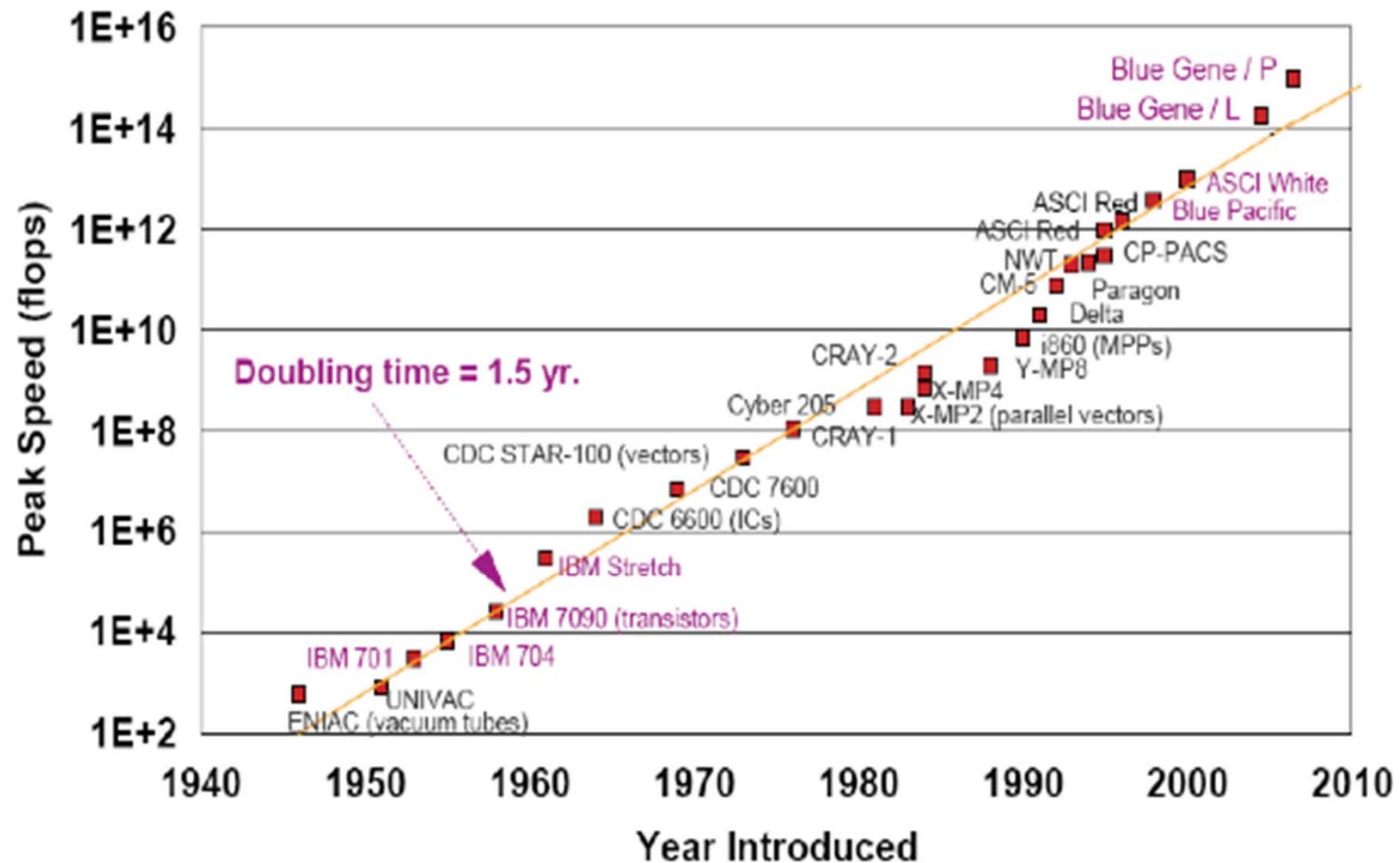


Why optimizations are difficult?

