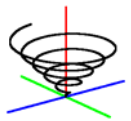
Program Generation and Optimization

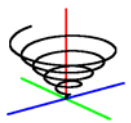
CS 860
summer 2005
July 26, 2005

Instructor: Jeremy Johnson
Guest instructor: Franz Franchetti



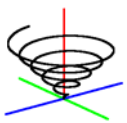
Organization

- Short Vector SIMD Extensions
  - 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



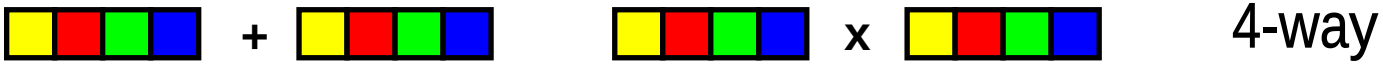
Organization

- 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

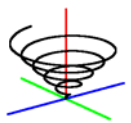
- What are these instructions?
  - Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats



- 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

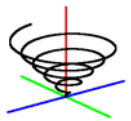
Overview Vector ISAs

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

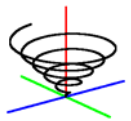
- MMX (1996, Pentium)
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - MMX register = Float register
  - Lost importance due to SSE2 and modern graphics cards
- SSE (1999, Pentium III)
  - Superset of MMX
  - 4-way float operations, single precision
  - 8 new 128 Bit Register
  - 100+ instructions
- 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

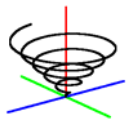
- Original SIMD machines (CM-2,…)
  - Don't really have anything in common with SIMD vector extension
- Vector Computers (NEC SX6, Earth simulator)
  - Vector lengths of up to 128
  - High bandwidth memory, no memory hierarchy
  - Pipelined vector operations
  - Support strided memory access
- Very long 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)



