Vector Processing

Andrew Lenharth
UT Austin - CS 378
9/2014

Some slides adapted from Maria Garzaran

Instructon Level Parallelism
• High speed execution based on *instruction level parallelism* (ilp): potential of short instruction sequences to execute in parallel
• High-speed microprocessors exploit ILP by:
  1) pipelined execution: overlap instructions
  2) superscalar execution: issue and execute multiple instructions per clock cycle
  3) Out-of-order execution (commit in-order)
• Memory accesses for high-speed microprocessor?
  – Data Cache, possibly multiported, multiple levels

Theoretical Limits to ILP?

(Figure 4.48, Page 332)

Limits to using ILP
1) *pipelined clock rate*: deeper pipelines have CPI increase (branches, other hazards)
2) *instruction fetch and decode*: fetch bandwidth and decode complexity
3) *cache hit rate*: Cache misses stall processor

Vector Processing
• Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

Types of Vector Machines

Register
• Fixed length vector register
• Looks like RISC, acts like RISC
• More flexibility for SW
  – Overlap vector ops
  – Software pipelining
• Less flexibility for SW
  – Must match vector length
  – Alignments restrictions
  – Limited load options

Memory-Memory
• Instructions specify base and length
• Very Concise
• Works with arbitrary vector length
• Often slower than scalar for small lengths
• Often flexible address generation

* Adapted from David Patterson
Single Instruction Multiple Data

- Recall ideal machine study: Case for multiple operations, especially floating point
- Short vector registers (128-512 bit)
  - Series of floats/doubles/ints
  - Often interpretation left to instruction
- 3 form instructions (usually)
  - \( X = A \text{ op } B \)
- Pairwise Execution (most operations)

Example code

```c
for (i = 0; i < n; ++i) c[i] = a[i] + b[i];
```

**SIMD**

- Intel SSE and IBM Altivec have 128-bit vector registers and functional units
  - 4 32-bit single precision floating point numbers
  - 2 64-bit double precision floating point numbers
  - 4 32-bit integer numbers
  - 2 64-bit integer
  - 8 16-bit integer or shorts
  - 16 8-bit bytes or chars
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.
- Newer processors, such as Sandy or Ivy Bridge have AVX that support 256-bit vector registers. Xeon PHI is 512-bit

**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 (Titanium,...)
  - Explicit parallelism
  - More flexible
  - No data reorganization necessary
- Superscalar processors (x86, ...)
  - No explicit parallelism
  - Memory hierarchy

**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 B6/8,...
  - Implement kernels using assembly (inline or coding of full modules)

* Adapted from Fran Franchetti
Characterization of Available Methods

- **Interface used**
  - Portable high-level language (possibly with pragmas)
  - Proprietary language extension (builtin functions and data types)
  - C++ Class interface
  - 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)

*Adapted from Fran Franchetti*

How do we access the SIMD units?

**Vectorizable code**

```c
for (i=0; i<LEN; i++)
    c[i] = a[i] + b[i];
```

**Assembly**

```assembly
B8.5
movaps a[%,rdx,4], %xmm0
addps b[%,rdx,4], %xmm0
```

**Intrinsics**

```c
void example() {
    __m128 rA, rB, rC;
    for (int i = 0; i < LEN; i++){
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC = _mm_add_ps(rA, rB);
        _mm_store_ps(&C[i], rC);
    }
}
```

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*

*Adapted from Fran Franchetti*

Why use compiler vectorization?

- Easier
- Portable across vendors and machines
  - Except when it isn’t
- Other optimizations

Compiler Failure

- Data dependencies exist
  - Vectorization will change meaning
- Unprovable Properties
  - Aliasing
  - Alignment
- Doesn’t seem profitable
  - Based on compiler’s heuristics
  - Programmer can override
  - Programmer can change code

*Example*

```c
void test(float* A, float* B, float* C, float* D, float* E) {
    for (int x = 0; x < LEN; ++x)
}
```

What dependencies exist?
Revised Example

void test(float* restrict A, float* restrict B, float* restrict C, float* restrict D, float* restrict E) {
    for (int x = 0; x < LEN; ++x)
}

• Now what dependencies exist?
• restrict new c99 keyword, ways only it or values derived from it will access target object
• You are claiming something is true, the compiler doesn’t have to check
  – You might be lying
• Compiler should now vectorize (2.5x)