- Architecture point of view

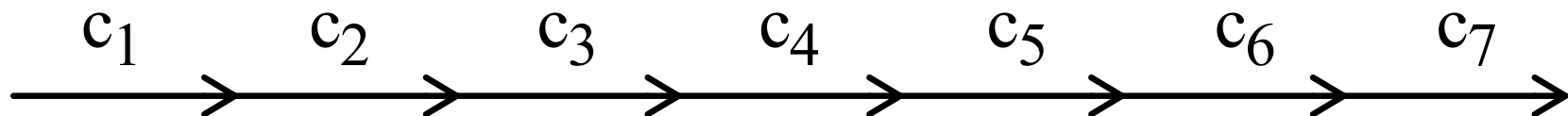
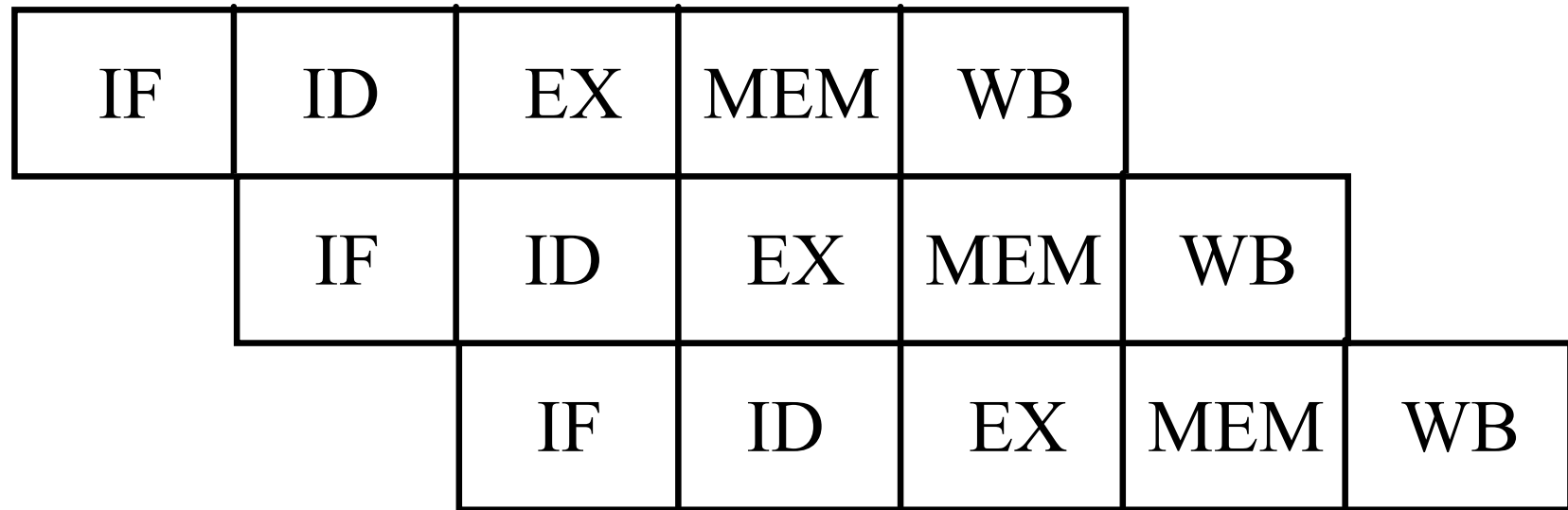
# Moor's Law



# Architectural features

- Pipelining
- Multiple execution units
  - pipelined
- Vector operations
- Parallel processing
  - Shared memory, distributed memory, message-passing
- VLIW and Superscalar instruction issue
- Registers
- Cache hierarchy
- Combinations of the above
  - Parallel-vector machines