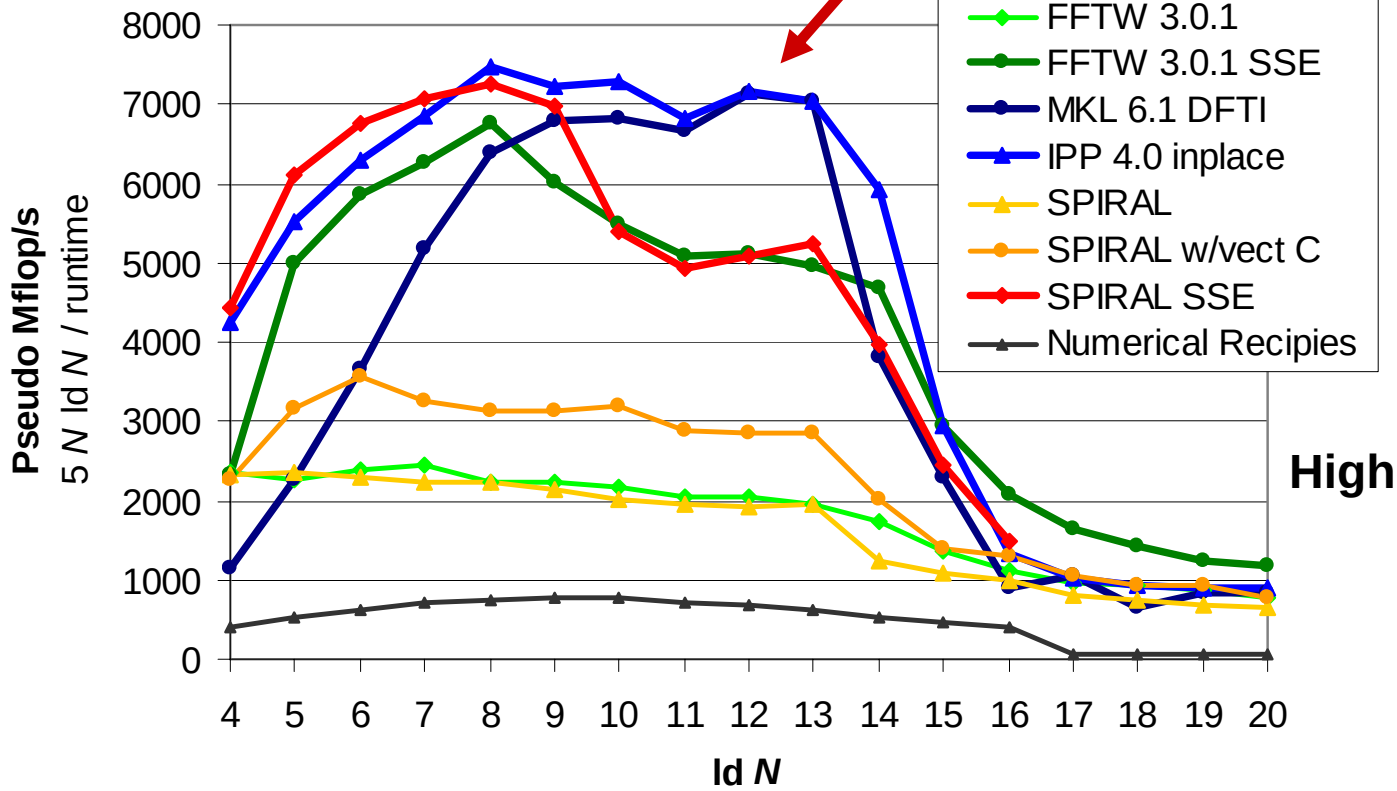
Characterization of Available Methods

- 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

Single precision code

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

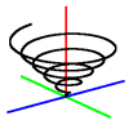
SPIRAL

Drexel UNIVERSITY

Problems

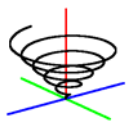
- Correct data alignment paramount
- Reordering data kills runtime
- Algorithms must be adapted to suit machine needs
- Adaptation and optimization is machine/extension dependent
- Thorough understanding of ISA + Micro architecture required

One can easily slow down a program by vectorizing it



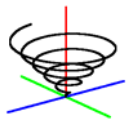
Organization

- 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

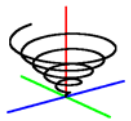


Intel Streaming SIMD Extension (SSE)

- 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)

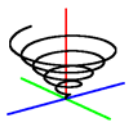


Blackboard

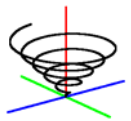


Organization

- 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

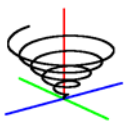


Blackboard



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



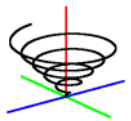
BlueGene/L Supercomputer

System at Lawrence Livermore National Laboratory (LLNL)

