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

\[
\begin{array}{ccc}
\text{red} & \text{green} & \text{blue} \\
\text{yellow} & \text{green} & \text{blue} \\
\end{array}
\quad + \\
\begin{array}{ccc}
\text{red} & \text{green} & \text{blue} \\
\text{yellow} & \text{green} & \text{blue} \\
\end{array}
\quad \times \\
\begin{array}{ccc}
\text{red} & \text{green} & \text{blue} \\
\text{yellow} & \text{green} & \text{blue} \\
\end{array}
\quad 4\text{-way}
\]

■ Why are they here?
  ▪ **Useful:** Many applications (e.g., multimedia) feature the required fine grain parallelism – code potentially faster
  ▪ **Doable:** Chip designers have enough transistors available, easy to implement
## Overview Vector ISAs

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Name</th>
<th>$n$-way</th>
<th>Precision</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>SSE</td>
<td>4-way</td>
<td>single</td>
<td>Pentium III</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pentium 4</td>
</tr>
<tr>
<td>Intel</td>
<td>SSE2</td>
<td>2-way</td>
<td>double</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>Intel</td>
<td>SSE3</td>
<td>4-way</td>
<td>single</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-way</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td>IPF</td>
<td>2-way</td>
<td>single</td>
<td>Itanium</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow!</td>
<td>2-way</td>
<td>single</td>
<td>K6</td>
</tr>
<tr>
<td>AMD</td>
<td>Enhanced 3DNow!</td>
<td>2-way</td>
<td>single</td>
<td>K7, Athlon XP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Athlon MP</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow! Professional</td>
<td>4-way</td>
<td>single</td>
<td>Athlon XP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Athlon MP</td>
</tr>
<tr>
<td>AMD</td>
<td>AMD64</td>
<td>2-way</td>
<td>single</td>
<td>Athlon 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-way</td>
<td>single</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-way</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>Motorola</td>
<td>Altivec</td>
<td>4-way</td>
<td>single</td>
<td>MPC 74xx G4</td>
</tr>
<tr>
<td>IBM</td>
<td>Altivec</td>
<td>4-way</td>
<td>single</td>
<td>PowerPC 970 G5</td>
</tr>
<tr>
<td>IBM</td>
<td>Double FPU</td>
<td>2-way</td>
<td>double</td>
<td>PowerPC 440 FP2</td>
</tr>
</tbody>
</table>
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

Vendor code: hand-tuned assembly?

Higher is better

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
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

System
(64 cabinets, 64x32x32)

Cabinet
(32 Node boards, 8x8x16)

Node Board
(32 chips, 4x4x2)
16 Compute Cards

Compute Card
(2 chips, 2x1x1)

Chip
(2 processors)

2.8/5.6 GF/s 4 MB
5.6/11.2 GF/s 8 GB DDR
2.9/5.7 TF/s 256 GB DDR

180/360 TF/s 16 TB DDR

One CPU

© 2004 IBM Corporation
BlueGene/L CPU: PowerPC 440 FP2

One CPU

SIMD unit: Double FPU

32k/32k L1

440 CPU

"Double FPU"

32k/32k L1

440 CPU

"Double FPU"

PLB (4:1)

32k/32k L1

440 CPU

"Double FPU"

Prefetch Buffers

128

256

Multiported SRAM Buffer Locks

32k/32k L1

440 CPU

"Double FPU"

Prefetch Buffers

128

256

256

256

Shared L3 directory for EDRAM

Includes ECC

4MB EDRAM

L3 Cache

Multibank

Ethernet Gbit

Gbit Ethernet

JTAG Access

Link buffers and Routing

6 Bi-directional 1.4 Gb/s links
+ 2.8 Gb/s tree

144 bit wide DDR (256MB)
5.6 GB/s

1024+ 144 ECC

© 2004 IBM Corporation
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)
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)
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
movapd     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

Scalar operations

<table>
<thead>
<tr>
<th></th>
<th>add+sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>T1</td>
</tr>
<tr>
<td>S2</td>
<td>T2</td>
</tr>
</tbody>
</table>

Vector operations

<table>
<thead>
<tr>
<th></th>
<th>npacc+swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>T2</td>
</tr>
<tr>
<td>S1</td>
<td>T1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>vadd+chshi</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>T2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>vsub+chslo</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
</tr>
</tbody>
</table>

Diagram:

- S1, S2, T1, T2, D1, D2 represent variables and data elements.
- The operations are visualized with arrows connecting the variables and data elements.
- The diagram illustrates the flow of data and operations for scalar and 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