

# **CS536**

## **Introduction**

### **to**

## **Advanced Compiler**

**A Sahu**  
**CSE, IIT Guwahati**

**<http://jatinga.iitg.ac.in/~asahu/cs536/>**

# Outline

- Course Structure
- Pre-requisite
- Reference Books
- Grading Policy and Assignments
- Introduction and Motivation to course

<http://jatinga.iitg.ac.in/~asahu/cs536/>

# Course Structure

- Introduction to **code optimization**, efficient code generation and parallelizing compilers.
- **Data-flow analysis**
  - Classical theory, bi-directional flows, unified algo. etc.
- **Efficient code generation**
  - Register allocation heuristics and automated tools.
- **Parallelism detection**
  - Data dependence, control dependence, various restructuring transformations on loops.
- **Inter-procedural analysis**
  - Constant propagation, data dependence etc.
- **Selected case studies:** DL Transformation, ML compiler

# Pre-requisite

- Officially : CS346 (Compiler Design)
- Non-Officially
  - As Compiler Design course has removed from CSE, B.Tech Curriculum
  - It is permitted take CS536 without CS346
  - But taker need to revise basic compiler part by them self
- **Warning: C++ Coding and DS-Algorithm course**
  - Need to understand LLVM Code: lot of class hierarchy, OOPs concepts

<http://jatinga.iitg.ac.in/~asahu/cs536/>

# Reference Book and Material

- **Text Books:**

- A. Aho, M S Lam, R. Sethi and J. D. Ullman *Compilers: Principles, Techniques and Tool*, Pearson Education India, 2009 (2nd Edition) **Chap 6 onwards**
- S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

- **References:**

- The LLVM Compiler Infrastructure Project ([LLVM Website](http://llvm.org))
- Compilers for Machine Learning (C4ML): ([C4ML.org](http://C4ML.org))
- Compilers for Machine Learning (C4ML) Courses: ([C4ML @IITH](http://C4ML@IITH))
- PAPERS: and other

<http://jatinga.iitg.ac.in/~asahu/cs536/>

# Code Optimization & Performance

- ❑ Machine-independent opt
  - ❑ Code motion
  - ❑ Reduction in strength
  - ❑ Common sub-expression Elimination
- ❑ Tuning: Identifying perf bottlenecks
- ❑ Machine-dependent opt
  - ❑ Pointer code
  - ❑ Loop unrolling
  - ❑ Enabling instruction-level parallelism
- ❑ Understanding processor Opt
  - ❑ Translation of INS into OPs
  - ❑ OOO
- ❑ Branches
- ❑ Caches and Blocking
- ❑ Advice

# Speed and optimization

## ☐ Programmer

- ☐ Choice of algorithm
- ☐ Intelligent coding

## ☐ Compiler

- ☐ Choice of instructions
- ☐ Moving code
- ☐ Reordering code
- ☐ Strength reduction
- ☐ *Must be faithful to original program*

## ☐ Processor

- ☐ Pipelining
- ☐ Multiple FU
- ☐ Memory accesses
- ☐ Branches
- ☐ Caches

## ☐ Rest of system

- ☐ Uncontrollable

# Compiler is still in Research?

- **Answer is Yes**
- As long as new architecture are evolving → Compiler need to evolves..
  - Intel/AMD new generation processors
  - ARM: Apple/SD new SoC
  - Example new ISA, increase extension, parallelism, cache method
  - ML, Tensor, Security extension to Arch.
- Top tier forum (journal/conf) for Compilers:  
**TACO, PACT, PLDI, CC**



# Grading Policy And Assignment

- Venue: 5203 (this room)
- Timing : Monday 4-5, Tue:4-5, **Fri:3-4**
- Grading
  - Mid Sem : 35%, End Sem: 40%, Assignment: 20%
  - Class participation: 5%
- Assignment
  - Will be based on LLVM
  - There will be 4 Assignments, two before mid sem and two after the mid sem.

<http://jatinga.iitg.ac.in/~asahu/cs536/>

# Why to study compiler?

**Impact!**

**Techniques in compilers helps all programmers**

# Compiler technology

- **Bridge gap between HLL and Machine**

Concept of Prog. Lang.

- High Level Prog. Lang.
- Domain Specific Lang.
- Natural Lang.

Concept of Comp. Arch.

- RISC vs CISC
- Locality: Cache, memory hierarchy
- Parallelism : Instruction level, Thread level

