CS 610: Parallelizing Loops

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-I

Enhancing Program Performance

- Loops are one of most **commonly used** constructs in HPC programs
- Compilers perform many loop optimizations automatically to
  - Exploit fine-grained parallelism
    - Multiple pipelined functional units in each core
    - Vector instruction sets (SSE, AVX, AVX-512)
  - Exploit coarse-grained parallelism for SMP systems
    - Keep multiple asynchronous processors busy with work
  - Minimize cost of memory accesses
- In some cases, source code modifications can enhance the optimizer's ability to transform code

Different Levels of Parallelization in Hardware

Instruction-level Parallelism

Microarchitectural techniques like pipelining, OOO execution, and superscalar instruction issue

Data-level Parallelism

Use Single Instruction Multiple Data (SIMD) vector processing instructions and units

Thread-level Parallelism

Simultaneous multithreading or hyperthreading

Vectorization

Vectorization

- Vectorization is the process of transforming a scalar operation on single data elements at a time (SISD) to an operation on multiple data elements at once (SIMD)
- Helps transforms a loop nest so that the same operation is performed on several vector elements at the same time

Don't use a single Vector lane/thread!

Un-vectorized and un-threaded software will under perform

Permission to Design for All Lanes

Threading and Vectorization needed to fully utilize modern hardware

K. Rogozhin. Vectorization.

Vectorization

```c
double *a, *b, *c;
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Scalar mode

One instruction (e.g., vaddsd/vaddss) produces one result

a[i]
+
b[i]
=
c[i]

Vector mode

One instruction (e.g., vaddpd/vaddps) can produce multiple results

a[i+7]	a[i+6]	a[i+5]	a[i+4]	a[i+3]	a[i+2]	a[i+1]	a[i]
			+				
b[i+7]	b[i+6]	b[i+5]	b[i+4]	b[i+3]	b[i+2]	b[i+1]	b[i]
			=				
c[i+7]	c[i+6]	c[i+5]	c[i+4]	c[i+3]	c[i+2]	c[i+1]	c[i]

Vectorization

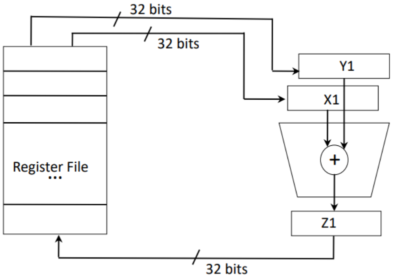
n times

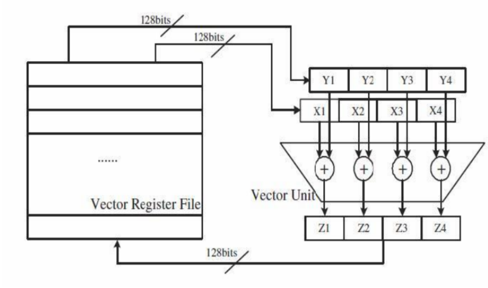
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

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

$\frac{n}{4}$ times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```

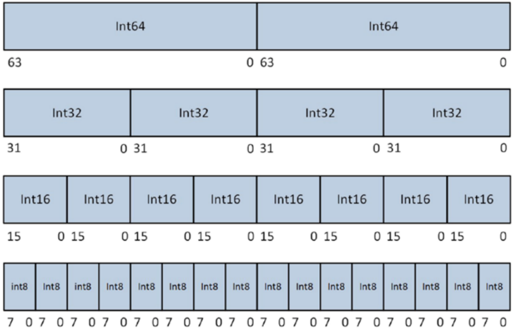




SIMD Vectorization

- Use of SIMD units can speed up the program
- Intel SSE has 128-bit vector registers and functional units
- 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

128-bit wide operands using integer types



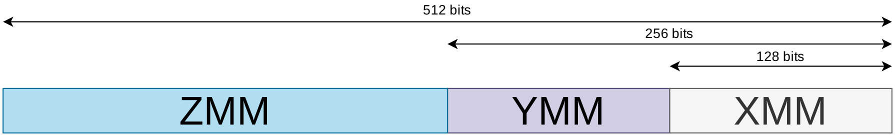
Daniel Kusswurm. Modern X86 Assembly Language Programming. 2nd edition, Apress.

Intel-Supported SIMD Extensions

SIMD Extensions	Width (bits)	SP calculations	DP calculations	Introduced
SSE2/SSE3/SSE4	128	4	2	∼2001–2007
AVX/AVX2	256	8	4	∼2011–2015
AVX-512	512	16	8	∼2017

Other platforms that support SIMD have different extensions (e.g., ARM Neon and Power AltiVec)

Intel-Supported SIMD Extensions



64-bit architecture

SSE	XMM0–XMM15	
AVX	YMM0–YMM15	Low-order 128 bits of each YMM register is aliased to a corresponding XMM register
AVX-512	ZMM0–ZMM31	Low-order 256 and 128 bits are aliased to registers YMM0–YMM31 and XMM0–XMM31 respectively

x86_64 Vector Operations

[]	required
()	optional

Example instructions

Move	`(V)MOV[A/U][P/S][D/S]`
Comparison	`(V)CMP[P/S][D/S]`
Arithmetic	`(V)[ADD/SUB/MUL/DIV][P/S][D/S]`

Instruction decoding

V	AVX
P,S	packed, scalar
A,U	aligned, unaligned
D,S	double-, single-precision
B,W,D,Q	byte, word, doubleword, quadword integers

x86_64 Vector Operations

`movss xmm1, xmm2`	Copy scalar single-precision floating-point value (low 32 bits) from xmm2 to xmm1
`vmovapd xmm1, xmm2`	Move aligned packed double-precision floating-point values from xmm2 to xmm1

```
vaddss xmm0,xmm1,xmm2
```

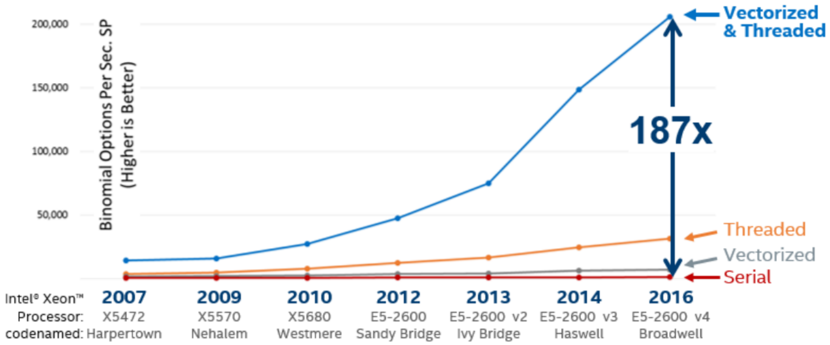
```
xmm0[31:0] = xmm1[31:0] + xmm2[31:0]
xmm0[127:32] = xmm1[127:32]
```