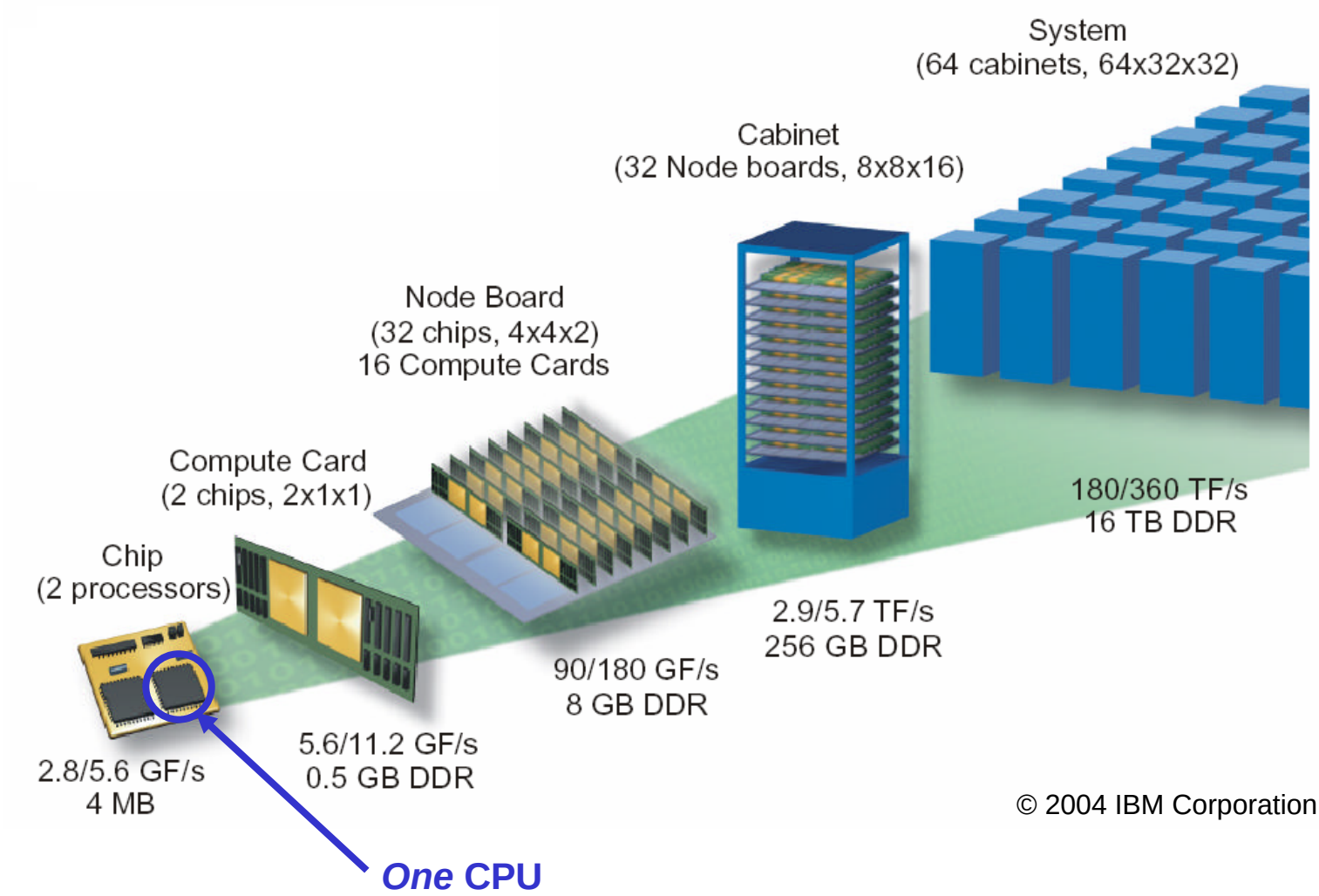
- Aims at #1 in Top 500 list of supercomputers
- 65,536 processors
  PowerPC 440 FP2 @ 700 MHz
- 360 Tflop/s peak performance
- 16 TByte RAM
- In operation by end of 2005