Programmers



Prog. Language

Compilers



Machine  
Code

Prog. Tools

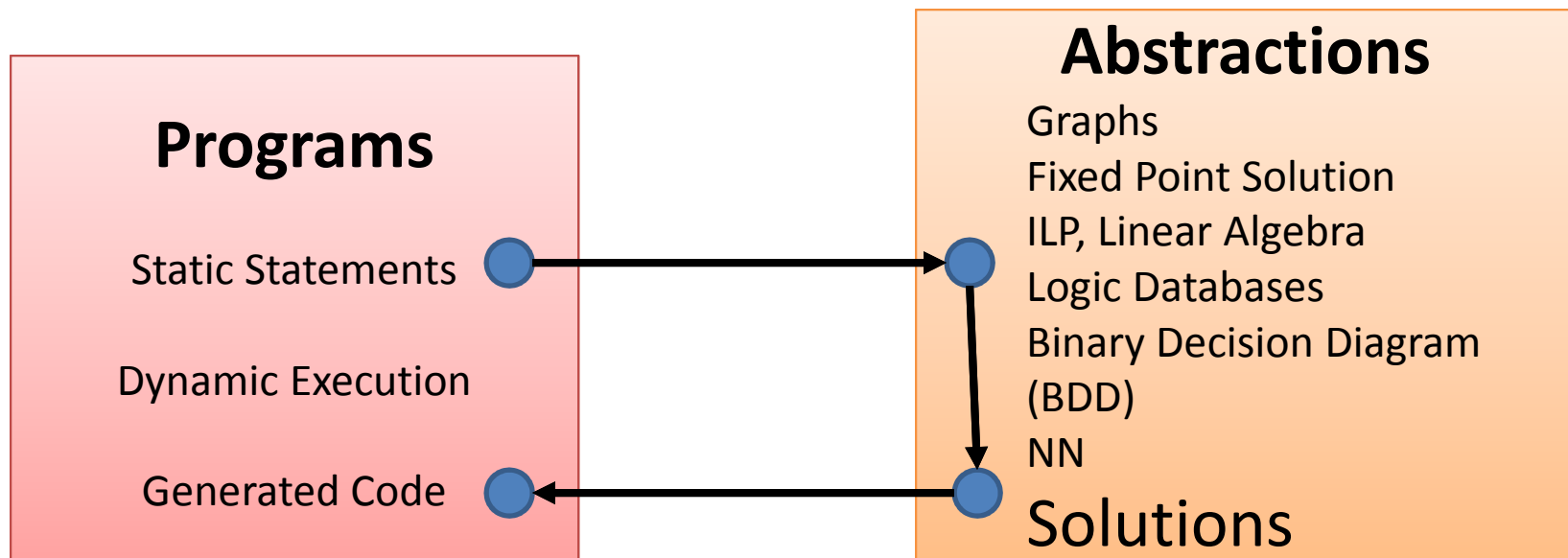
- Binary Translation
- Security

# Compiler Study Trains Good Developers

- Reasoning about programs makes better programmers
- Tool building: there are programmers and there are tool builders
- Excellent software engineering case study: Compilers are hard to build
  - Input: all programs
  - Objectives:
- Methodology for solving complex real -life problems
  - Build upon mathematical / programming abstractions

# Compilers: Where theory meets practice

- Desired solutions are often NP-complete /undecidable
- Key to success: Formulate right abstraction/ approx.
  - Can't be solved by just pure hacking
    - theory aids generality and correctness
  - Can't be solved by just theory :
    - Expt. validates & provides feedback to problem formulation
- Tradeoffs: Generality, power, simplicity, and efficiency



# Compiler Study Trains Good Developers

- Reasoning about programs makes better programmers
- Tool building: there are programmers and there are tool builders
- Excellent software engineering case study: Compilers are hard to build
  - Input: all programs
  - Objectives:
- Methodology for solving complex real -life problems
  - Build upon mathematical / programming abstractions

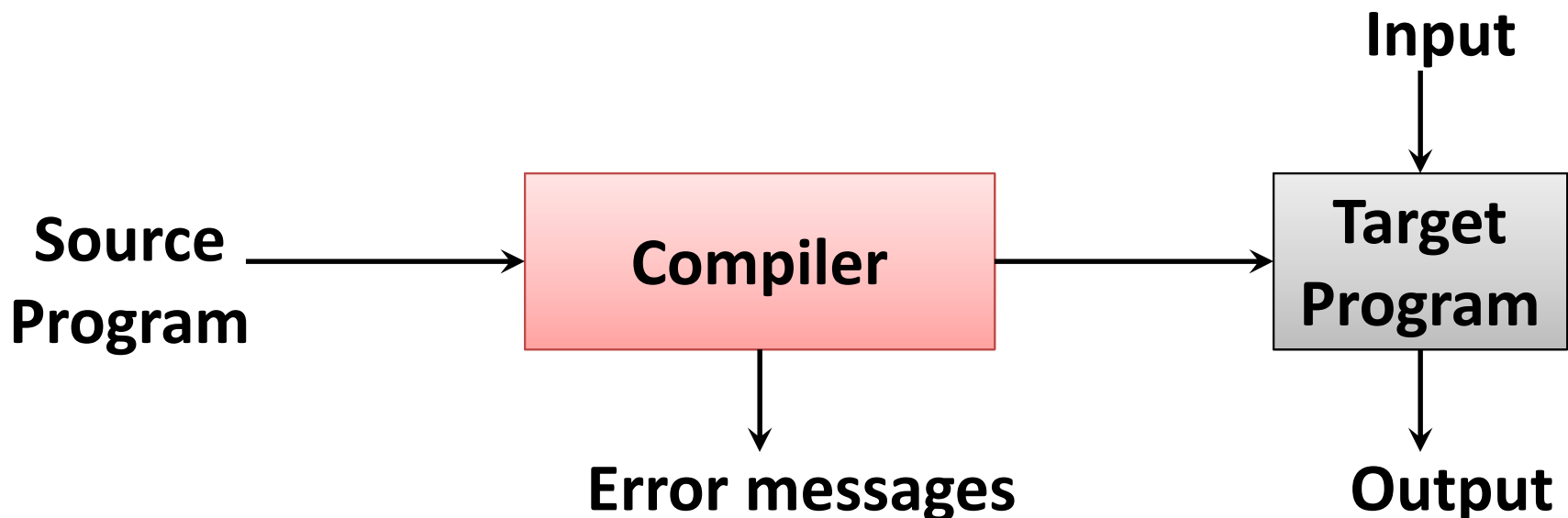
# How Can the Compiler Improve Performance?

