What is a Compiler?

- Compilers lower programs from high-level languages to the low-level instruction set executed by the target processor
- The translation occurs via compiler "passes" that successively transform the program
  - A suitable Intermediate Representation (IR) is used to perform the transformations
- In addition to the lowering, compilers also perform many optimizing transformations
  - Reduce the number of instructions
  - Reduce accesses to memory (orders of magnitude slower than arithmetic ops)

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

C code

```
.L13:
    vmovaps (%r11,%rax), %ymm0
    addl    $1, %ecx
    vmulps  %ymm2, %ymm0, %ymm1
    vaddps  %ymm1, %ymm0, %ymm0
    vmovaps %ymm0, (%r11,%rax)
    addq    $32, %rax
    cmpl    %ecx, %r9d
    ja      .L13
```

x86 AVX

Grand Challenge for Computing

Achieve Productivity + Performance + Portability

- Significant SW Problem: High effort to develop high-performance (energy-efficient) apps for parallel/heterogeneous systems
- Challenges:
  - Productivity: Deep knowledge of lower-level arch. details needed to develop high-performance implementation of algorithms
  - Performance: Even with deep knowledge of arch. details, too many factors with complex inter-dependences to create "optimal" implementation
  - Portability: Different programming models for CPU/GPU/Spatial_Arrays/Clusters
    - Even within GPUs, optimizing for Nvidia Ampere vs. Hopper vs. Blackwell is very different
- Why haven't compilers solved the problem?
  - Compiler optimization requires precise information about data and control dependences: virtually impossible for languages like C++
  - Data movement (between processors and through the memory hierarchy) is much more expensive than performing the operations on the data
    - Minimizing data movement overheads is extremely challenging
- Domain-specific languages/compilers: Much better prospects for compiler-based solution to the challenge of productivity + performance + portability
- MLIR is rapidly becoming the compiler framework to build DSLs, especially for tensor-centric computational domains

Illustration: Data Movement is Expensive

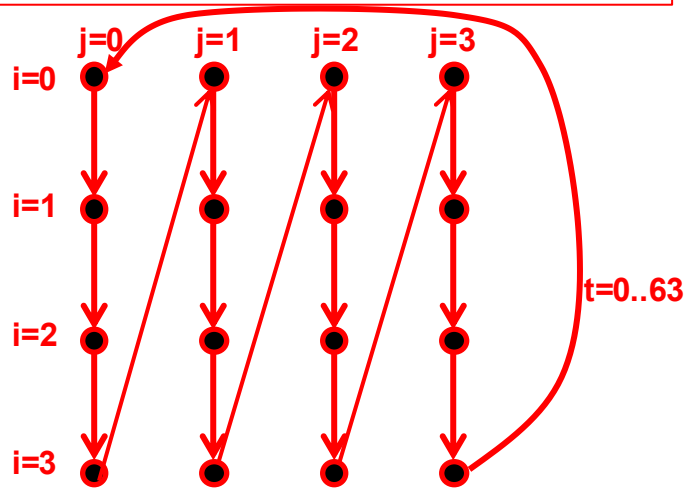
- A key reason for performance loss is data movement overheads
  - –Between nodes in a multi-node system
  - –Through the memory hierarchy at each node
- Illustrative synthetic example
  - –Functionally identical codes with very different performance

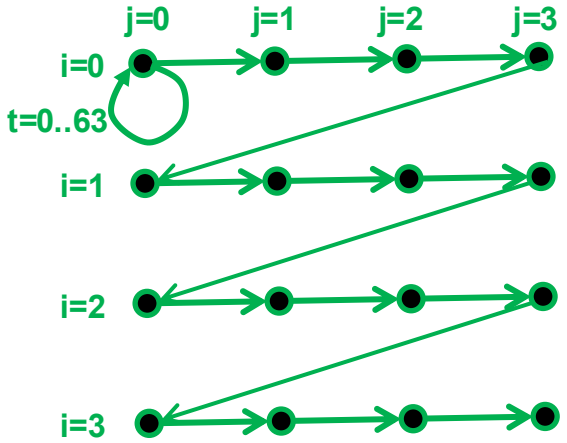
```
double A[4096][4096];
// Initialize A[][] to 0.0
for (t=0; t<64; t++)
 for (j=0; j<4096; j++)
  for (i=0; i<4096; i++)
   A[i][j] += (2*t+j)+(3*t+i);
```