```
vaddsd xmm0,xmm1,xmm2
```

```
xmm0[63:0] = xmm1[63:0] + xmm2[63:0]
xmm0[127:64] = xmm1[127:64]
```

Intel Intrinsics Guide

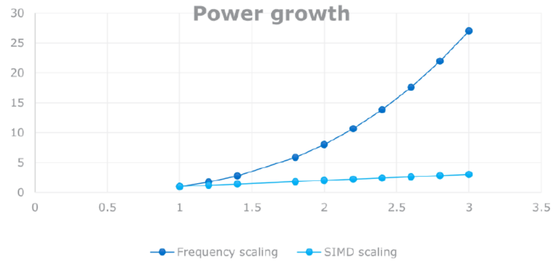
The combined effect of vectorization and threading



The Difference Is Growing With Each New Generation of Hardware

M. Voss. Topics in Loop Vectorization.

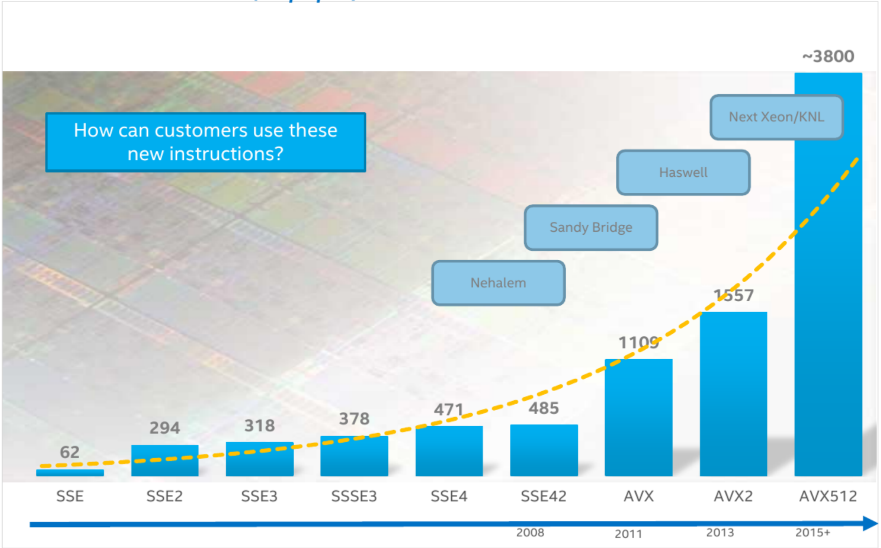
Why SIMD vector parallelism?



Wider SIMD -- Linear increase in area and power
Wider superscalar – Quadratic increase in area and power
Higher frequency – Cubic increase in power

With SIMD we can go faster with less power

Cumulative (app.) # of Vector Instructions



Vectorization Examples

- Vectorization output can vary across compiler versions and architecture generations
- Correlate the assembly code with the high-level C++ statements

- Check CPU features
- Vectorize a loop nest with increasing control

Enhancing Fine-Grained Parallelism

Focus is on vectorization of inner loops

Relevance of Data Dependence Graph in Vectorization

- Loop dependences guide vectorization
  - Statements not data dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation
- If the Data Dependence Graph (DDG) is acyclic, then vectorizing the program is straightforward

```
    for (i=1; i<=n; i++) {
S1    a[i] = b[i] + 1;
S2    c[i] = a[i-1] + 2;
    }
```

$\Rightarrow$

```
a[1:n] = b[1:n] + 1;
c[1:n] = a[0:n-1] + 2;
```

Loop Interchange (Loop Permutation)

- Switch the nesting order of loops in a perfect loop nest
- Can increase parallelism, can improve spatial locality

Loop-Carried Dependence on i

```
    DO J = 1, M
      DO I = 1, N
S         A(I+1,J) = A(I,J) + B
```

$\Downarrow$

```
    DO I = 1, N
      DO J = 1, M
S         A(I+1,J) = A(I,J) + B
```

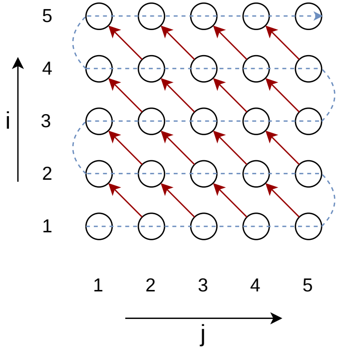
- Dependence is now carried by the outer loop, inner loop can be vectorized

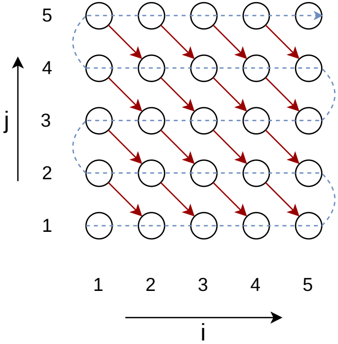
Example of Loop Interchange

```
DO i = 1, n
  DO j = 1, n
    C(i, j) = C(i-1,j+1)
```

```
DO j = 1, n
  DO i = 1, n
    C(i, j) = C(i-1,j+1)
```

valid?





Validity of Loop Permutation

(i) Construct direction vectors for all possible dependences in the loop to form a direction matrix
   - Identical direction vectors are represented by a single row in the matrix

(ii) Compute direction vectors based on the intended loop permutation

(iii) Permutation is illegal if any permuted vector is lexicographically negative

A loop nest is **fully** permutable if any permutation transformation to the loop nest is legal

Example: $d_1 = (1, -1, 1)$ and $d_2 = (0, 2, -1)$

`ijk` $\rightarrow$ `jik?` $(1, -1, 1) \rightarrow (-1, 1, 1)$: illegal

`ijk` $\rightarrow$ `kij?` $(0, 2, -1) \rightarrow (-1, 0, 2)$: illegal

`ijk` $\rightarrow$ `ikj?` $(0, 2, -1) \rightarrow (0, -1, 2)$: illegal

Does Loop Interchange/Permutation Always Help?

```
DO i = 1, 10000
  DO j = 1, 1000
    a(i) = a(i) + b(j,i) * c(i)
```

```
DO i = 1, N
  DO j = 1, M
    DO k = 1, L
      a(i+1,j+1,k) = a(i,j,k) + b
```

- Benefits from loop interchange depends on the target machine, the data structures accessed, memory layout, and stride patterns
- Optimization choices for the snippet on the right
  - Vectorize J and K
  - Move K outermost and parallelize K with threads
  - Move I innermost and vectorize assuming column-major layout

Loop Shifting

- In a perfect loop nest, if loops at level $i, i+1, \ldots i+n$ carry no dependence, i.e., all dependences are carried by loops at level smaller than i or greater than $i+n$, then it is always legal to shift these loops inside of loop $i+n+1$
- These loops will not carry any dependences in their new position

Loops i to i+n

Dependence carried by outer loops	+	0	+	0	0	0
	0	+	-	+	+	0
	0	0	0	0	+	+
	0	0	0	0	0	+

Dependence carried by inner loops

Loop Shift for Matrix Multiply

```
    DO I = 1, N
      DO J = 1, N
        DO K = 1, N
S         A(I,J) = A(I,J) + B(I,K)*C(K,J)
```

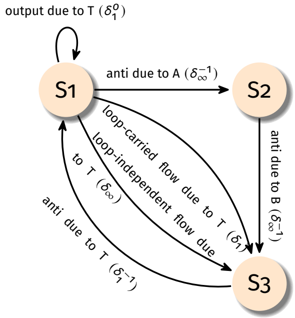
Is the loop nest vectorizable as is?

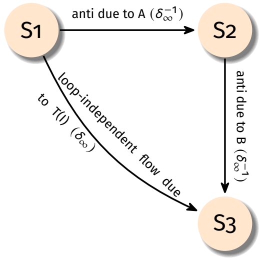
Scalar Expansion

Eliminates dependences that arise from reuse of memory locations at the cost of extra memory