**Execution time = Operation count \* Cycles per operation**

- Minimize the number of operations
  - arithmetic operations, memory accesses
- Replace expensive operations with simpler ones
  - e.g., replace 4 - cycle multiplication with 1 - cycle shift
- Minimize cache misses
  - both data and instruction accesses
- Perform work in parallel
  - instruction scheduling within a thread
  - parallel execution across multiple threads

# Compilers and Interpreters

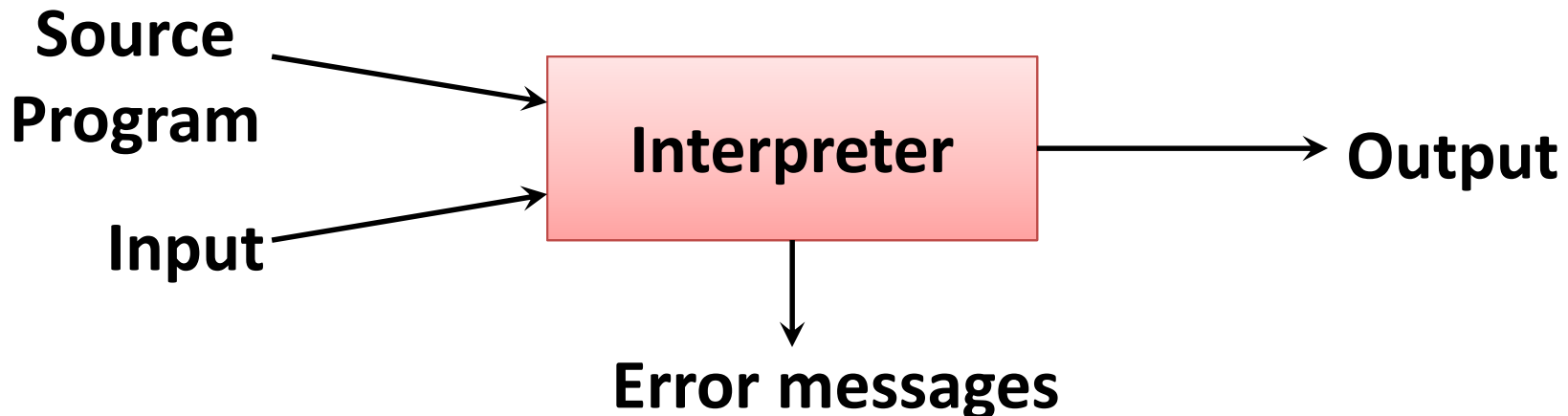
- “*Compilation*”
  - Translation of a program written in a source language into a semantically equivalent program written in a target language