6.9 sec

```
double A[4096][4096];
// Initialize A[][] to 0.0
for (i=0; i<4096; i++)
 for (j=0; j<4096; j++)
  for (t=0; t<64; t++)
   A[i][j] += (2*t+j)+(3*t+i);
```

0.17 sec





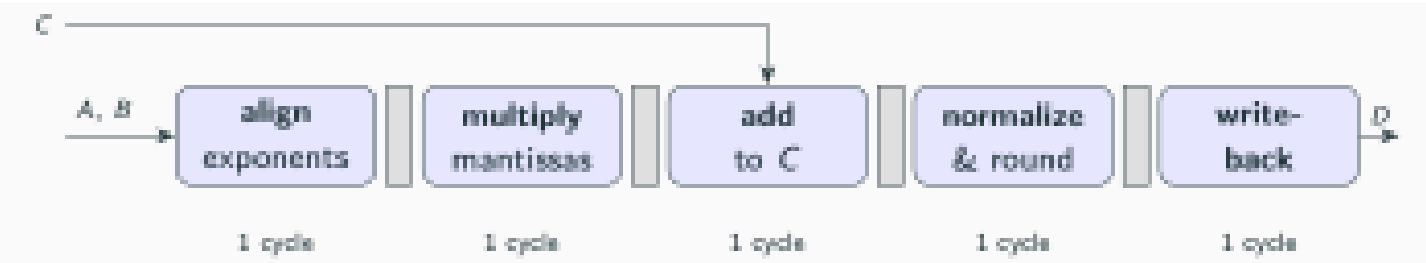
- 40x difference in execution time on my laptop (cc –Ofast)
  - Same arithmetic operations executed by both codes, but the red version accesses all data from main memory while the green version makes most accesses from cache/registers

Fundamental Performance Factors

- Three key types of HW components
  - Arithmetic Units (AFU): where the algorithm's operations get executed
    - Scalar, Vector, Matrix
  - Storage Components: where input and intermediate data is stored
    - Registers, L1/L2/L3 caches, scratchpads, main-memory (volatile), NVM
  - Networking Components
    - NOC (Network on Chip), Buses on a node, interconnection networks for clusters
- Fundamental goal is to keep AFUs as busy as possible
  - Main challenge is to keep operands flowing to the Arithmetic units
  - Two fundamental issues
    - Minimizing overheads in accessing and moving data to/from the ALUs
    - Scheduling work to keep all ALUs busy (enough load-balanced parallelism)
- Two grand challenges
  - Hardware: Architect a system where AFUs are utilized for key algs/apps?
    - Cost of access/movement of data is the main problem
  - Software: Performance modeling for effective mapping/scheduling of ops and data movement to keeps AFUs busy
    - For large dense tensor computations (high degree of "regular" parallelism), key challenge is effective layout and orchestration of data movement

Hardware Efficiency: Pipelining

- Hardware for Arithmetic Ops are internally built as multi-stage pipelines
  - Cannot build fast hardware to perform an FMA in a single clock cycle
  - Multiple smaller stages, each a combinational circuit, with latches inbetween
  - Each Op takes multiple clock cycles, but independent ops can start every cycle