```
    DO I = 1, N
S1    T = A(I)
S2    A(I) = B(I)
S3    B(I) = T
```

```
    DO I = 1, N
S1    $T(I) = A(I)
S2    A(I) = B(I)
S3    B(I) = $T(I)
    T=$T(N)
```





Understanding Scalar Expansion

Pros

+ Eliminates dependences due to reuse of memory locations, helps with parallelism

Cons

– Increases memory and addressing overhead

```
DO I = 1, N
  T = A(I) + A(I+1)
  A(I) = T + B(I)
```

can also try forward substitution

⇒

strip mining

```
DO I = 1, N, 64
  DO i = 0, 63
    T = A(I+i) + A(I+1+i)
    A(I+i) = T + B(I+i)
```

⇒

strip loop

```
DO I = 1, N, 64
  DO i = 0, 63
    $T(i) = A(I+i) + A(I+1+i)
    A(I+i) = $T(i) + B(I+i)
```

Strip mining (also known as sectioning) is a special case of 1-D loop tiling

Limits of Scalar Expansion

```
DO I = 1, N
  T = T + A(I) + A(I-1)
  A(I) = T
```

```
$T(0) = T
DO I = 1, N
  $T(I) = $T(I-1) + A(I) + A(I-1)
  A(I) = $T(I)
T = $T(N)
```

Can we parallelize the I loop?

```
      DO I = 1, 100
S1      T = A(I) + B(I)
S2      C(I) = T + T
S3      T = D(I) - B(I)
S4      A(I+1) = T * T
```

```
      DO I = 1, 100
S1      $T(I) = A(I) + B(I)
S2      C(I) = $T(I) + $T(I)
S3      $T(I) = D(I) - B(I)
S4      A(I+1) = $T(I) * $T(I)
```

Can we vectorize the loop nest?

Scalar Renaming

```
    DO I = 1, 100
S1    T = A(I) + B(I)
S2    C(I) = T + T
S3    T = D(I) - B(I)
S4    A(I+1) = T * T
```

```
    DO I = 1, 100
S1    T1 = A(I) + B(I)
S2    C(I) = T1 + T1
S3    T2 = D(I) - B(I)
S4    A(I+1) = T2 * T2
    T = T2
```

Can we vectorize the loop nest?

Allows Vectorization with Statement Interchange

```
      DO I = 1, 100
S1       T1 = A(I) + B(I)
S2       C(I) = T1 + T1
S3       T2 = D(I) - B(I)
S4       A(I+1) = T2 * T2
      T = T2
```

⇒

```
      DO I = 1, 100
S3       T2 = D(I) - B(I)
S4       A(I+1) = T2 * T2
S1       T1 = A(I) + B(I)
S2       C(I) = T1 + T1
      T = T2
```

⇒

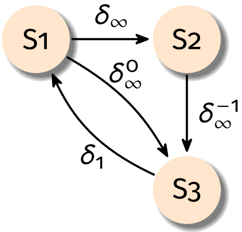
```
S3    T2(1:100) = D(1:100) - B(1:100)
S4    A(2:101) = T2(1:100) * T2(1:100)
S1    T1(1:100) = A(1:100) + B(1:100)
S2    C(1:100) = T1(1:100) + T1(1:100)
      T = T2(100)
```

Array Renaming

```
     DO I = 1, 100
S1      A(I) = A(I-1) + X
S2      Y(I) = A(I) + Z
S3      A(I) = B(I) + C
```

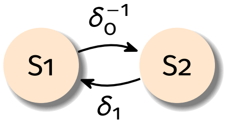
```
     DO I = 1, 100
S1      $A(I) = A(I-1) + X
S2      Y(I) = $A(I) + Z
S3      A(I) = B(I) + C
```

Array renaming requires sophisticated analysis



Node Splitting

```
    DO I = 1, 100
S1      A(I) = X(I+1) + X(I)
S2      X(I+1) = B(I) + 10
```




```
    DO I = 1, 100
S0      $X(I) = X(I+1)
S1      A(I) = $X(I) + X(I)
S2      X(I+1) = B(I) + 10
```

Can we vectorize the loop nest?

Index-Set Splitting

```
DO I = 1, 100
  A(I+20) = A(I) + B
```

strip mining

```
DO I = 1, 100, 20
  DO i = I, I+19
    A(i+20) = A(i) + B
```

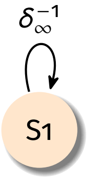
An index-set splitting transformation subdivides the loop into different iteration ranges

Loop Splitting

```
DO I = 1, N
  A(I) = A(N/2) + B(I)
```

assume N is divisible by 2

```
M = N/2
DO I = 1, M-1
  A(I) = A(N/2) + B(I)
A(M) = A(N/2) + B(I)
DO I = M+1, N
  A(I) = A(N/2) + B(I)
```



Loop Peeling

- Splits any problematic iterations (could be first, middle, or last few) from the loop body
- Change a loop-carried dependence to a loop-independent dependence
- Transformed loop carries no dependence, can be parallelized
- Peeled iterations execute in the original order, transformation is always legal to perform

```
DO I = 1, N
  A(I) = A(I) + A(1)
```

```
A(1) = A(1) + A(1)
DO I = 2, N
  A(I) = A(I) + A(1)
```

Loop splitting

Section-Based Splitting

```
    DO I = 1,N
      DO J = 1, N/2
S1      B(J,I) = A(J,I) + C
      DO J = 1,N
S2      A(J,I+1) = B(J,I) + D
```

⟹

```
    DO I = 1,N
      DO J = 1, N/2
S1      B(J,I) = A(J,I) + C
      DO J = 1,N/2
S2      A(J,I+1) = B(J,I) + D
      DO J = N/2+1, N
S3      A(J,I+1) = B(J,I) + D
```

S3 is independent

⟹

```
    DO I = 1,N
      DO J = N/2+1, N
S3      A(J,I+1) = B(J,I) + D
    DO I = 1,N
      DO J = 1,N/2
S1      B(J,I) = A(J,I) + C
      DO J = 1, N/2
S2      A(J,I+1) = B(J,I) + D
```

⟹

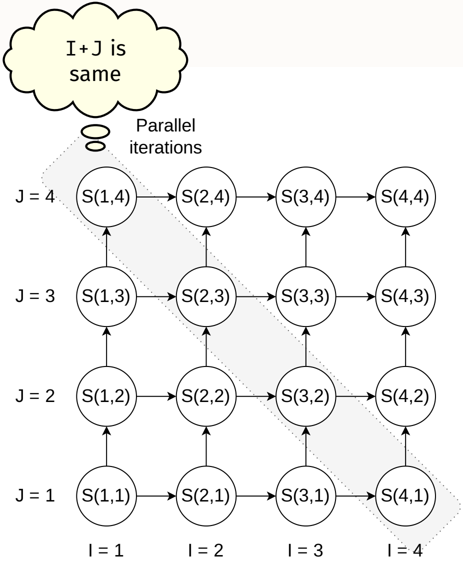
```
    M = N/2
S3  A(M+1:N,2:N+1) = B(M+1:N,1:N) + D
    DO I = 1, N
S1    B(1:M,I) = A(1:M,I) + C
S2    A(1:M,I+1) = B(1:M,I) + D
```

cannot vectorize I

Loop Skewing