Smaller systems will be commercially available

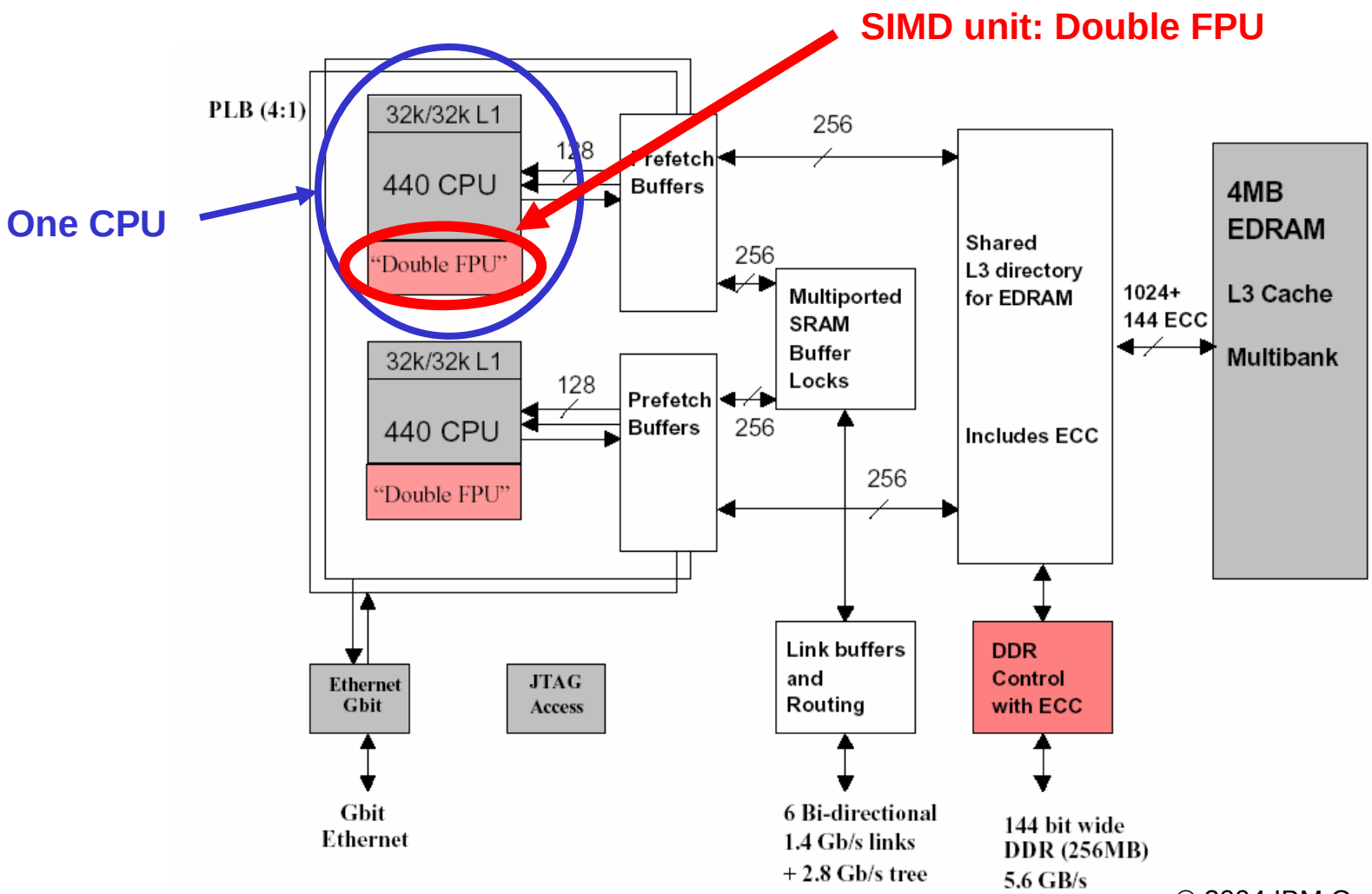
- Other national labs, universities, Japan, Germany,…
- BlueGene/L consortium: open to everybody, community effort



The BlueGene/L System at LLNL



BlueGene/L CPU: PowerPC 440 FP2



SPIRAL

Drexel UNIVERSITY

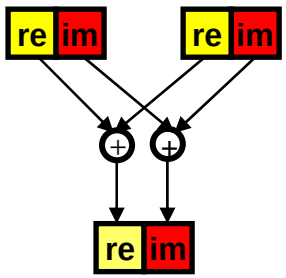
The Double FPU

BlueGene/L Double FPU: Two coupled FPUs

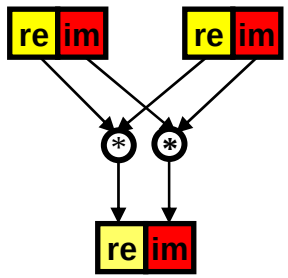
- Scalar and two-way vector FPU instructions
- Per cycle: Either two-way FMA or two-way move, and one two-way load or store
- Double precision

Supports complex arithmetic and two-way SIMD

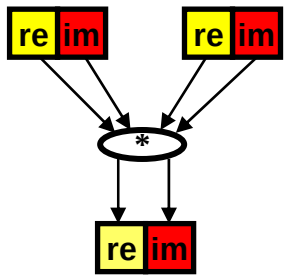
- 20 instructions supporting complex multiply-add
- Implicit parallel, cross and copy operations
- Vector sign changes and cross moves



Parallel add = 1 instr.
Complex add



Parallel mul = 1 instr.