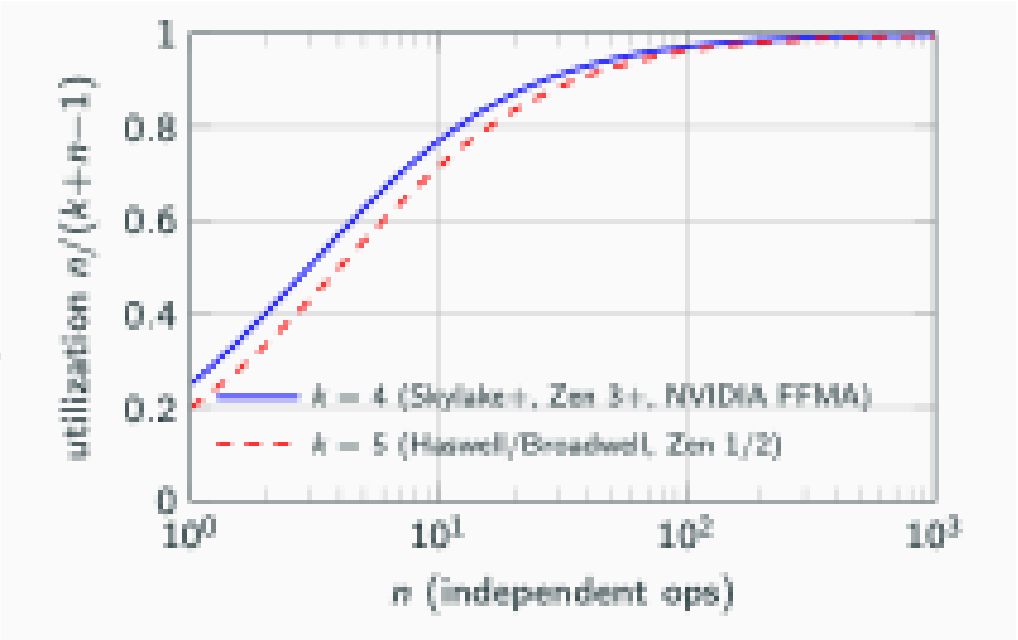


D = A*B+C

- For a k-stage pipeline and a stream of 'n' independent ops:

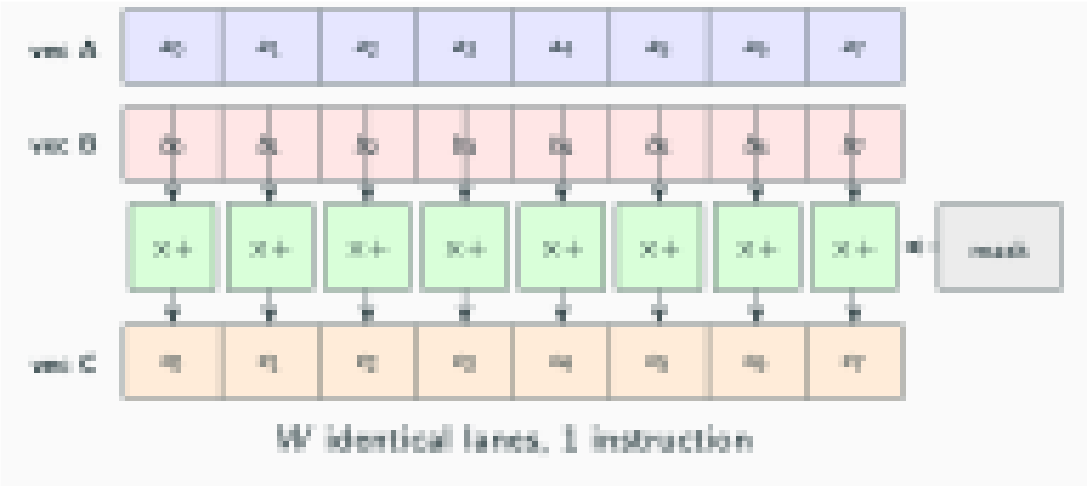
$$T(n) = (k + n - 1)\, t_{clk}, \quad \text{util}(n) = \frac{n}{k+n-1}$$