```
    DO I = 1, N
      DO J = 1, N
S       A(I,J) = A(I-1,J) + A(I,J-1)
```

Which loops carry dependences?

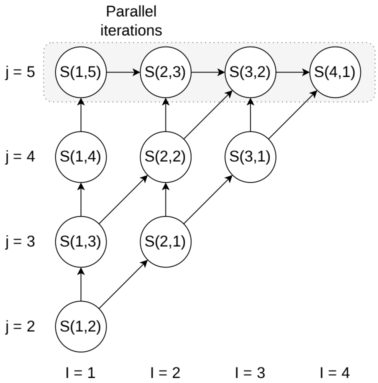


Loop Skewing

j=I+J

```
    DO I = 1, N
      DO j = I+1, I+N
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
```

What are the dependences now? Which loop carries the dependence?



Loop skewing skews the inner loop relative to the outer loop by adding the index of the outer loop times a skewing factor f to the bounds of the inner loop and subtracting the same value from all the uses of the inner loop index

Perform Loop Interchange

Given a dependency vector (a, b), skewing transforms it to $(a, fa + b)$

```
    DO I = 1, N
      DO j = I+1, I+N
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
```

$\Downarrow$

can use Fourier-Motzkin elimination

```
    DO j = 2, N+N
      DO I = max(1,j-N), min(N,j-1)
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
```

Understanding Loop Skewing

Pros

- + Reshapes the iteration space to find possible parallelism
- + Preserves lexicographic order of the dependences, is always legal
- + Allows for loop interchange in future

Cons

- – Resulting iteration space can be trapezoidal
- – Irregular loops are not very amenable for vectorization
- – Need to be careful about load imbalance

Loop Unrolling (Loop Unwinding)

- Reduce number of iterations of loops
- Add statement(s) to do work of missing iterations
- JIT compilers try to perform unrolling at run-time

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    y[i] = y[i] + a[i][j]*x[j];
  }
}
```

4-way inner loop unrolling

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

Outer Loop Unrolling + Inner Loop Jamming

```
for (i=0; i<2*n; i++) {
  for (j=0; j<m; j++) {
    loop-body(i,j);
  }
}
```

```
for (i=0; i<2*n; i+=2) {
  for (j=0; j<m; j++) {
    loop-body(i,j);
  }
  for (j=0; j<m; j++) {
    loop-body(i+1,j);
  }
}
```

```
for (i=0; i<2*n; i+=2) {
  for (j=0; j<m; j++) {
    loop-body(i,j);
    loop-body(i+1,j);
  }
}
```

2-way outer unroll does not increase operation-level parallelism in the inner loop

Is Loop Unroll and Jam Legal?

```
DO I = 1, N
  DO J = 1, M
    A(I,J) = A(I-1,J+1)+C
```

```
DO I = 1, N, 2
  DO J = 1, M
    A(I,J) = A(I-1,J+1)+C
    A(I+1,J) = A(I,J+1)+C
```

Validity Condition for Loop Unroll and Jam

- Complete unroll and jam of a loop is equivalent to a loop permutation that moves that loop innermost, without changing order of other loops
- If such a loop permutation is valid, unroll and jam of the loop is valid

- Example: 4D loop `ijkl`; $d_1 = (1, -1, 0, 2)$, $d_2 = (1, 1, -2, -1)$
  - i $d_1 \rightarrow (-1, 0, 2, 1)$, $\implies$ invalid to unroll and jam
  - j $d_1 \rightarrow (1, 0, 2, -1)$; $d_2 \rightarrow (1, -2, -1, 1)$, $\implies$ valid to unroll and jam
  - k $d_1 \rightarrow (1, -1, 2, 0)$; $d_2 \rightarrow (1, 1, -1, -2)$, $\implies$ valid to unroll and jam
  - l d_1 and d_2 are unchanged; innermost loop can always be unrolled

Understanding Loop Unrolling

Pros

- + Small loop bodies are problematic, reduces control overhead of loops
- + Increases operation-level parallelism in loop body
- + Allows other optimizations like reuse of temporaries across iterations

Cons

- – Increases the executable size
- – Increases register usage
- – May prevent function inlining

Loop Tiling (Loop Blocking)

- Improve data reuse by chunking the data in to smaller tiles (blocks)
  - All the required blocks are supposed to fit in the cache
- Performs strip mining in multiple array dimensions
- Tries to exploit spatial and temporal locality of data
- Determining the tile size
  - Requires accurate estimate of array accesses and the cache size of the target machine
  - Loop nest order also influences performance
  - Difficult theoretical problem, usually heuristics are applied
  - Cache-oblivious algorithms make efficient use of cache without explicit blocking

```
for (i = 0; i < N; i++) {
  ...
}
```

```
for (ii = 0; ii < N; ii+=B) {
  for (i = ii; i < min(N,ii+B), i++) {
    ...
  }
}
```

Validity Condition for Loop Tiling

- A band of loops is fully permutable if all permutations of the loops in that band are legal
- A contiguous band of loops can be tiled if they are fully permutable

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      loop_body(i,j,k)
```

- Example: $d = (1, 2, -3)$
  - Tiling all three loops `ijk` is not valid, since the permutation `kij` is invalid
  - 2D tiling of band `ij` is valid
  - 2D tiling of band `jk` is valid

```
for (it = 0; it < n; it+=T)
  for (jt = 0; jt < n; jt+=T)
    for (i = it; i < it+T; i++)
      for (j = jt; j < jt+T; j++)
        for (k = 0; k < n; k++)
          loop_body(i,j,k)
```

Ways to Vectorize Code

Ways to Vectorize Code

- Auto-vectorizing compiler

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

easier, but less control

- Vector intrinsics

```
void example() {
  __m128 rA, rB, rC;
  for (int i = 0; i <LEN; i+=4) {
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
    _mm_store_ps(&C[i], rC);
  }
}
```

- Assembly programming

```
..B8.5
  movaps a(,%rdx,4), %xmm0
  addps b(,%rdx,4), %xmm0
  movaps %xmm0, c(,%rdx,4)
  addq $4, %rdx
  cmpq $rdi, %rdx
  jl ..B8.5
```

harder, but more control

- Use SIMD-capable libraries like Intel Math Kernel Library (MKL)

Auto-Vectorization

Compiler vectorizes automatically	No code changes
Semi auto-vectorization	Use pragmas as hints to guide compiler
Explicit vector programming	OpenMP SIMD pragmas

Advantages

+ Transparent to programmers
+ Compilers can apply other transformations
+ Code is portable across architectures
  - Vectorization instructions may differ, but compilers take care of it

Compilers may fail to vectorize

- Programmers may give hints to help the compiler
- Programmers may have to manually vectorize their code

Steps in Loop Vectorization

- Compiler computes the dependences
  (i) The compiler figures out dependences by
    - Solving a system of (integer) equations (with constraints)
    - Demonstrating that there is no solution to the system of equations
  (ii) Tries to remove cycles in the dependence graph
  (iii) Determines data alignment
  (iv) Determines if vectorization is profitable
    - Loop vectorization is not always a legal and profitable transformation

Vectorizing a loop with several statements is equivalent to strip-mining the loop and then applying loop distribution

