **Conclusion: How to write good vector code**

Intel Streaming SIMD Extension (SSE)

- **Used syntax: Intel C++ compiler**
  - Data type: `__m128 d; // ={float d3, d2, d1, d0}`
  - Intrinsics: `_mm_add_ps(), _mm_mul_ps(),…`
  - Dynamic memory: `_mm_malloc(), _mm_free()`

- **Instruction classes**
  - Memory access (explicit and implicit)
  - Basic arithmetic (+, -, *)
  - Expensive arithmetic (1/x, sqrt(x), min, max, /, 1/sqrt)
  - Logic (and, or, xor, nand)
  - Comparison (+, <, >, …)
  - Data reorder (shuffling)

Blackboard