Complex mul = 2 instr. (6 flops)

SPIRAL

Drexel UNIVERSITY

Vectorization Overhead

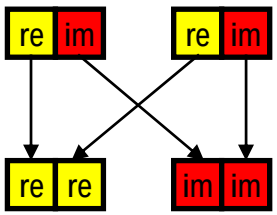
Complex arithmetic

- Native mode for BlueGene/L Double FPU
- However, many codes use real arithmetic
- Real codes require vectorization

Real vector code = faster computation but overhead

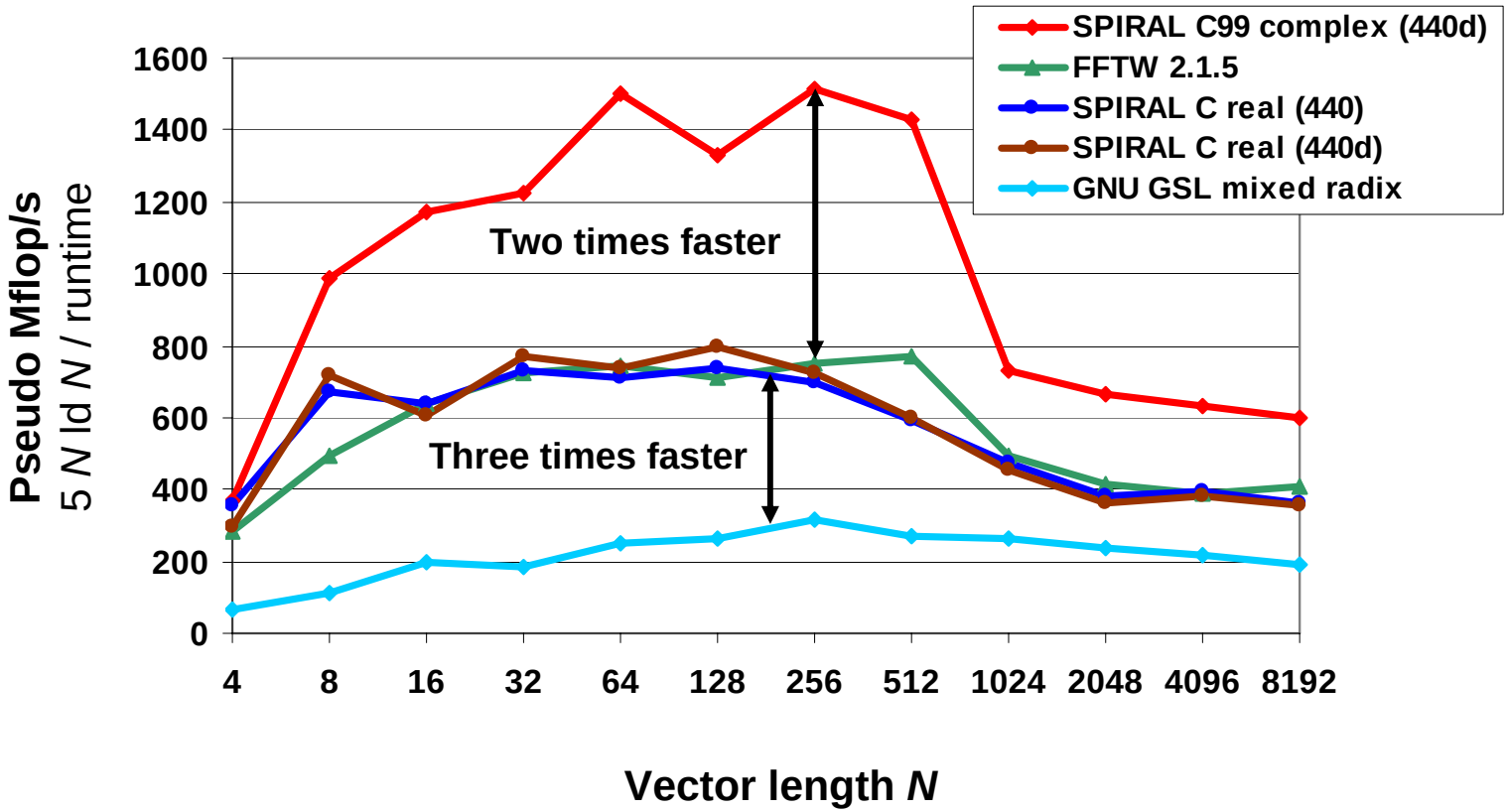
- Overhead: prepare data for parallel computation
- Goal: minimize or eliminate these reorder operations