```
for (i=0; i<LEN; i++) {
  a[i] = b[i] + 1;
  c[i] = b[i] + 2;
}
```

```
for (i=0; i<LEN; i+=strip_size){
  for (j=i; j<i+strip_size; j++)
    a[j] = b[j] + 1;
  for (j=i; j<i+strip_size; j++)
    c[j] = b[j] + 2;
}
```

Acyclic Dependences

Forward dependences are vectorized

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

```
a[0:LEN-1]= b[0:LEN-1] + c[0:LEN-1];
d[0:LEN-1] = a[0:LEN-1] + 1;
```

Backward dependences can sometimes be vectorized

```
    for (i=0; i<LEN; i++) {
S1    a[i] = b[i] + c[i]
S2    d[i] = a[i+1] + 1;
    }
```

```
S2 d[0:LEN-1] = a[1:LEN] + 1;
S1 a[0:LEN-1]= b[0:LEN-1] + c[0:LEN-1];
```

```
    for (int i = 1; i<LEN; i++) {
S1    a[i] = d[i-1] + c[i];
S2    d[i] = b[i] + e[i];
    }
```

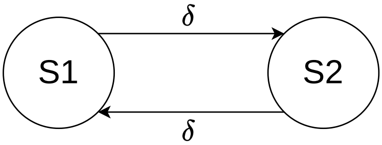
```
S2 d[1:LEN-1] = b[1:LEN-1] + e[1:LEN-1];
S1 a[1:LEN-1] = d[0:LEN-2] + c[1:LEN-1];
```

Data Dependence Graph and Vectorization

- If the DDG is cyclic, then try to transform the DDG to an acyclic graph
  - When cycles are present, vectorization can be achieved by
    - Freezing loops
    - Separating (distributing) the statements not in a cycle
    - Removing dependences
    - Changing the algorithm

Freezing Loops

```
    FOR I=1, N
      FOR J=1, M
S1        A(I,J) = B(I-1,J+1) + C
S2        B(I,J) = A(I-1,J-1) + K
```



Cycles in the DDG

```
    for (int i=0; i<LEN-1; i++) {
S1    a[i]=a[i+1]+b[i];
    }
```

Self anti-dependence can be vectorized

```
    for (int i=1; i<LEN; i++) {
S1    a[i]=a[i-1]+b[i];
    }
```

Self true dependence cannot be vectorized

```
    for (int i=1; i<LEN; i++) {
S1    a[i]=a[i-4]+b[i];
    }
```

Self true dependence with larger distance vectors can be vectorized

Vectorization in Presence of Cycles

Loop Distribution (or Fission)

```
    for (i=1; i<n; i++) {
S1    b[i] = b[i] + c[i];
S2    a[i] = a[i-1]*a[i-2]+b[i];
S3    c[i] = a[i] + 1;
    }
```

```
S1  b[1:n-1] = b[1:n-1] + c[1:n-1];
    for (i=1; i<n; i++){
S2    a[i] = a[i-1]*a[i-2]+b[i];
    }
S3  c[1:n-1] = a[1:n-1] + 1;
```

Scalar Expansion

```
    for (i=0; i<n; i++) {
S1    a = b[i] + 1;
S2    c[i] = a + 2;
    }
```

```
    for (i=0; i<n; i++) {
S1    $a[i] = b[i] + 1;
S2    c[i] = $a[i] + 2;
    }
    a = $a[n-1]
```

```
  $a[0:n-1] = b[0:n-1] + 1;
  c[0:n-1] = $a[0:n-1] + 2;
  a = $a[n-1]
```

Vectorization in Presence of Cycles

Are there transformations which allow vectorizing the following loop?

```
    for (int i=1; i<LEN; i++) {
S1    a[i] = b[i] + c[i];
S2    d[i] = a[i] + e[i-1];
S3    e[i] = d[i] + c[i];
    }
```

Cycles in the DDG

```
    for (int i=0; i<LEN; i++) {
S1    a[r[i]] = a[r[i]] * 2;
    }
```

Are there `i` and $\texttt{i}'$ such that `r[i] ==` $\texttt{r[i}'\texttt{]}$ and $\texttt{i} \neq \texttt{i}'$?

Cycles can appear in the DDG because the compiler cannot prove that there cannot be dependences

Change the Algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize
- Compilers use pattern matching to identify the recurrence and then replace it with a parallel version
- Examples of recurrences include

Reductions	`sum += A[i]`
Linear recurrences	`A[i] = B[i]*A[i-1]+C[i]`
Boolean recurrences	`if (A[i]>max) { max = A[i] }`

Reductions

Reduction is an operation, such as addition, which is applied to the elements of an array to produce a result of a lesser rank

```
sum = 0;
for (int i=0; i<LEN; ++i) {
  sum += a[i];
}
```

```
x = a[0];
index = 0;
for (int i=0; i<LEN; ++i) {
  if (a[i] > x) {
    x = a[i];
    index = i;
  }
}
```

Challenges in Vectorization

Loop Transformations using Compiler Directives

When the compiler does not vectorize automatically due to dependences, the programmer can inform the compiler that it is safe to vectorize

```c
for (int i = val; i < LEN-k; i++)
  a[i] = a[i+k]+b[i];
```

- Assume vector width is 4 elements
- This loop can be vectorized when `k < -3` and `k >= 0`

Suppose programmers know that `k>0`. How can the programmer tell the compiler that `k >= 0`?

Compiler Directives

Compiler vectorizes many loops, but many more can be vectorized if appropriate directives are used

Intel ICC	
`#pragma ivdep`	Ignore data dependences
`#pragma vector always`	Override efficiency heuristics
`#pragma novector`	Disable vectorization

GCC	
`#pragma GCC ivdep`	Asserts absence of loop-carried dependences
`#pragma GCC novector`	Disable vectorization

https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html

Aliasing

```c
void test(float* A, float* B, float* C) {
  for (int i = 0; i < LEN; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```c
float *C = &A[1];
...
void test(float* A, float* B, float* C) {
  for (int i = 0; i < LEN; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Aliasing

- To vectorize, the compiler needs to guarantee that the pointers are not aliased
- When the compiler does not know if two pointers are aliases, it can still vectorize but needs to add up to $\mathcal{O}\left(n^2\right)$ run-time checks, where n is the number of pointers
  - When the number of pointers is large, the compiler may decide to not vectorize

- Two possible workarounds
  (i) Static and globally defined arrays
  (ii) Use the `__restrict__` keyword