# Compilers and Interpreters (cont'd)

- “*Interpretation*”
  - Performing the operations implied by the source program



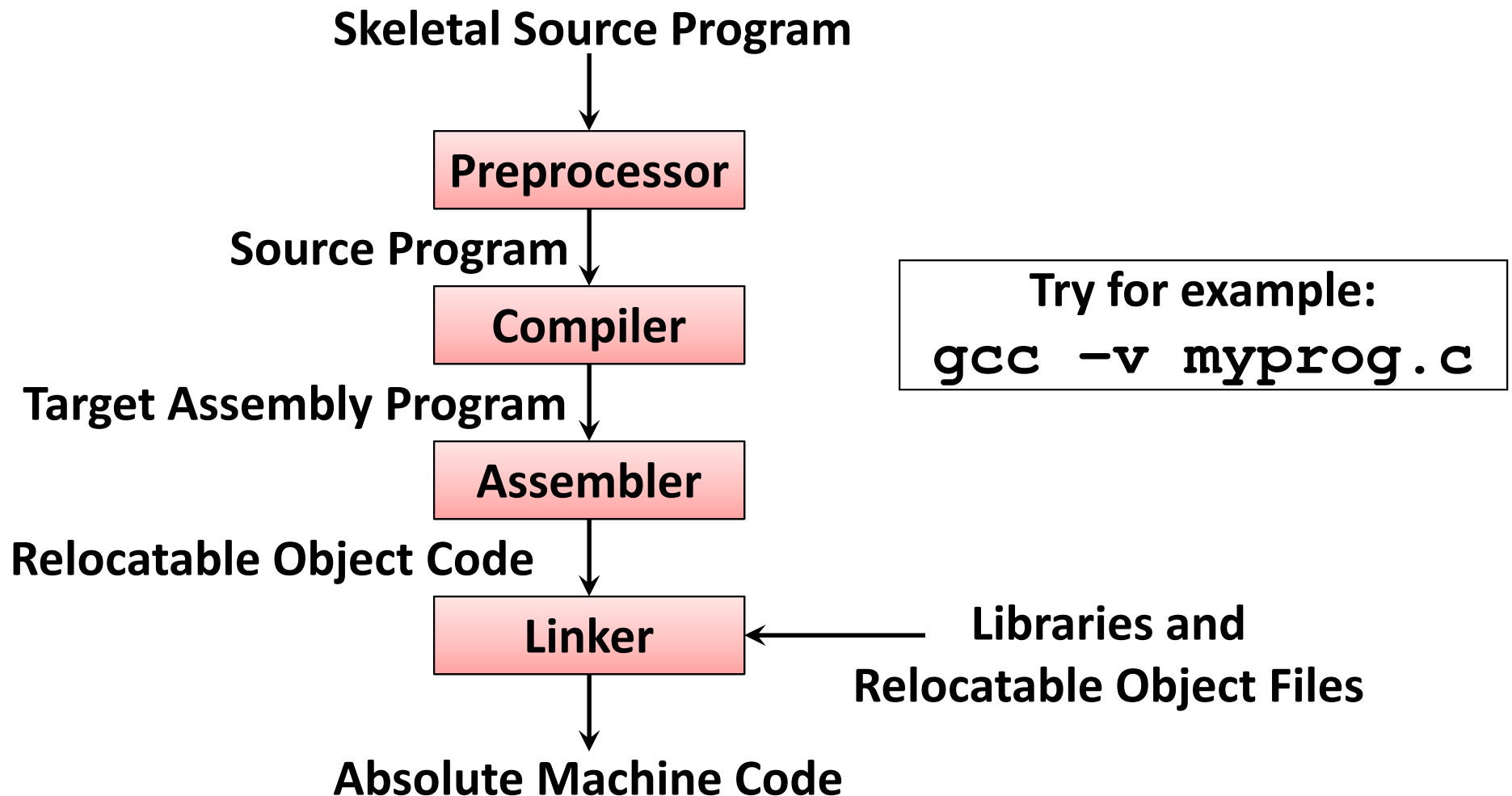
# The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

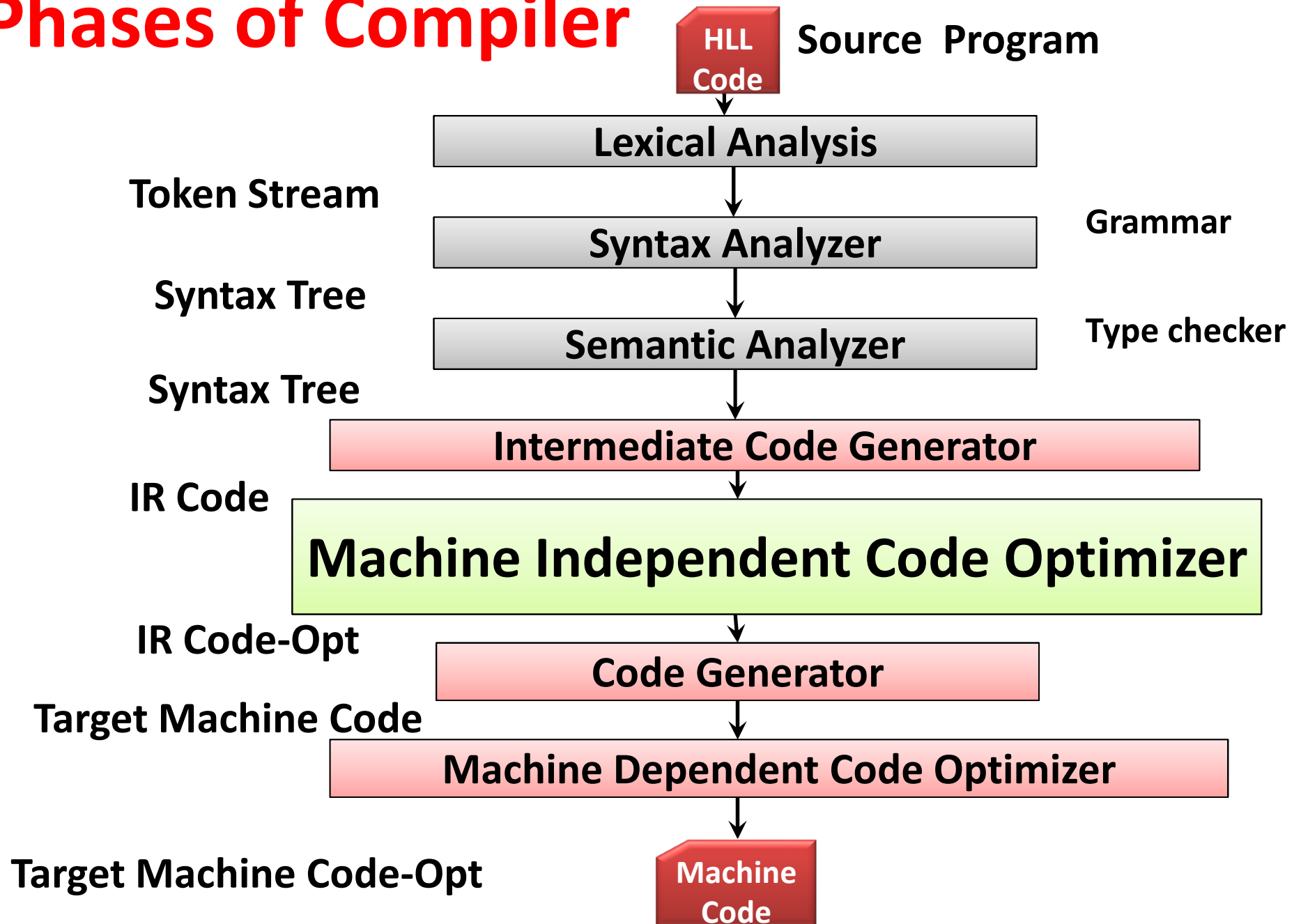
# Other Tools that Use the Analysis-Synthesis Model