Loop Vectorization – Step 1

for (i=0; i<LEN; i++){
    a[i]=b[i]+(float)1.0;
    c[i]=b[i]+(float)2.0;
}

Compute dependencies: NONE

Loop Vectorization – Step 2

for (i=0; i<LEN; i+=strip_size){
    for(j = i; j < i + strip_size; ++j {
        a[i]=b[i]+(float)1.0;
        c[i]=b[i]+(float)2.0;
    }
}

Strip-mine by vector size

Loop Vectorization – Step 4

for (i=0; i<LEN; i+=strip_size){
    V = vector_load(&b[i]);
    V2 = V + <1.0,...,1.0>;
    vector_store(&a[i], V);
    V = vector_load(&b[i]);
    V3 = V + <2.0,...,2.0>;
    vector_store(&b[i], V3);
}

Compiler will later remove redundant load

Match short loops with vector ops

Legality

• Legal when:
  – Only forward dependencies
  – Dependence distance is greater than vector length
  – Some types of cycles
Forward Dependence

for (i=0; i<LEN; i++) {
    a[i] = b[i] + c[i]
    d[i] = a[i] + (float) 1.0;
}

Backward Dependence

for (i=0; i<LEN; i++) {
    a[i] = b[i] + c[i]
    d[i] = a[i+1] + (float) 1.0;
}

Backwards Dependence -- Reordered

for (i=0; i<LEN; i++) {
    d[i] = a[i+1] + (float) 1.0;
    a[i] = b[i] + c[i];
}

Cycles

Self-antidependence
for (int i=0; i<LEN-1; i++) {
    a[i+1] = a[i+1] + b[i];
}

Self true-dependence
for (int i=1; i<LEN; i++) {
    a[i+1] = a[i+1] + b[i];
}

“Long” cycles

for (int i=4; i<LEN; i++) {
    a[i] = a[i-4] + b[i];
}

a[x] read 4 iterations after it is written, thus for vector length 4, a[0..3] read at iter 4..7 after iter 0..3 wrote a[0..3], so vectorization safe

Corollary: SIMD vectorization legal in some cases full vectorization not legal

Outline

• Introduction
• Data Dependencies
• Data Alignment
• Aliasing
• Non-unit Strides
• Conditional Statements
• Intrinsics
Data Alignment

• Vector loads/stores load/store 128 consecutive bits to a vector register.
• Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
  — Intel platforms support aligned and unaligned load/stores
  — IBM platforms do not support unaligned load/stores

```
void test1(float *a, float *b, float *c) {
  for (int i=0;i<LEN;i++)
    a[i] = b[i] + c[i];
}
```

Computing Data Alignment

• 16-byte aligned = address is a multiple of 16
• 16 = 10000b = 0x10, thus last 4 bits are 0
• If ptr & 0x0F is zero, ptr is 16-byte aligned

```
void test(float *a, float *b, float *c, int LEN) {
  if (a & 0x0F == b & 0x0F == c & 0x0F)
    for (int x = 0; x < LEN; ++x)
      a[x] = b[x] + c[x];
  else {
    vector code;
  }
}
```

Data Alignment for Simple Type

• Primitive data types usually aligned to native type size
  — But not always, Linux (x86, 32bit) long int is 8 bytes but 4 byte aligned (8 byte aligned on 64 bit)
• Arrays packed to tightest alignment
  — sizeof(float) == alignment_of(float) == 4
  — 4 floats = 16 bytes, array of 4 floats = 16 bytes

```
struct {
  char a;
  long b;
  char c;
}
```

Alignments and Sizes

<table>
<thead>
<tr>
<th></th>
<th>Linux (Windows), x86</th>
<th>Linux (Windows), amd64</th>
<th>Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>alignment</td>
<td>size</td>
<td>alignment</td>
</tr>
<tr>
<td>bool</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16 (B)</td>
<td>16 (8)</td>
</tr>
<tr>
<td>void*</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Composing Alignment

• Structures are layed out in memory according to member alignment
  — Assume struct starts at 0 and all elements aligned, “padding” fills in the gaps
  — Padding at end so each struct in an array is aligned

```
struct {
  char a;
  padding N/A 1 3 4
  long b 4 4 8
  char c 8 1 9
  padding N/A 9 3 12
}
```