Resolving Aliases Using Static and Global Arrays

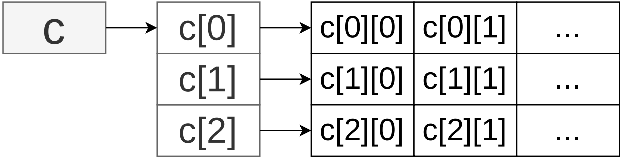
```c
float A[LEN] __attribute__ ((aligned(16)));
float B[LEN] __attribute__ ((aligned(16)));
float C[LEN] __attribute__ ((aligned(16)));
void func1() {
  for (int i=0; i<LEN; i++)
    A[i] = B[i] + C[i];
}
int main() {
  ...
  func1();
}
```

Resolving Aliases Using `__restrict__` Keyword

```
...
float *A = &B[i];
void test(float* __restrict__ A,float* __restrict__ B,
                                float* __restrict__ 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];
  }
}
int main() {
  float* A=(float*) memalign(16,LEN*sizeof(float));
  float* B=(float*) memalign(16,LEN*sizeof(float));
  float* C=(float*) memalign(16,LEN*sizeof(float));
  ...
  func1(A,B,C);
}
```

Aliasing in Multidimensional Arrays

```
void func1(float** __restrict__ a, float** __restrict__ b,
                                  float** __restrict__ c) {
  for (int i=0; i<LEN; i++)
    for (int j=1; j<LEN; j++)
      a[i][j] = b[i][j-1] * c[i][j];
}
```



Aliasing in Multidimensional Arrays

Three solutions to try when `__restrict__` does not enable vectorization

(i) Static and global arrays
(ii) Linearize the arrays and then use `__restrict__` keyword
(iii) Use compiler directives

Static and global declaration

```
float a[N][N] __attribute__ ((aligned(16)));
void t() {
  a[i][j] ...
}
int main() {
  ...
  t();
  ...
}
```

Aliasing in Multidimensional Arrays

Linearize the array

```
void t(float* __restrict__ a){
  // Access to a[i][j] is now a[i*128+j]
  ...
}
int main() {
  float* a = (float*) memalign(16,128*128*sizeof(float));
  ...
  t(a);
}
```

Use compiler directives

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

Induction Variables

Induction variables can be expressed as a function of the loop iteration variable

```c
float s = 0.0;
for (int i=0; i<LEN; i++) {
  s += 2.0;
  a[i] = s * b[i];
}
```

```c
for (int i=0; i<LEN; i++) {
  a[i] = 2.0*(i+1)*b[i];
}
```

Coding style may influence a compiler's ability to vectorize

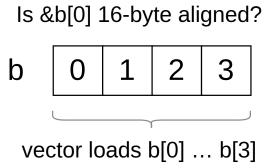
```c
for (int i=0; i<LEN; i++) {
  *a = *b + *c;
  a++; b++; c++;
}
```

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

Data Alignment

- Suppose 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

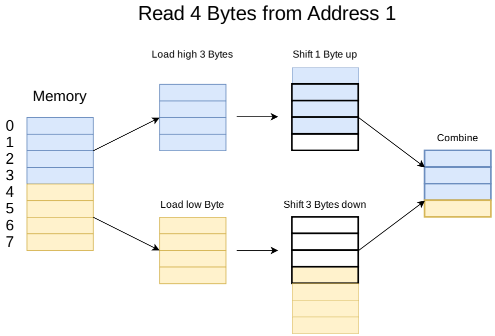
```
void test1(float *a,float *b,float *c) {
  for (int i=0;i<LEN;i++) {
    a[i] = b[i] + c[i];
  }
}
```



Why Data Alignment May Improve Efficiency?

- Vector load/store from aligned data requires one memory access
- Vector load/store from unaligned data requires multiple memory accesses and some shift operations

- A pointer is 16-byte aligned if the address is divisible by 16
  - That is, the last digit of the pointer address in hex must be 0




```c
float B[1024] __attribute__ ((aligned(16)));
int main() {
  printf("%p, %p\n", &B[0], &B[4]);
}
// Output: 0x7fff1e9d8580, 0x7fff1e9d8590
```

Data Alignment

Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16

```
// Static allocation
float b[N] __attribute__ ((aligned(16))) ;
// Dynamic allocation
float* a = (float*) memalign(16,N*sizeof(float));
```

When a pointer is passed to a function, the compiler can be made aware of alignment

```
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];
}
```

Alignment in a struct

```c
struct st {
  char A;
  int B[64];
  float C;
  int D[64];
};
int main() {
  st s1;
  printf("%p\n", &s1.A); // 0x7fffe6765f00
  printf("%p\n", &s1.B); // 0x7fffe6765f04
  printf("%p\n", &s1.C); // 0x7fffe6766004
  printf("%p\n", &s1.D); // 0x7fffe6766008
}
```

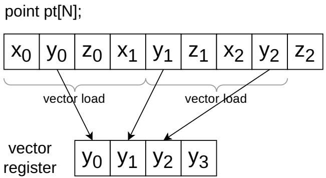
```c
struct st {
  char A;
  int B[64] __attribute__ ((aligned(16)));
  float C;
  int D[64] __attribute__ ((aligned(16)));
};
int main() {
  st s1;
  printf("%p\n", &s1.A); // 0x7fffe6765f00
  printf("%p\n", &s1.B); // 0x7fff1e9d8590
  printf("%p\n", &s1.C); // 0x7fffe6766004
  printf("%p\n", &s1.D); // 0x7fff1e9d86a0
}
```

Arrays B and D are not 16-bytes aligned

Non-unit Stride

Array of structures

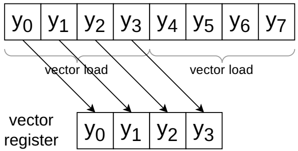
```
typedef struct{int x, y, z} point;
point pt[LEN];
for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```



Structure of arrays

```
int ptx[LEN], pty[LEN], ptz[LEN];
for (int i=0; i<LEN; i++) {
  pty[i] *= scale;

}
```



Conditional Statements

- A compiler may not vectorize a loop with a conditional if it is unsure about the profitability
  - Furthermore, removing the condition may lead to exceptions
- You may need to introduce hints like `#pragma vector always`
- Compiler may create multiple versions of the code (e.g., scalar and vector)
  - Compiler may remove the conditions when generating the vector version

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

Vectorization Examples

- Vectorization output can vary across compiler versions and architecture generations
- Correlate the assembly code with the high-level C++ statements

- Understanding alignment
  - `struct.cpp`
  - `unaligned-cost-gcc.cpp`
- `Makefile`
  - Check the `Makefile` for relevant options passed to GCC

Vectorization with Intrinsics

Vector Intrinsics

- Intrinsics are useful when
  - the compiler fails to vectorize, or
  - when the programmer thinks it is possible to generate better code than what is produced by the compiler
- Intrinsics are architecture specific

Intel Intrinsics Header Files

- We will focus on the Intel vector intrinsics
- You have to include one of the following header files for using intrinsics

SSE	`#include <xmmintrin.h>`
SSE2	`#include <emmintrin.h>`
SSE3	`#include <pmmintrin.h>`
SSSE3	`#include <tmmintrin.h>`
SSE4.1	`#include <smmintrin.h>`
SSE4.2	`#include <nmmintrin.h>`
AVX	`#include <immintrin.h>`
AVX2	`#include <immintrin.h>`
AVX512	`#include <immintrin.h>`

- Alternatively, use `#include <x86intrin.h>`, it includes all relevant headers

Intel Intrinsics Guide

Format of Intel Intrinsic APIs

