© Jaejin Lee

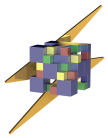
Lecture 14

Loop Optimization and Vectorization

이재진

서울대학교 컴퓨터공학부

http://aces.snu.ac.kr



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

Loop Optimization

- Consumes 90% of the execution time
  - Larger payoff to optimize the code within a loop
- Techniques
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling
  - Loop fusion
  - Loop fission

Induction Variables

- A variable x is called an induction variable of a loop if every time the variable x changes values, it is incremented or decremented by some constant
- Basic induction variables
  - Variables x whose only assignments within loop L are of the form $x = x + c$ or $x = x - c$, where c is a constant
- Dependent induction variables
  - Additional induction variables y that are defined only once within L, and whose value is a linear function of some basic induction variable x in L

Induction Variables and Strength Reduction

```
for (i =0; i < 100; i++)
{
    a[i] = 202 - 2 * i
}
```

```
t1 = 202
for (i =0; i < 100; i++)
   t1 = t1 - 2
   a[i] = t1
}
```

Induction Variable Elimination

- If there are multiple induction variables in a loop, we can eliminate those that are used only in the test condition

```
s = 0
for (i =0; i < 100; i++)
{
    s = s + 4;
}
```

```
s = 0
while (s < 400)
{
    s = s + 4;
}
```

Loop Unrolling

- Replaces the body of a loop by several copies of the body and adjusts the loop-control code accordingly
- Unrolling factor
  - The number of copies of the loop body
- Reduces the overhead of executing an indexed loop and may improve the effectiveness of other optimizations, such as common-subexpression elimination, induction-variable optimizations, instruction scheduling, and software pipelining

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

```
for (i = 0; i < 100; i += 4)
{
    s = s + a[i];
    s = s + a[i+1];
    s = s + a[i+2];
    s = s + a[i+3];
}
```

Loop Unrolling (cont'd)

- In general, loop bounds are not known constants

```
for (i = 0; i < n; i++)
{
    s = s + a[i];
}
```

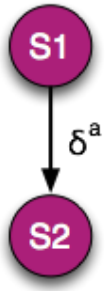
```
for (i = 0; i < (n/4)*4; i += 4)
{
    s = s + a[i];
    s = s + a[i+1];
    s = s + a[i+2];
    s = s + a[i+3];
}

for (j = i; j < n; j++)
{
    s = s + a[j];
}
```

Loop Peeling

- Similar to unrolling
- But unroll the first and/or last iterations

Loop Fusion

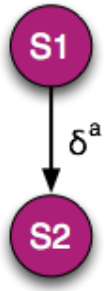
- Takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop
- Fusion is illegal if fusing two loops causes the dependence direction to be changed




```
for (i = 1; i < N; i++)
S1:   a[i] = b[i];

for (i = 1; i < N; i++)
S2:   b[i-1] = c[i] + 1;
```

Legal

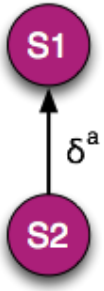
```
for (i = 1; i < N; i++) {
S1:   a[i] = b[i];
S2:   b[i-1] = c[i] + 1;
}
```




```
for (i = 1; i < N; i++)
S1:   a[i] = b[i] + 3;

for (i = 1; i < N; i++)
S2:   d[i] = a[i+1] + 1;
```

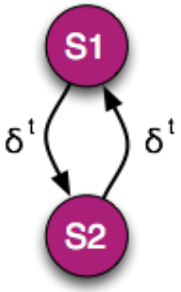
Illegal

```
for (i = 1; i < N; i++) {
S1:   a[i] = b[i] + 3;
S2:   d[i] = a[i+1] + 1;
}
```



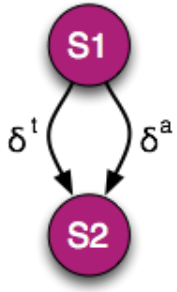
Loop Distribution (Fission)

- Takes a loop that contains multiple statements and splits it into two loops with the same iteration-space traveral
- Can be used to convert a sequential loop to multiple parallel loops
  - Converts loop-carried dependences to loop-independent dependences
- Legal when it does not result in breaking any cycles in the dependence graph of the original loop



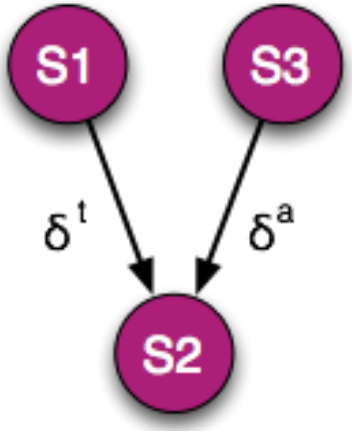

```
for (i = 1; i < N; i++) {
S1:  a[i] = b[i] + 3;
S2:  b[i-1] = a[i+1] + 1;
}
```