- Editors (syntax highlighting)
- Pretty printers (e.g. doxygen)
- Static checkers (e.g. lint and splint)
- Interpreters
- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)

# Preprocessors, Compilers, Assemblers, and Linkers



# Phases of Compiler



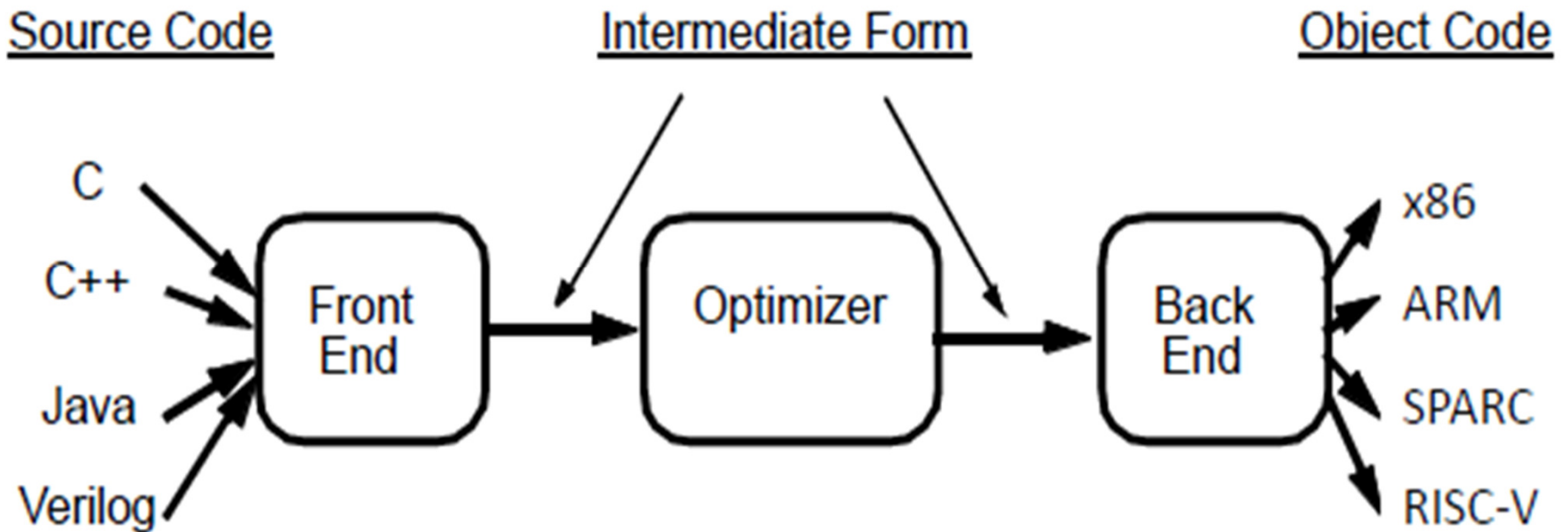
# The Phases of a Compiler

Phase	Output	Sample
<i>Programmer</i>	Source string	<b>A=B+C;</b>
<i>Scanner</i> (performs <i>lexical analysis</i> )	Token string	<b>'A', '=', 'B', '+', 'C', ';''</b> And <i>symbol table</i> for identifiers
<i>Parser</i> (performs <i>syntax analysis</i> based on the grammar of the programming language)	Parse tree or abstract syntax tree	<pre>       /               =      / \     A  +        / \       B  C           </pre>
<i>Semantic analyzer</i> (type checking, etc)	Parse tree or abstract syntax tree	
<i>Intermediate code generator</i>	Three-address code, quads, or RTL	<pre> int2fp B          t1 +      t1      C   t2 :=      t2          A           </pre>
<i>Optimizer</i>	Three-address code, quads, or RTL	<pre> int2fp B          t1 +      t1      #2.3 A           </pre>
<i>Code generator</i>	Assembly code	<pre> MOVF    #2.3, r1 ADDF2   r1, r2 MOVF    r2, A           </pre>
<i>Peephole optimizer</i>	Assembly code	<pre> ADDF2   #2.3, r2 MOVF    r2, A           </pre>