```
_mm_instruction_suffix(...)
_mm256_instruction_suffix(...)
```

Suffix can take many forms

- **ss** scalar single precision
- **ps** packed (vector) singe precision
- **sd** scalar double precision
- **pd** packed double precision
- **si#** scalar integer (8, 16, 32, 64, or 128 bits)
- **su#** scalar unsigned integer (8, 16, 32, 64, or 128 bits)

Data Types

Few examples

__m128	packed single precision (vector XMM register)
__m128d	packed double precision (vector XMM register)
__m128i	packed integer (vector XMM register)

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 four single precision values

```
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```

Examples with Intrinsics

- Understanding alignment cost with intrinsics
- Inclusive prefix sum with SSE
- Makefile

Summary

- Relevance of vectorization to improve program performance is likely to increase in the future as vector lengths grow
- Compilers are often only partially successful at vectorizing code
- When the compiler fails, programmers can
  - add compiler directives, or
  - apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), use vector extensions (e.g., intrinsics or assembly) directly

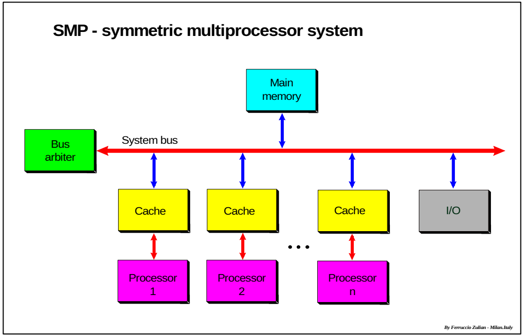
Enhancing Coarse-Grained Parallelism

Focus is on parallelization of outer loops

Find Work for Threads

Setup
- Symmetric multiprocessors with shared memory
- Threads are running on each core and are coordinating execution with occasional synchronization

Challenge Balance the granularity of parallelism with communication overheads



Challenges in Coarse-Grained Parallelism

Minimize communication and synchronization overhead while evenly load balancing across the processors

Running everything on one processor achieves minimal communication and synchronization overhead

Very fine-grained parallelism achieves good load balance, but benefits may be outweighed by frequent communication and synchronization

Privatization

Temporaries can be made local to each iteration

```
    DO I = 1,N
S1    T = A(I)
S2    A(I) = B(I)
S3    B(I) = T
```

```
    PARALLEL DO I = 1,N
      PRIVATE t
S1    t = A(I)
S2    A(I) = B(I)
S3    B(I) = t
```

Privatization

A scalar variable x in a loop L is privatizable if every path from the entry of L to a use of x in the loop passes through a definition of x

- No use of the variable is upward exposed, i.e., the use never reads a value that was assigned outside the loop
- No use of the variable is from an assignment in an earlier iteration

Computing upward-exposed variables from a block BB

$$up(BB) = use(BB) \cup \left(\neg def(BB) \cap \bigcup_{y \in succ(BB)} up(y)\right)$$

Computing privatizable variables for a loop body B where BB_0 is the entry block

$$private(B) = \neg up(BB_0) \cap \left(\bigcup_{y \in B} def(y)\right)$$

Comparing Privatization and Scalar Expansion

- Privatiztaion is preferred compared to scalar expansion
  - Less memory requirement
  - Scalar expansion may suffer from false sharing
- There are examples where scalar expansion works but privatization does not

```
DO I = 1, N
  T = A(I) + B(I)
  A(I-1) = T
```

$\xRightarrow[\text{expansion}]{\text{Scalar}}$

```
PARALLEL DO I = 1, N
  T$(I) = A(I) + B(I)
  A(I-1) = T$(I)
```

$\Downarrow$ Privatization

```
DO I = 1, N
  PRIVATE T
  T = A(I) + B(I)
  A(I-1) = T
```

$\Downarrow$

```
PARALLEL DO I = 1, N
  T$(I) = A(I) + B(I)
PARALLEL DO I = 1, N
  A(I-1) = T$(I)
```

Loop Distribution (Loop Fission)

```
    DO I = 1, 100
      DO J = 1, 100
S1      A(I,J) = B(I,J) + C(I,J)
S2      D(I,J) = A(I,J-1) * 2.0
```

```
    DO I = 1, 100
      DO J = 1, 100
S1      A(I,J) = B(I,J) + C(I,J)

      DO J = 1, 100
S2      D(I,J) = A(I,J-1) * 2.0
```

Eliminates loop-carried dependences

Validity Condition for Loop Distribution

A loop with two statements can be distributed if there are no dependences from any instance of the **later** statement to any instance of the **earlier** one

- Sufficient (but not necessary) condition
- Generalizes to more statements

```
    DO I = 1, N
S1      A(I) = B(I) + C(I)
S2      E(I) = A(I+1) * D(I)
```

```
    DO I = 1, N
S1      A(I) = B(I) + C(I)
S2      E(I) = A(I-1) * D(I)
```

Performing Loop Distribution

Steps

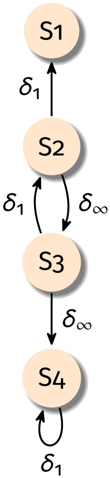
(i) Build the DDG

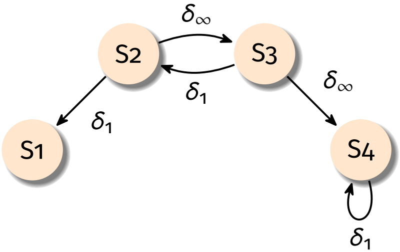
(ii) Identify strongly-connected components (SCCs) in the DDG

(iii) Make each SCC a separate loop

(iv) Arrange the new loops in a topological order of the DDG

```
    DO I = 1, N
S1    A[I] = A[I] + B[I-1]
S2    B[I] = C[I-1] + X
S3    C[I] = B[I] + Y
S4    D[I] = C[I] + D[I-1]
```






```
    DO I = 1, N
S2    B[I] = C[I-1] + X
S3    C[I] = B[I] + Y
    DO I = 1, N
S1    A[I] = A[I] + B[I-1]
    DO I = 1, N
S4    D[I] = C[I] + D[I-1]
```

Understanding Loop Distribution

Pros

+ Execute source of a dependence before the sink
+ Reduces the memory footprint of the original loop for both data and code
+ Improves opportunities for vectorization

Cons

– Can increase the synchronization required between dependence points

Loop Alignment

Unlike loop distribution, realign the loop to compute and use the values in the same iteration

```
    DO I = 2, N
S1    A(I) = B(I) + C(I)
S2    D(I) = A(I-1) * 2.0
```

cannot be parallelized

```
    DO i = 1, N+1
      if i > 1 && i < N+1
S1      A(i) = B(i) + C(i)
      if i < N
S2      D(i+1) = A(i) * 2.0
```

carried dependence becomes loop independent

Can Loop Alignment Eliminate All Carried Dependences?

