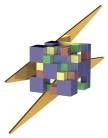
Lecture 24

Vectorization

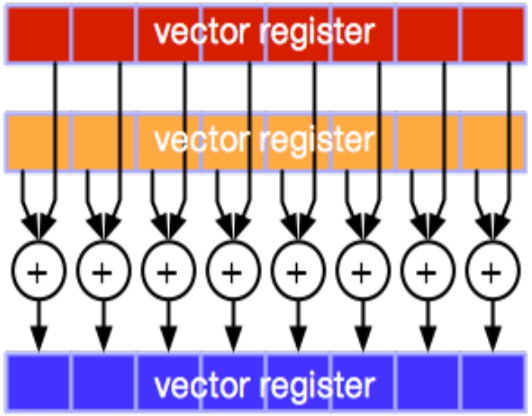
이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

SIMDization

- A special case of parallelization
  - Vectorization
- SIMD
  - Typically a single instruction to perform the same operation in parallel on multiple data elements of the same type and size
  - Broader sense: a single program counter (e.g., GPUs)
- Multiple parallel execution units are called lanes



SIMDization (cont'd)

- Compilers typically support auto-vectorization

- SIMD support
  - Intel's MMX, SSE, and AVX
  - Intel Xeon Phi's 512-bit wide vector
  - IBM PowerPC's AltiVec
  - ARM's NEON

Vector Supercomputer

- Cray-1, 1976
  - Scalar unit + Vector extensions

- Load/store architecture
- Scalar registers
- 64 64-bit vector registers
- Vector instructions
- Highly pipelined functional units
- Interleaved memory system
- No data caches
- No virtual memory

Multimedia Extensions

- Intel's MMX and SSE
- IBM PowerPC's AltiVec
- Limited vector instruction set
  - No vector length control
  - No strided load/store or scatter/gather
  - Unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length
  - Requires superscalar dispatch to keep multiply/add/load units busy
  - Loop unrolling to hide latencies increases register pressure

Unaligned Memory Accesses

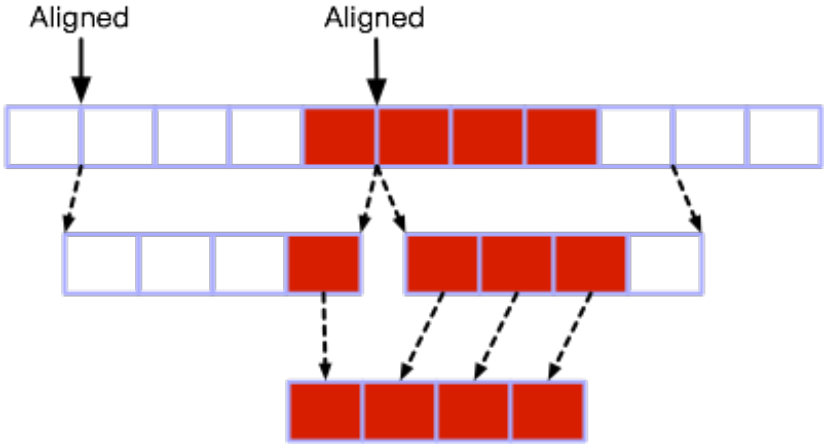
- When a memory reference accesses a location that does not match with the memory access granularity of the processor, it is called an unaligned access

- Data typically need to be aligned for vector operations
  - e.g., 16-byte boundary, 32-byte boundary
  - If unaligned, performance penalty

- If data items accessed by vector load/store instructions are not adjacent, they must be accessed separately using multiple load/store operations
  - Less efficient
  - Strided load/store or scatter/gather instructions required

Unaligned Memory Accesses (contd.)

- Read the aligned memory word that is located before the unaligned position and shift out the unnecessary bytes
- Read the aligned word that is located next to the unaligned position and discard the unnecessary bytes
- Merge the two parts that were extracted previously



Instructions for Vectors

- Modern processors typically support the following types of vector instructions:
  - Memory accesses
  - Data copying
  - Type conversion
  - Data processing

- Vector intrinsics (intrinsic functions)
  - Provide similar functionality to inline assembly that uses vector instructions
  - Provide additional features, such as type checking and automatic register allocation

벡터 인트린직

- 컴파일러가 제공하는 빌트인(built-in) 함수
  - 사용자가 C 코드 내에 SIMD 인스트럭션을 삽입할 수 있음
  - 일반적으로 하나의 빌트인 함수는 하나의 SIMD 인스트럭션에 해당
- 사용하기 복잡하나, 효율적인 벡터화를 수행할 수 있음

벡터 인트린직의 예

- 예제의 루프는 루프 캐리드 디펜던스가 없음
- 세 가지 종류의 벡터 명령어를 사용하여 예제의 루프를 벡터화 할 수 있음
  - 배열 a, b에서 데이터를 읽는 인스트럭션
  - 덧셈을 수행하는 인스트럭션
  - 연산 결과를 Array C에 저장하는 인스트럭션

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

벡터 인트린직의 예

- Intel의 SSE 벡터 인트린직을 사용하여 벡터화 한 경우

```
typedef float v4sf __attribute__ ((vector_size (16)));
// 인트린직을 사용하기 위해서 벡터 타입을 선언해주어야 함
// 4개의 float 원소를 가진 벡터

void add(float *a, float *b, float *c, int size){
    int i, j;

// 덧셈을 수행하는 루프를 벡터 인트린직을 사용하여 벡터화 함
    for(i = 0; i+3 < size; i+=4){
    v4sf A = __builtin_ia32_loadups(a + i);
    v4sf B = __builtin_ia32_loadups(b + i);
    v4sf C = __builtin_ia32_addps(A, B);
    __builtin_ia32_storeups(c + i, C);
     }
// 벡터화하여 처리하고 남은 부분은 일반적인 명령어로 처리
    for(; i < size; i++)
        c[i] = a[i] + b[i];
}
```