Compiler Assumptions

• Often cannot know alignment statically
• May be dependent on call site
• 2-version code, loop pealing

```
void test(float *a, float *b, float *c, int LEN) {
  for (int x = 0; x < LEN; ++x)
    a[x] = b[x] + c[x];
}  
```
Lying Asserting Alignment

• __assume_aligned (intel)
• __builtin_assume_aligned (gcc)

```
void func1(float *a, float *b, float *c) {
    __assume_aligned(a, 16);
    __assume_aligned(b, 16);
    __assume_aligned(c, 16);
    for (int i = 0; i < LEN; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Specifying alignment

• Variables (including globals):
  – __attribute__((aligned(16))) float b[N];
• Heap:
  – new[] and malloc return memory conservatively aligned for "standard" types
    • Vector types are not standard
    • Alignment of malloc is ABI concern
      • Common case allocations small, higher alignment = wasted space
  – float* a = (float*) memalign(16, N*sizeof(float));

Example

```
void func1(float *a, float *b, float *c) {
    float* aa = __builtin_assume_aligned(a, 16);
    float* bb = __builtin_assume_aligned(b, 16);
    float* cc = __builtin_assume_aligned(c, 16);
    for (int i = 0; i < LEN; i++) {
        aa[i] = bb[i] + cc[i];
    }
}
```

Vector in Struct

• Specify alignment manually
  • Don’t malloc array!

```
struct st {
    char A;
    int B[64];
    float C;
    int D[64];
};
```

Outline

• Introduction
• Data Dependencies
• Data Alignment
  • Aliasing
• Non-unit Strides
• Conditional Statements
• Intrinsics

Aliasing

• Can the compiler vectorize this loop?
  – Assume alignment

```
void func1(float *a, float *b, float *c) {
    for (int i = 0; i < LEN; i++) {
        a[i] = b[i] + c[i];
    }
}
```
Aliasing

- What if: `float* a = &b[0]`
- Or `float* a = &b[1]`

```c
void func1(float* a, float* b, float* c){
  for (int i = 0; i < LEN; i++) {
    a[i] = b[i] + c[i];
  }
}
```

To vectorize, must know pointers don’t alias

Could check… How many?

Solution 1: Globals

- Globals referenced by name are sufficient
- Not flexible, poor SW engineering

```c
__attribute__((aligned(16))) float a[LEN];
__attribute__((aligned(16))) float b[LEN];
__attribute__((aligned(16))) float c[LEN];
void func1(){
  for (int i=0; i<LEN; i++) {
    a[i] = b[i] + c[i];
  }
}
```

Meaning: restrict

- All accesses to object pointed to by restrict pointer are derived from restrict pointer

```c
float* restrict p = a;
a[3] ... // BAD
p[7] ... // Good
```

Solution 2: restrict

- `__restrict__` keyword (gcc/C++)
- restrict keyword (C99, not c++)
- How do you know?

```c
void func1(float* restrict a, float* b, float* c){
  for (int i=0; i<LEN; i++) {
    a[i] = b[i] + c[i];
  }
}
```

Non-unit stride

- Array of structs

```c
typedef struct{
  int x, y, z
} point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```

```c
<table>
<thead>
<tr>
<th>x0</th>
<th>y0</th>
<th>z0</th>
<th>x1</th>
<th>y1</th>
<th>z1</th>
<th>x2</th>
<th>y2</th>
<th>z2</th>
<th>x3</th>
<th>y3</th>
<th>z3</th>
</tr>
</thead>
</table>
```
Non-unit stride

- Array of structs

```c
typedef struct {int x, y, z} point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
    pt[i].y *= scale;
}
```

Masked loads and shuffles

```c
for (int i=0; i<LEN; i+=3) {
    vector a = masked load pt[i];
    b = masked load pt[i+1];
    c = masked load pt[i+1].y
    d = masked load pt[2].z;
    v = shuffle a,b,c,d appropriately and merge
    v *= scale;
    copy out, shuffle, use masked stores;
}
```