- Implication: High instruction-level, fine-grained parallelism is needed to attain a good fraction of `single-core' performance peak



Hardware Efficiency: Vector/Matrix Ops

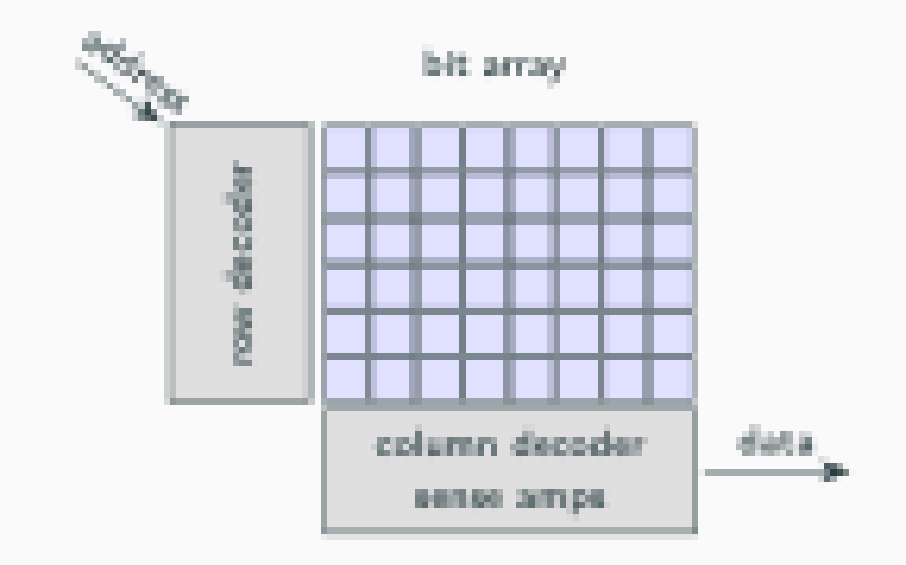
- CPUs and GPUs use SIMD (Single Instruction Multiple Data) AFUs
  - Single instruction fetch/decode/execute drives many independent arithmetic ops



- Much lower energy/op by change from Scalar Ops => Vector Ops
  - Energy/power are fundamental limiters of aggregate VLSI chip performance
- Recent trends have now gone from Vector => Matrix Ops
  - Significant benefits from reducing data movement via operand reuse and customized datapaths
- Main benefit is for `regular' algorithms (Vector/SIMD) and matrix-matrix-multiply-amenable algorithms (MMA)
  - Performance gap between dense and sparse computations increased

Why Memory Access is Fundamentally Expensive

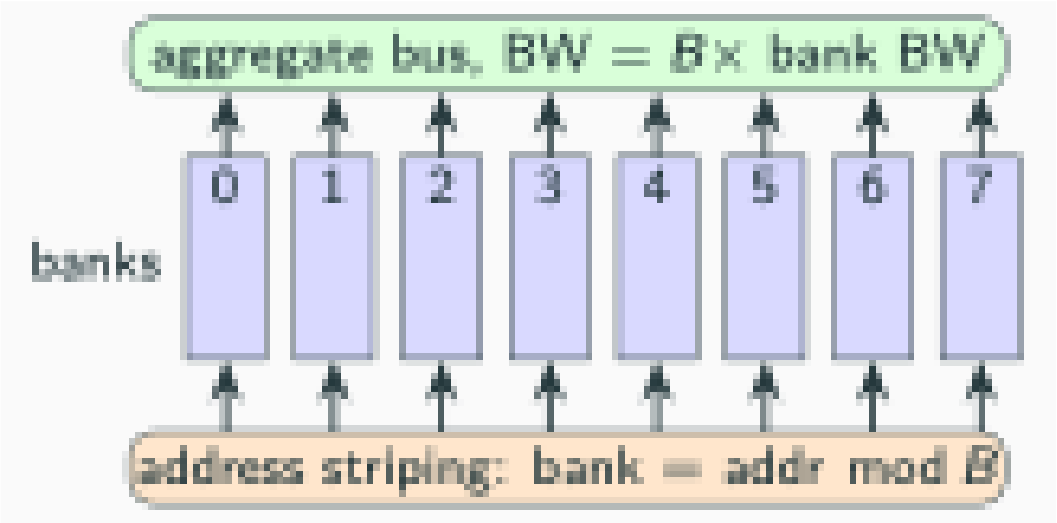
- Memory is built from 2-D arrays of bit cells