Illegal




```
for (i = 1; i < N; i++)
S1:  a[i] = b[i] + 3;

for (i = 1; i < N; i++)
S2:  b[i-1] = a[i+1] + 1;
```

Loop Distribution (cont'd)



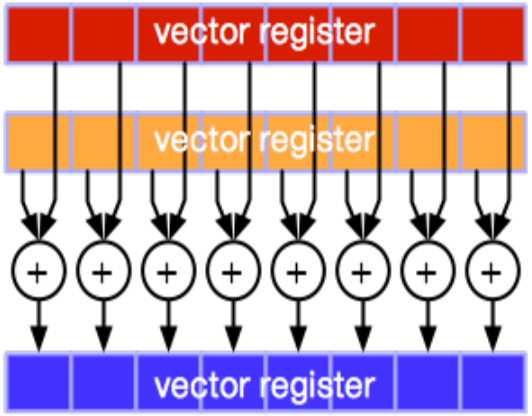

```
for (i = 1; i < N; i++) {
S1:   a[i] = b[i] + 3;
S2:   c[i] = a[i] + 1;
S3:   d[i] = c[i+1];
}
```

Legal

```
for (i = 1; i < N; i++)
S1:   a[i] = b[i] + 3;

for (i = 1; i < N; i++)
S3:   d[i] = c[i+1];

for (i = 1; i < N; i++)
S2:   c[i] = a[i] + 1;
```

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, AVX, and AVX2
  - 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

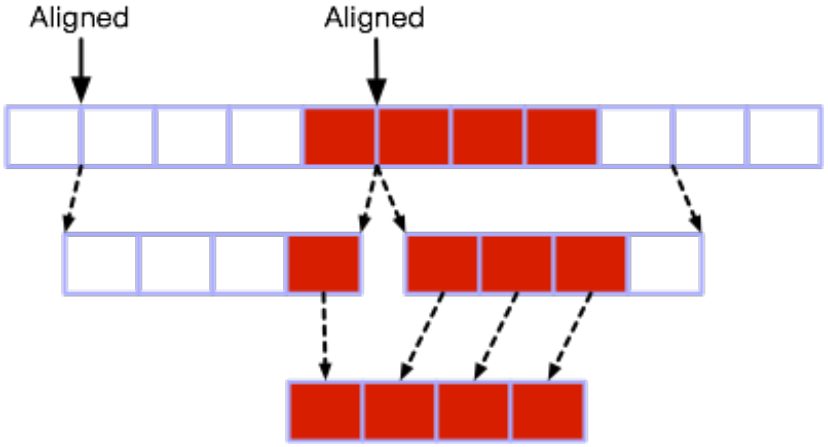
- Intel's MMX and SSE
- IBM PowerPC's AltiVec
- Limited vector instruction set
  - No vector length control
  - No strided load/store or scatter/gather
  - Unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length
  - Requires superscalar dispatch to keep multiply/add/load 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 (cont'd)

- 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

Vector Intrinsics

- Provide similar functionality to inline assembly that uses vector instructions
- Provide additional features, such as type checking and automatic register allocation

Vector Intrinsics Example

- No loop carried dependence in the loop below

```
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];
}
```

- Using Intel's SSE vector intrinsics, we obtain the code in the next slide for the loop

Vector Intrinsics Example (cont'd)

```
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];
}
```

```
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];
```

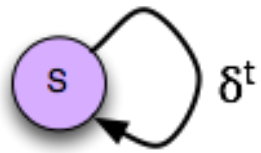
```
for(i = 0; i <= m; i = i + 2)
{
    S: a[i] = b[m - i];
}
```

```
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];
}
```



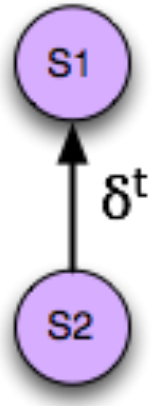
A cyclic true dependence

Conditions for Vectorization (cont'd)

- Vectorizable after reordering statements and applying loop distribution

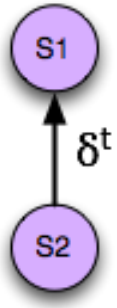
```
for(i = 1; i <= 100; i++)
{
  S1: d[i] = a[i-1] + 3.0;
  S2: a[i] = b[i] + c[i];
}
```

```
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];
...
```

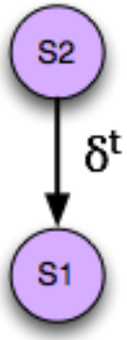


Reordering S1 and S2

```
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;
```

Vector Scatter/Gather Instructions

- Want to vectorize loops with indirect accesses
- Found in Intel Xeon Phi coprocessors

```
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;
    }
}
```