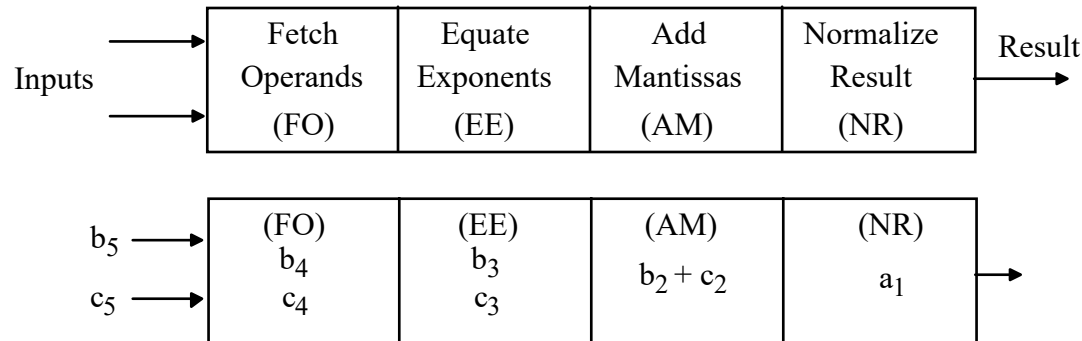
# Instruction Pipeline



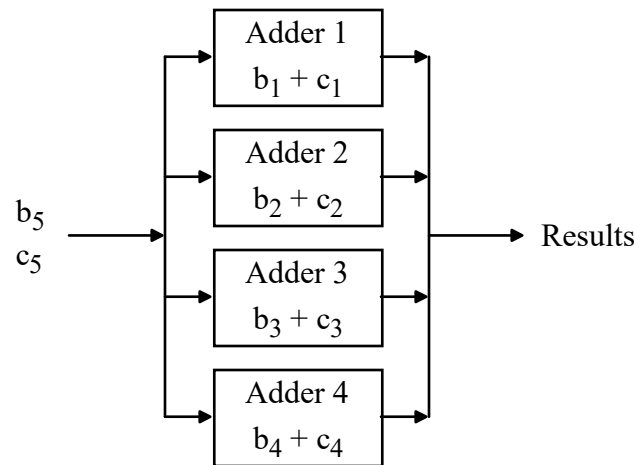
- What is the performance challenge?
  - Keep pipeline full

# Replicated Execution Logic

- Pipelined execution units



- Multiple execution units



# Vector operations

- Apply same operations to different elements

- » VLOAD        V1,A
- » VLOAD        V2,B
- » VADD         V3,V1,V2
- » VSTORE       V3,C

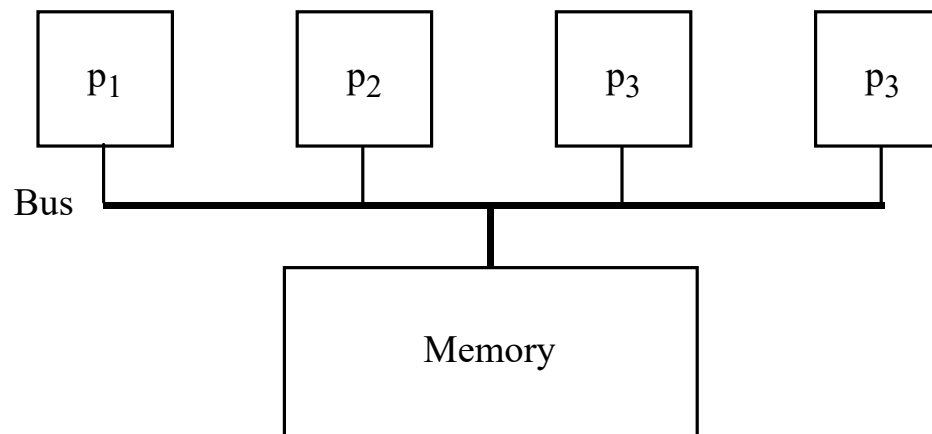
- Challenges:
  - Use all available units
    - Add copying overhead
  - Keep pipelines of execution units full

# VLIW

- Multiple instruction issue on the same cycle
  - Wide word instruction (or superscalar)
  - Usually one instruction slot per functional unit
- What are the performance challenges?
  - Finding enough parallel instructions
  - Avoiding interlocks
    - Scheduling instructions early enough

# SMP

- Multiple processors with uniform shared memory
  - Task Parallelism
    - Independent tasks
  - Data Parallelism
    - the same task on different data





# Execute programs correctly

- Bernstein's Conditions
  - When is it safe to run two tasks R1 and R2 in parallel? If none of the following holds:
    1. R1 writes into a memory location that R2 reads
    2. R2 writes into a memory location that R1 reads
    3. Both R1 and R2 write to the same memory location
- Consistency model
  - A contract between programmers and systems
  - E.g., strict consistency, casual consistency
- How can we convert this to loop parallelism?
  - Think of loop iterations as tasks

# Consistency Models

- Thread 1

l11@1ms:  $c = a + b$

l12@3ms:  $d = x + y$

- Thread 2

l21@2ms:  $t = m + n$

l22@4ms:  $c = g + h$

- **Strict** consistency model

Correct	Wrong
$c = a + b$ $t = m + n$ $d = x + y$ $c = g + h$	$c = a + b$ $t = m + n$ $c = g + h$ $d = x + y$
	$c = a + b$ $c = g + h$ $t = m + n$ $d = x + y$