# Grouping of Phases

- Compiler front and back ends:
  - Analysis (*machine independent* front end)
  - Synthesis (*machine dependent* back end)
- Passes
  - A collection of phases may be repeated only once (*single pass*) or multiple times (*multi pass*)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory

# Structure of a Compiler



- Optimizations are performed on an “intermediate form ”
  - similar to a generic RISC instruction set
- Enables easy portability to multiple source languages, target machines



# General GCC Optimization Options

- GCC optimization : - O0, - O1, -O2, -O3
- `$man gcc`
- At – O0 level:
  - Compiler refrain from most of the opt.
  - It is correct choice for analyzing the code with debugger
- At high level
  - Mixed up source lines, eliminate redundant variable, rearrange arithmetic expressions
  - Debugger has a hard time to give user a consistent view on code and data

# General GCC Optimization Options

- Level 1
  - -fauto-inc-dec, -fmove-loop-invariant, -fmerge-constants, -ftree-copy-prop, -finline-fun-called-once
- Level 2
  - -falign-functions, -falign-loops, level, -finlining-small-fun, -finling-indirect-fun, -freorder-fun, -fstrict-aliasing
- Level 3
  - -ftree-slp-vectorize, -fvect-cost-model

# CS536 Course Flow

- Intermediate Representation
- IR Code Generation
- Runtime Environment
- Code Generation: Basic Block, SSA, Reg. Alloc.
- **Machine In/Dependent Code Optimization**
- **Optimization for ILP**
- **Opt. for Parallelism (TLP) and Cache Locality**
- Inter procedural Analysis

# Motivation 1: Optimizing Compilers for High Level Prog. Language

Example: Bubblesort

program that sorts array A allocated in static storage

```
for (i = n - 2; i >= 0; I -- ) {  
  for (j = 0; j <= I ; j++) {  
    if (A[j] > A[j+1]) {  
      temp = A[j];  
      A[j] = A[j+1];  
      A[j+1] = temp;  
    }  
  }  
}
```

# Code generated by front end

```

i := n-2
s5: if i<0 goto s1
j := 0
s4: if j>i goto s2
t1 = 4*j
t2 = &A
t3 = t2+t1
t4 = *t3          ;A[j]
t5 = j+1
t6 = 4*t5
t7 = &A
t8 = t7+t6
t9 = *t8          ;A[j+1]
if t4 <= t9 goto s3
t10 = 4*j
t11 = &A
t12 = t11+t10
temp = *t12      ;temp=A[j]

t13 = j+1
t14 = 4*t13
t15 = &A
t16 = t15+t14
t17 = *t16       ;A[j+1]
t18 = 4*j
t19 = &A
t20 = t19+t18    ;&A[j]
*t20 = t17       ;A[j]=A[j+1]
t21 = j+1
t22 = 4*t21
t23 = &A
t24 = t23+t22
*t24 = temp      ;A[j+1]=temp
s3: j = j+1
    goto s4
s2: i = i-1
    goto s5
s1:
```

*(t4=\*t3 means read memory at address in t3 and write to t4,  
\*t20=t17 means store value of t17 into memory at address in t20)*

# After optimization

Result of applying:

global common subexpression

loop invariant code motion

induction variable elimination

dead-code elimination

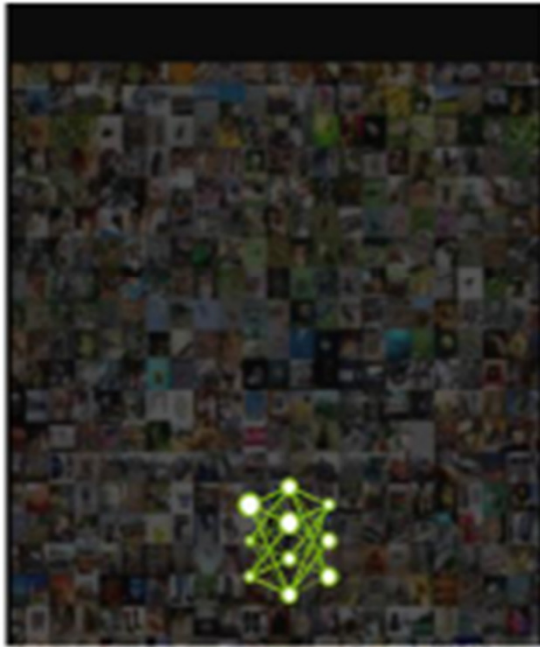
to all the scalar and temp. variables

These traditional optimizations can make  
a big difference!

```
i = n-2
t27 = 4*i
t28 = &A
t29 = t27+t28
t30 = t28+4
s5: if t29 < t28 goto s1
t25 = t28
t26 = t30
s4: if t25 > t29 goto s2
t4 = *t25           ;A[j]
t9 = *t26           ;A[j+1]
if t4 <= t9 goto s3
temp = *t25         ;temp=A[j]
t17 = *t26          ;A[j+1]
*t25 = t17          ;A[j]=A[j+1]
*t26 = temp         ;A[j+1]=temp
s3: t25 = t25+4
t26 = t26+4
goto s4
s2: t29 = t29-4
goto s5
s1:
```