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    B(I+1) = A(I) + D
```

A is aligned, B is misaligned

⇒

```
    DO i = 1, N+1
      if i > 1
S2      B(i) = A(i-1) + D
      if i < N+1
S1      A(i) = B(i) + C
```

```
    DO I = 1, N
S1    A(I+1) = B(I) + C
S2    X(I) = A(I+1) + A(I)
```

⇒

```
    DO i = 0, N
      if i > 0
S1      A(i+1) = B(i) + C
      if i < N
S2      X(i+1) = A(i+2) + A(i+1)
```

Loop Fusion (Loop Jamming)

```
    DO I = 1, N
S1    A(I) = B(I) + 1
S2    C(I) = A(I) + C(I-1)
S3    D(I) = A(I) + X
```

$\Longrightarrow$

```
L1  DO I = 1, N
      A(I) = B(I) + 1
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
L3  DO I = 1, N
      D(I) = A(I) + X
```

$\Longrightarrow$

```
L13  DO I = 1, N
       A(I) = B(I) + 1
       D(I) = A(I) + X
L2   DO I = 1, N
       C(I) = A(I) + C(I-1)
```

Validity Condition for Loop Fusion

- Consider a loop-independent dependence between statements in two different loops (i.e., from S1 to S2)
- A dependence is fusion-preventing if fusing the two loops causes the dependence to be carried by the combined loop in the reverse direction (from S2 to S1)

```
    DO I = 1, N
S1    A(I) = B(I) + C
    DO I = 1, N
S2    D(I) = A(I+1) + E
```

loop-independent flow dependence

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    D(I) = A(I+1) + E
```

backward loop-carried anti dependence

Understanding Loop Fusion

Pros

+ Reduce overhead of loops
+ May improve temporal locality

```
    DO I = 1, N
S1    A(I) = B(I) + C
    DO I = 1, N
S2    D(I) = A(I-1) + E
```

Cons

– May decrease data locality in the fused loop

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    D(I) = A(I-1) + E
```

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J) + B(I,J)
```

Parallelizing J is good for vectorization, but not for coarse-grained parallelism

⇒

```
DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
```

Dependence-free loops should move to the outermost level

⇓

```
PARALLEL DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
```

Condition for Loop Interchange

In a perfect loop nest, a loop can be parallelized at the outermost level iff that column of the direction matrix contains only “0” entries

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J+1) = A(I,J) + B(I,J)
```

Parallel Code Generation Strategy

(i) Continue till there are no more columns to move
  - Choose a loop from the direction matrix that has all “0” entries in the column
  - Move it to the outermost position
  - Eliminate the column from the direction matrix

(ii) Pick loop with most “+” entries, move to the next outermost position
  - Generate a sequential loop
  - Eliminate the column
  - Eliminate any rows that represent dependences carried by this loop

(iii) Repeat from Step (i)

Code Generation Example

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
```

```
DO I = 1, N
  PARALLEL DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
```

How did we pick loop J for parallelization?

What is the direction matrix? Can we permute the loops?

How can we parallelize this loop?

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
```

No single loop carries all the dependences, so we can only parallelize loop K

Loop Reversal

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
```

$\Downarrow$

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = L, 1, -1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
```

When the iteration space of a loop is reversed, the direction of dependences within that reversed iteration space are also reversed

- A "+" dependence becomes a "-" dependence, and vice versa
- We cannot perform loop reversal if the loop carries a dependence

Perform Interchange after Loop Reversal

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = L, 1, -1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
```

$$\Downarrow$$

```
DO K = L, 1, -1
  DO I = 2, N+1
    DO J = 2, M+1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
```

increases options for performing other optimizations

Unimodular Transformations

Loop Transformations

```
for (int i = 0; i < N; i++)
  for (int j = 0; i < M; j++)
    S(i,j);
```

The iteration space *IS* is given by

$$IS = \{(i, j) \mid 0 \leq i < N, 0 \leq j < M\}$$

An affine loop transformation maps I to $I' = TI + t'$, where I, I' are iteration vectors.

- T has to be an integer matrix and t is a translation vector.

Unimodular Transformations

- A **unimodular** matrix is a square integer matrix having determinant 1 or -1 (e.g., $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$)
- Few loop transformations can be modeled as matrix transformations involving unimodular matrices
  - Loop interchange maps iteration (i, j) to iteration (j, i)

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}$$

  - Given transformation T is linear, the transformed dependence is given by Td where d is the dependence vector in the original iteration space

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$$

  - The transformation matrix for loop reversal of the outer loop i in a 2D loop nest is $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
  - The transformation matrix for loop skewing of a 2D loop nest (i, j) is the identity matrix T with $T_{j,i}$ equal to f, where we skew loop j with respect to loop i by a factor f

M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. TPDS'91.

Example of Loop Skewing

Original

```
FOR I=1,5
  FOR J=1,5
    A(I,J) = A(I-1,J) + A(I,J-1)
```

Dependences $D = \{(1,0),(0,1)\}$

Skewed

```
FOR I=1,5
  FOR j=I+1,I+5
    A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
```

Transformation matrix = $\left[\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right]$

Dependences $D^{'} = TD = \{(1,1),(0,1)\}$

Representing Compound Transformations

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I-1,J+1) + C
```

Loop interchange is illegal because

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Let us try loop interchange followed by loop reversal. The transformation matrix T is

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Applying T to the loop nest is legal because

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
DO J = N, 1, -1
  DO I = 1, N
    A[I,J] = A[I-1,J+1] + C
```

Challenges in Applying Transformations

Challenges in Applying Transformations

- We have discussed transformations (legality and benefits) in isolation
- Compilers need to apply compound transformations (e.g., loop interchange followed by reversal)
- It is challenging to decide on the desired transformations and their order of application
  - Choice and order is sensitive to the program input, a priori order does not work
- Other considerations: conditional execution, symbolic loop bounds, and indirect memory accesses

Which Transformations are Most Important?

- Selecting the best loops for parallelization is a NP-complete problem
- Flow dependences are difficult to remove
  - Try to reorder statements as in loop peeling, loop distribution
  - Loops often use scalars for temporary values
    - Techniques like scalar expansion and privatization can be useful

+	+	O	O
+	O	+	O
+	O	O	+
O	+	O	O
O	O	+	O
O	O	O	+

carries the most dependences

Code Examples

- check-cpu-capability.cpp
- diff-variants-v1.cpp
- diff-variants-v2.cpp
- struct.cpp
- unaligned-cost-gcc.cpp
- inclusive-prefix-sum-gcc-sse4.cpp
- unaligned-cost-intrinsics.cpp
- Makefile

References

- R. Allen and K. Kennedy. Optimizing Compilers for Multicore Architectures. Sections 5.2–5.4, 5.7.2, 5.9, 6.2.1–6.2.2, 6.2.5, 6.3.1–6.3.4, Morgan Kaufmann.
- S. Midkiff. Automatic Parallelization: An Overview of Fundamental Compiler Techniques. Sections 4.1–4.2, 4.5, 5.1–5.6, Springer Cham.
- J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Sections 4.1–4.2, 4.5, 6th edition, Morgan Kaufmann.
- M. Garzarán et al. Program Optimization Through Loop Vectorization.
- M. Voss. Topics in Loop Vectorization.
- K. Rogozhin. Vectorization.