Pad, Gather/Scatter

```c
typedef struct {int x, y, z, pad} point;
point pt[LEN];
for (int i=0; i<LEN; i+=4) {
    vector v = gather(pt[i], 1, 3); //addr, offset, stride
    v *= scale;
    scatter(v, pt[i], 1, 3); //val, addr, offset, stride
}
```

Non-unit stride

- Array of structs

```c
typedef struct {int x, y, z} point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
    pt[i].y *= scale;
}
```

Gather/Scatter

```c
typedef struct {int x, y, z, pad} point;
point pt[LEN];
for (int i=0; i<LEN; i+=4) {
    vector v = gather(pt[i], 1, 4); //addr, offset, stride
    v *= scale;
    scatter(v, pt[i], 1, 4); //val, addr, offset, stride
}
```

- Shuffling data around is expensive, more expensive than doing operations serially
- Scatter/Gather help, but not panacea
- Design data structures to avoid problem
Struct of arrays

```c
struct point {
    int x[LEN];
    int y[LEN];
    int z[LEN];
}
```

```c
for (int i=0; i<LEN; i++) {
    pt.y[i] *= scale;
    // trivially vectorizable
}
```

Conditional Statements

- Vector instructions do the same thing to every data item
- Conditionals change what happens to each data item

```c
#pragma vector always
for (int i=0; i<LEN; i++) {
    if (c[i] < (float) 0.0) {
        a[i] = a[i] * b[i] + d[i];
    }
}
```

If-conversion

- do both sides of the condition unconditionally
- conditionally move result
  - CMOV (x86)
- Allows branch free code
  - at cost of doing more work
  - not always safe

```c
for (int i = 0; i < LEN; i++) {
    t = a[i] * b[i] + d[i];
    a[i] = c[i] < 0 ? t : a[i];
}
```

Outline

- Introduction
- Data Dependencies
- Data Alignment
- Aliasing
- Non-unit Strides
- Conditional Statements
- Intrinsics

Vector conditional: masks

```c
for (int i=0;i<1024;i++){
    if (c[i] < (float) 0.0) a[i]=a[i]*b[i]+d[i];
}
```

Access SIMD through intrinsics

- Intrinsics are vendor/architecture specific
  - We will focus on the Intel vector intrinsics
  - gcc attempts to have platform-independent vector intrinsics
- Intrinsics are useful when
  - the compiler fails to vectorize
  - when the programmer thinks it is possible to generate better code than the one produced by the compiler
Intel SSE Headers
- SSE can be accessed using intrinsics.
- You must use one of the following header files:
  - `#include <xmmintrin.h>` (for SSE)
  - `#include <emmintrin.h>` (for SSE2)
  - `#include <pmmintrin.h>` (for SSE3)
  - `#include <smmintrin.h>` (for SSE4)
- These include the prototypes of the intrinsics.

Intel SSE Data Types
- We will use the following data types:
  - `__m128` packed single precision (XMM register)
  - `__m128d` packed double precision (XMM register)
  - `__m128i` packed integer (XMM register)

#include <xmmintrin.h>
int main () {
  ...
  __m128 A, B, C; /* three packed s.p. variables */
  ...
}

Intel SSE Intrinsics Instructions
- Intrinsics operate on these types and have the format:
  - `__mm_instruction_suffix(...)
- Suffix can take many forms. Among them:
  - `ss` scalar single precision
  - `ps` packed (vector) single precision
  - `sd` scalar double precision
  - `pd` packed double precision
  - `si#` scalar integer (8, 16, 32, 64, 128 bits)
  - `su#` scalar unsigned integer (8, 16, 32, 64, 128 bits)

Intel SSE Examples
- Load four 16-byte aligned single precision values in a vector:
  float a[4]={1.0,2.0,3.0,4.0}; //a must be 16-byte aligned
  __m128 x = _mm_load_ps(a);

- Add two vectors containing for floats:
  __m128 a, b;
  __m128 c = _mm_add_ps(a, b);

SSE Full example
```c
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16)))
float a[n], b[n], c[n];

int main() {
  for (i = 0; i < n; i++) {
    c[i] = a[i]*b[i];
  }
}
```
```c
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16)))
float a[n], b[n], c[n];

int main() {
  __m128 rA, rB, rC;
  for (i = 0; i < n; i+=4) {
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_mul_ps(rA, rB);
    _mm_store_ps(&c[i], rC);
  }
}
```