# Motiv2: High Perf. Machine Learning

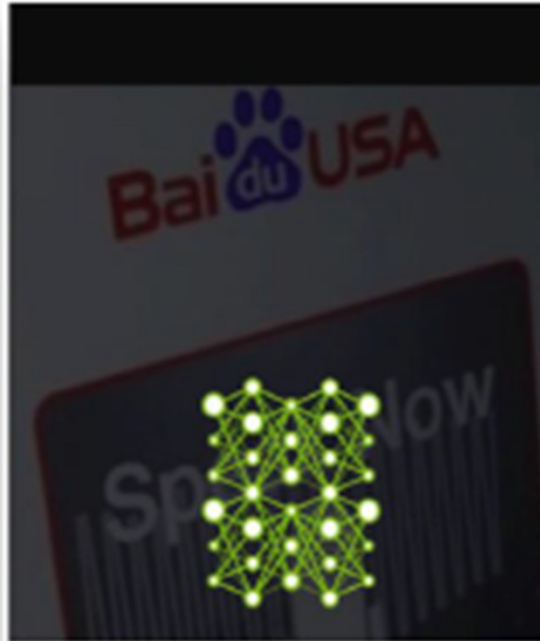
7 ExaFLOPS  
60 Million Parameters



2015

Microsoft ResNet  
Superhuman Image  
Recognition

20 ExaFLOPS  
300 Million Parameters



2016

Baidu Deep Speech 2  
Superhuman Voice  
Recognition

100 ExaFLOPS  
8700 Million Parameters



2017

Google Neural  
Machine Translation  
Near Human  
Language Translation

1 ExaFLOPS =  $10^{18}$  FLOPS

2  NVIDIA



# Motiv2:Nvidia Hopper GPU Arch



80B transistors  
900 mm<sup>2</sup>  
1650 MHz

132 Streaming Multiprocessors (SM)  
528 Tensor cores  
16,896 CUDA cores



# Each SM



**128 FP32 cores**

**128 Int cores**

**64 FP64 cores**

**4 Tensor cores**

**Tensor Cores**

**$D = A \times B + C$ ; A, B, C, D are 4x4 matrices**

**4 x 4 x 4 matrix processing array**

**1024 floating point ops / clock**

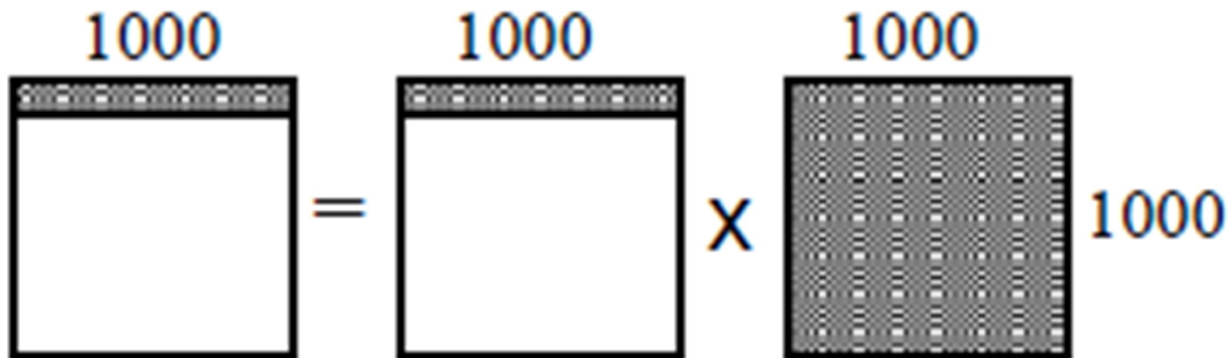
**FP64: 30 TFLOPS**

**FP32: 500 TFLOPS**

**FP16: 1000 TFLOPS**

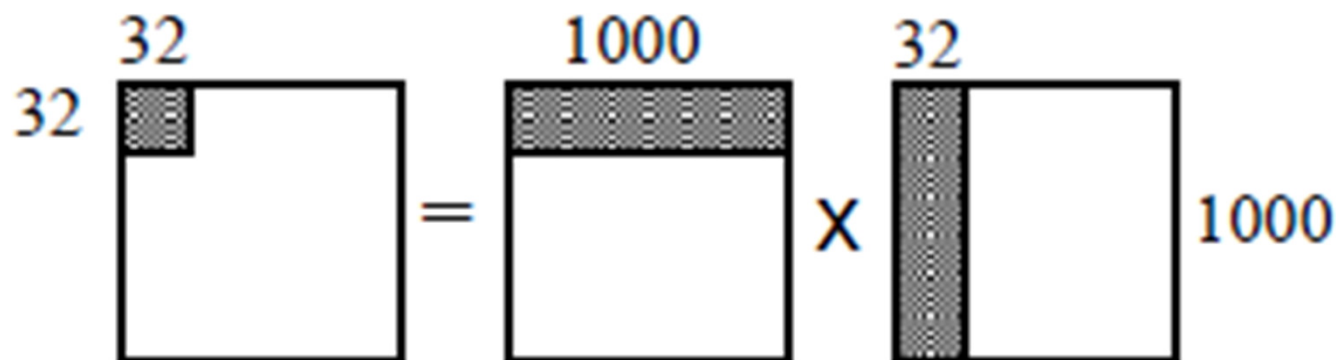
**FP8/INT8 Tensor: 2000 TFLOPS**

# Blocking for Matrix Multiplication



Data  
Accessed

1002000



65024

# Blocking for Matmul: Original Code

```
for (i= 0; i< n; i++) {  
    for (j = 0; j < n; j++) {  
        for (k = 0; k < n; k++) {  
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];  
        }  
    }  
}
```