Triplet Notation

- To represent array sections
- [x1 : x2 : x3] specifies the first and last subscripts, and the stride
- : x3 is optional and the default value is 1
- For simplicity, we assume x3 divides (x2 - x1)

```
a[0:m:2]

a[m:0:-2]
```

Conditions for Vectorization

- Vectorization
  - To determine if statements in an inner loop can be vectorized
- Any single-statement loop that carries no dependence can be vectorized
- All S(i) can be executed concurrently

```
for(i = 0; i < 100; i++)
{
    S: a[i] = b[i];
}
```

```
a[0:99] = b[0:99];
```

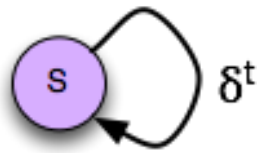
```
for(i = 0; i <= m; i = i + 2)
{
    S: a[i] = b[m - i];
}
```

```
a[0:m:2] = b[m:0:-2];
```

Conditions for Vectorization (cont'd)

- A statement contained in at least one loop can be vectorized if the statement is not included in any cycle of dependences

```
for(i = 0; i <= 100; i++)
{
  S: a[i+1] = a[i] + b[i];
}
```



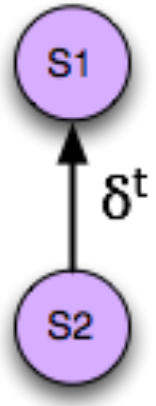
A cyclic true dependence

Conditions for Vectorization (cont'd)

- Vectorizable after reordering statements and applying loop distribution

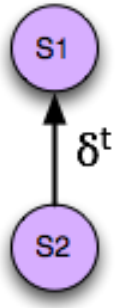
```
for(i = 1; i <= 100; i++)
{
  S1: d[i] = a[i-1] + 3.0;
  S2: a[i] = b[i] + c[i];
}
```

```
S1: d[1] = a[0] + 3.0;
S2: a[1] = b[1] + c[1];
S1: d[2] = a[1] + 3.0;
S2: a[2] = b[2] + c[2];
S1: d[3] = a[2] + 3.0;
S2: a[3] = b[3] + c[3];
...
```

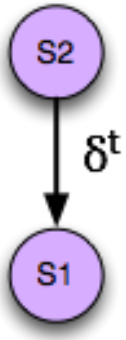


Reordering S1 and S2

```
for(i = 1; i <= 100; i++)
{
  S1: d[i] = a[i-1] + 3.0;
  S2: a[i] = b[i] + c[i];
}
```




```
for(i = 1; i <= 100; i++)
{
  S2: a[i] = b[i] + c[i];
  S1: d[i] = a[i-1] + 3.0;
}
```



Loop Distribution

```
for(i = 1; i <= 100; i++) {
  S2: a[i] = b[i] + c[i];
  S1: d[i] = a[i-1] + 3.0;
}
```

```
for(i = 1; i <= 100; i++) {
  S2: a[i] = b[i] + c[i];
}

for(i = 1; i <= 100; i++) {
  S1: d[i] = a[i-1] + 3.0;
}
```

```
a[1:100] = b[1:100] + c[1:100];
d[1:100] = a[0:99] + 3.0;
```

Conditions for Vectorization (Example)

- S가 디펜던스 사이클에 포함됨
- 벡터화가 불가능

```
for(i = 0; i <= 100; i++)
{
  S: a[i+1] = a[i] + b[i];
}
```

flow dependence의 사이클

Vector Scatter/Gather Instructions

- Want to vectorize loops with indirect accesses
- Found in Intel Xeon Phi coprocessors
  - 옛날 벡터 슈퍼컴퓨터에서 지원하던 인스트럭션

```
for (i = 0; i < N; i++)
    A[i] = B[i] + C[D[i]]; // gather
```

```
for (i=0; i<N; i++)
    A[B[i]] = C[i] + 1;     //scatter
```

Vector Conditional Execution

- Want to vectorize loops with conditional
  - Conditional execution via masking

```
for (i = 0; i < N; i++)
    if (A[i] > 0) A[i] = B[i];
```

```
if (A[0:N-1] > 0) A[0:N-1] = B[0:N-1];
```

Vector Reduction

- Loop-carried dependence on reduction variables

```
sum = 0;
for (i = 0; i < N; i++)
    sum += A[i];
```

```
// S is the vector length
sum[0:S-1] = 0;
for (i = 0; i < N; i += S)
    sum[0:S-1] += A[i:i+S-1];
do {
    S = S/2;
    sum[0:S-1] += sum[S:2*S-1];
} while (S > 1);
```

Strip Mining

- Vector registers have finite length
  - Break loops into pieces that fit into vector registers
- Converts the available parallelism into a form more suitable for the hardware

```
for (i = 0; i < N; i++) {
    a[i] = b[i] + 3;
}
```

```
// P processors available
K = ceil(N/P)
for (j = 0; j < N; j += K) {
    for (i = j; i < MIN(j+K, N); i++) {
        a[i] = b[i] + 3;
    }
}
```