# Consistency Models

- Thread 1

I11@1ms:  $c = a + b$

I12@3ms:  $d = x + y$

- Thread 2

I21@2ms:  $t = m + n$

I22@4ms:  $c = g + h$

- **Sequential** consistency model

Correct	Wrong
$c = a + b$ $t = m + n$ $d = x + y$ $c = g + h$	
$c = a + b$ $t = m + n$ $c = g + h$ $d = x + y$	$c = a + b$ $c = g + h$ $t = m + n$ $d = x + y$

# Consistency Models

- Thread 1

I11@1ms:  $c = a + b$

I12@3ms:  $d = x + y$

- Thread 2

I21@2ms:  $t = m + n$

I22@4ms:  $c = g + h$

- **Causal** consistency model

Correct	Wrong
$c = a + b$ $t = m + n$ $d = x + y$ $c = g + h$	
$c = a + b$ $c = g + h$ $t = m + n$ $d = x + y$	

# Memory hierarchy

- Problem: memory is moving farther away in processor cycles
  - Latency and bandwidth difficulties
- Solution
  - Reuse data in cache and registers
- Challenge: How can we enhance reuse, only to the point of bandwidth saturation?

# Optimizing for memory hierarchy

- Coloring register allocation works well
  - DO I = 1, N
    - »  $CI = C(I)$
  - DO J = 1, N
    - $CI = CI + A[J]$
- But only for scalars
  - DO I = 1, N
    - DO J = 1, N
      - $C(I) = C(I) + A(J)$
- Strip mining to reuse data from cache

# Distributed memory

- Memory packaged with processors
  - Message passing
  - Distributed shared memory
- SMP clusters
  - Shared memory on node, message passing off node
- What are the performance issues?
  - Minimizing communication
    - Data placement
  - Optimizing communication
    - Aggregation
    - Overlap of communication and computation

# Optimization techniques

- Program Transformations
  - Most of these architectural issues can be dealt with by restructuring transformations that can be reflected in source
    - Vectorization, parallelization, cache reuse enhancement
  - Challenges:
    - Determining when transformations are legal
    - Selecting transformations based on profitability
- Low level coding
- Some issues must be dealt with at a low level
  - Prefetch insertion
  - Instruction scheduling
- All require some understanding of the ways that instructions and statements depend on one another (share data)



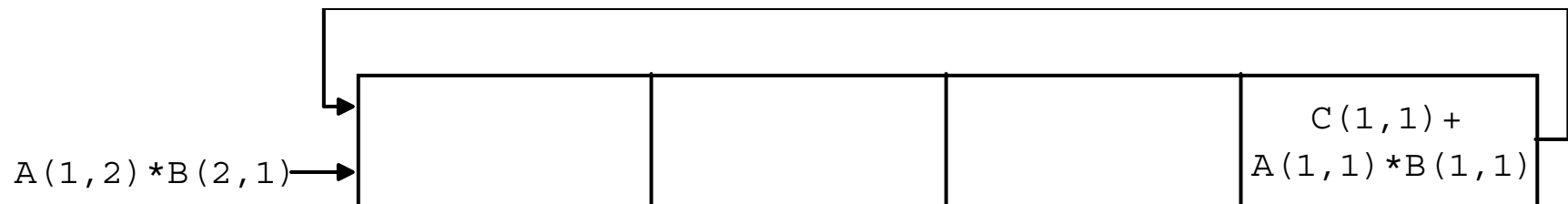
# Case study: Matrix Multiplication

# Matrix-matrix multiplication

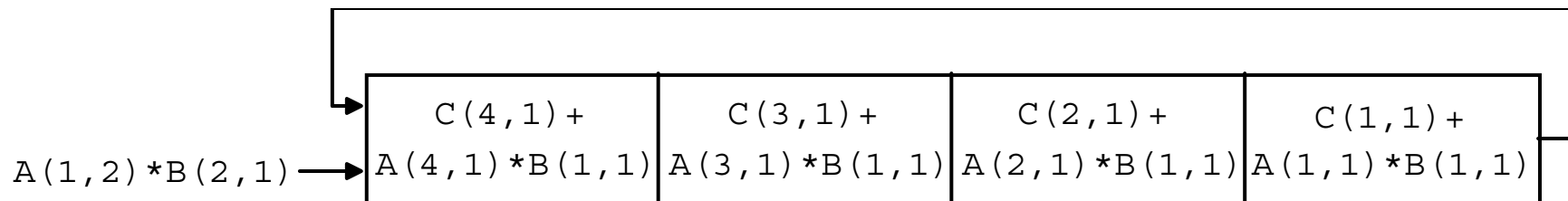
```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# Optimizing for pipeline

- Inner loop of matrix multiply is a reduction



- Solution:
  - work on several iterations of the J-loop simultaneously



# Optimized MatMul