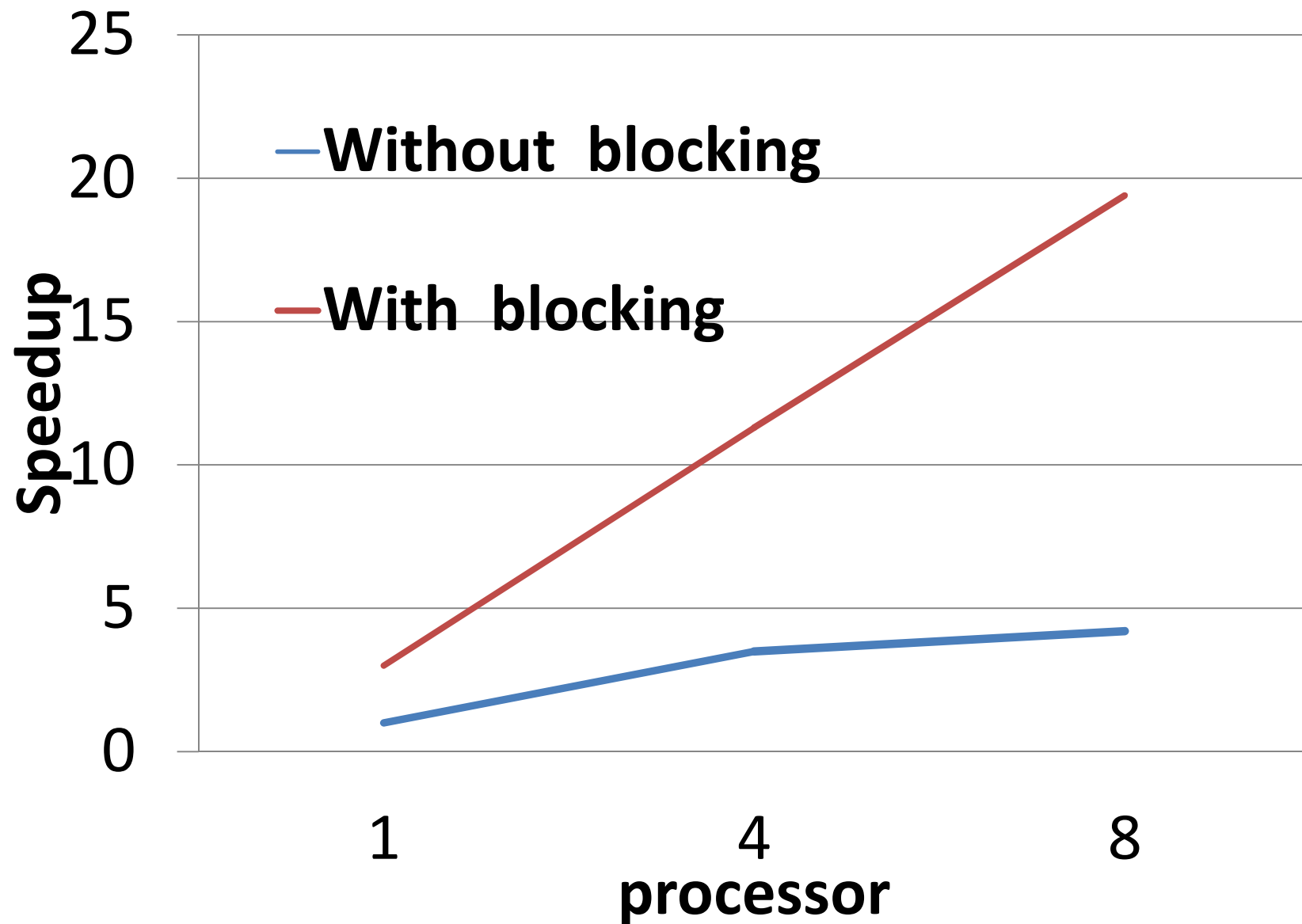
# Blocking for Matmul: Stripmine 2 outerloop

```
for (ii = 0; ii < n; ii = ii+B) {  
    for (i= ii; i< min(n,ii+B); i++) {  
        for (jj= 0; jj< n; jj= jj+B) {  
            for (j = jj; j < min(n,jj+B); j++) {  
                for (k = 0; k < n; k++) {  
                    Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];  
                }  
            }  
        }  
    }  
}
```

# Blocking for Matmul: permute

```
for (ii = 0; ii < n; ii = ii+B) {  
  for (jj= 0; jj< n; jj= jj+B) {  
    for (k = 0; k < n; k++) {  
      for (i= ii; i< min(n,ii+B); i++) {  
        for (j = jj; j < min(n,jj+B); j++) {  
          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];  
        }  
      }  
    }  
  }  
}
```

# Blocking for Matmul :Impact



# Thanks