BlueGene/L: Expensive data reorganization



- Work in parallel on real and imaginary parts
- One copy and two cross-copies
  On BlueGene/L: 3 cycles = 12 flops

Benchmark: DFT, 2-powers, BlueGene/L



BlueGene/L DD2 prototype at IBM T.J. Watson Research Center
Single BlueGene/L CPU at 700 MHz (one Double FPU), IBM XL C compiler

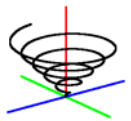
- **Utilization of complex FPU via C99 _Complex double**
- **Factor 2 over real code with compiler vectorization (IBM XL C)**

SPIRAL

Drexel UNIVERSITY

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



Example: Complex Multiplication SSE3

Complex C99 code + compiler vectorization
works reasonably well

Complex code features intrinsic
2-way vector parallelism

The Corresponding Assembly Code

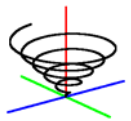
SSE3:

```
movapd   xmm0, XMMWORD PTR A
movddup  xmm2, QWORD PTR   B
mulpd    xmm2, xmm0
movddup  xmm1, QWORD PTR   B+8
shufpd   xmm0, xmm0, 1
mulpd    xmm1, xmm0
addsubpd xmm2, xmm1
movapd   XMMWORD PTR C, xmm2
```

SSE2:

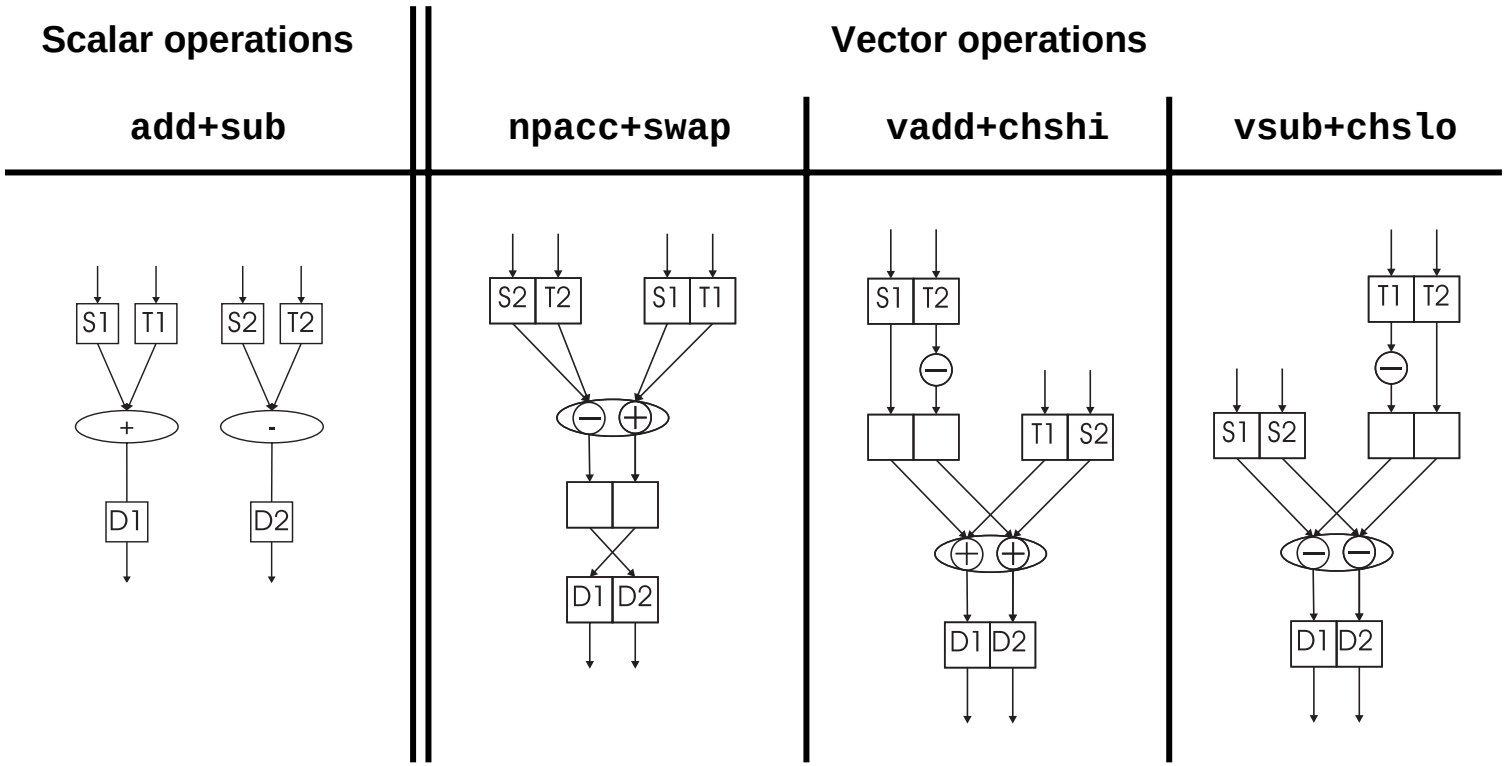
```
movsd    xmm3, QWORD PTR A
movapd   xmm4, xmm3
movsd    xmm5, QWORD PTR A+8
movapd   xmm0, xmm5
movsd    xmm1, QWORD PTR B
mulsd    xmm4, xmm1
mulsd    xmm5, xmm1
movsd    xmm2, QWORD PTR B+8
mulsd    xmm0, xmm2
mulsd    xmm3, xmm2
subsd    xmm4, xmm0
movsd    QWORD PTR C, xmm4
addsd    xmm5, xmm3
movsd    QWORD PTR C, xmm5
```