```
DO I = 1, N,  
  DO J = 1, N, 4  
    C(J,I) = 0.0           !Register 1  
    C(J+1,I) = 0.0         !Register 2  
    C(J+2,I) = 0.0 !Register 3  
    C(J+3,I) = 0.0         !Register 4  
    DO K = 1, N  
      C(J,I) = C(J,I) + A(J,K) * B(K,I)  
      C(J+1,I) = C(J+1,I) + A(J+1,K) * B(K,I)  
      C(J+2,I) = C(J+2,I) + A(J+2,K) * B(K,I)  
      C(J+3,I) = C(J+3,I) + A(J+3,K) * B(K,I)  
    ENDDO  
  ENDDO  
ENDDO
```

# Matrix-matrix multiplication

- How to optimize for vector operations

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# Problems for vector operation

- Inner loop must be vector
  - And should be stride 1
- Vector registers have finite length (Cray: 64 elements)
  - Would like to reuse vector register in the compute loop
- Solution
  - Strip mine the loop over the stride-one dimension to 64
  - Move the iterate over strip loop to the innermost position
    - Vectorize it there

# Step 1

```
DO I = 1, N
  DO J = 1, N, 64
    DO JJ = 0, 63
      C(JJ, I) = 0.0

      DO K = 1, N

        C(J, I) = C(J, I) + A(J, K) * B(K, I)

      ENDDO
    ENDDO
  ENDDO
ENDDO
```

# Step 2

```
DO I = 1, N
  DO J = 1, N, 64
    DO JJ = 0, 63
      C(JJ, I) = 0.0
    ENDDO
    DO K = 1, N
      DO JJ = 0, 63
        C(J, I) = C(J, I) + A(J, K) * B(K, I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```



# MatMul in vector form

```
DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) +
A(J:J+63,K)*B(K,I)
    ENDDO
  ENDDO
ENDDO
```


# Matrix-matrix multiplication

- How to optimize for SMPs

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# Optimizing for SMPs

- Parallelism must be found at the outer loop level
  - But how do we know?
- Solution
  - Bernstein's conditions
    - Can we apply them to loop iterations?
    - Yes, with dependence
  - Statement S2 depends on statement S1 if
    - S2 comes after S1
    - S2 must come after S1 in any correct reordering of statements
  - Usually keyed to memory
    - Path from S1 to S2
    - S1 writes and S2 reads the same location
    - S1 reads and S2 writes the same location
    - S1 and S2 both write the same location



```
DO I = 1, N    ! Independent for all I
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) *
        B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# MatMul for shared-memory

```
PARALLEL DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
END PARALLEL DO
```

# MatMul on vector SMP

```
PARALLEL DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) +
A(J:J+63,K)*B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# Matrix-matrix multiplication

- How to optimize for cache

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

# Problems of current MatMul

- There is reuse of C but no reuse of A and B
- Solution
  - Block the loops so you get reuse of both A and B
    - Multiply a block of A by a block of B and add to block of C
  - When is it legal to interchange the iterate over block loops to the inside?



# MatMul for uniprocessor

```
DO I = 1, N, S
  DO J = 1, N, S
    DO p = I, I+S-1
      DO q = J, J+S-1
        C(q,p) = 0.0
      ENDDO
    ENDDO
    DO K = 1, N, T
      DO p = I, I+S-1
        DO q = J, J+S-1
          DO r = K, K+T-1
            C(q,p) = C(q,p) + A(q,r)*B(r,p)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

ST elements      ST elements

$S^2$  elements

# MatMul on distributed-memory

```
PARALLEL DO I = 1, N
  PARALLEL DO J = 1, N
    C(J,I) = 0.0
  ENDDO
ENDDO
PARALLEL DO I = 1, N, S
  PARALLEL DO J = 1, N, S
    DO K = 1, N, T
      DO p = I, I+S-1
        DO q = J, J+S-1
          DO r = K, K+T-1
            C(q,p) = C(q,p) + A(q,r) * B(r,p)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

# Summary

- Modern computer architectures present many performance challenges
- Most of the problems can be overcome by transforming loop nests
  - Transformations are not obviously correct
- Dependence tells us when this is feasible
  - Most of the book is about how to use dependence to do this