In SSE2 scalar code is better



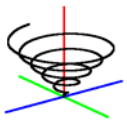
Example: 3DNow! Basic Block Vectorization

- Utilizing instruction-level parallelism
- Inter-operand and intra-operand vector instructions



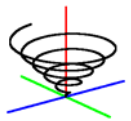
Organization

- 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



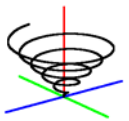
Things that don't work as expected

- Intel SSE/SSE2/SSE3
  - SSE2 can't do complex arithmetic well
  - Early application notes showed really bad code examples (split radix FFT)
  - Intel Compiler doesn't vectorize despite pragmas,…
- Intel Itanium processor family (IPF)
  - No intrinsic interface to IPF native vector instruction
  - Can only use 4-way SSE intrinsics to program 2-way IPF
  - With Itanium 2, no vectorization speed-up possible any more
- AMD 3DNow! and AMD64
  - AMD64 can do 3DNow! and SSE2 in parallel – have fun!
  - For a long time they had no compiler support
  - K7: One intra operand instruction is just missing (++,+-, --; -+??)



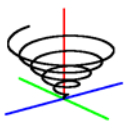
Things that don't work as expected (2)

- Motorola/IBM AltiVec
  - No unaligned memory access (raises exception)
  - Subvector access: the actually read/written vector element depends on the memory address referenced (!!)
  - A general shuffle requires a 128 bit register "howto" operand
  - Only fused-multiply-add (FMA) instruction – have to add explicitly (0,0,0,0) for multiplication only
  - For a while, the GNU C compiler was buggy and the only compiler available
- IBM Double FPU (BlueGene/L)
  - One shuffle *or* one vector FMA per cycle
  - Data reorganization prohibitively expensive
  - Have to fold that into special FMAs and multiply by one



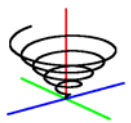
Organization

- 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



How to Write Good Vector Code?

- Take the “right” algorithm and the “right” data structures
  - Fine grain parallelism
  - Correct alignment in memory
  - Contiguous arrays
- Use a good compiler (e. g., vendor compiler)
- First: Try compiler vectorization
  - Right options, pragmas and dynamic memory functions
    (Inform compiler about data alignment, loop independence,…)
  - Check generated assembly code *and* runtime
- If necessary: Write vector code yourself
  - Most expensive subroutine first
  - Use intrinsics, no (inline) assembly
  - Important: Understand the ISA



Remaining time: Discussion