- Capacity vs. speed trade-off
  - Increase in #stages in hardware structures for row/column decoders (tree)
    - Latency: ~ $\log_2$(capacity)
  - Increase in capacitance of the bit lines (row) and word line (column)
    - Latency: ~ sqrt(capacity)
- Hence we use memory hierarchies
  - Small and fast cache/scratchpad close to ALUs
  - Larger and slower caches further from ALUs

Enhancing Bandwidth with Banks

- Mult-banked memory to increase bandwidth



- Bandwidth can be increased without (much) increase in latency
  - Address space is mapped in round-robin fashion across banks
  - Contiguous chunks of data can be accessed by concurrently accessing all banks

- Implication: Spatial locality is important for efficient data access
  - Accessing contiguous chunks of data from memory is fundamentally much more efficient than scattered data access

Compiler Optimization: Transformations

- Some broad categories of compiler optimizations
  - Reduce the number of executed instructions
    - Common Subexpression Elimination, Dead Code elimination, Constant Propagation
  - Improve instruction throughput
    - Reordering instructions based on dependences
  - Reduce overheads of data movement
    - Loop-level transformations to change data access pattern to increase reuse in registers/caches

Common Subexpression Elimination

- If an expression is recomputed and its operands are unchanged, compute it once and reuse the result

```
x = a*b + c;
y = a*b - d;
```

$\Longrightarrow$

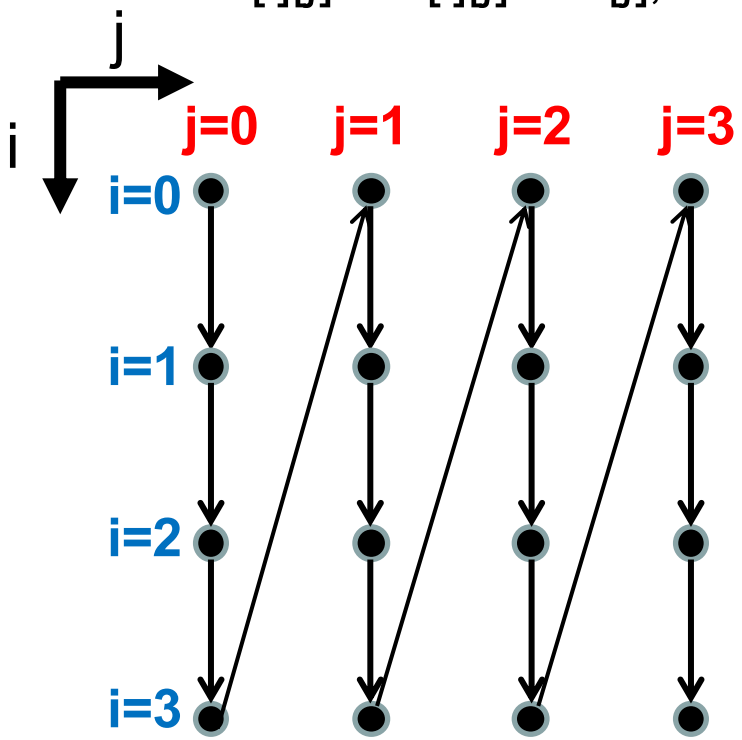
```
t = a*b;
x = t + c;
y = t - d;
```

- Saves one multiplication by recognizing the common sub-expression

Loop Transformation: Permutation

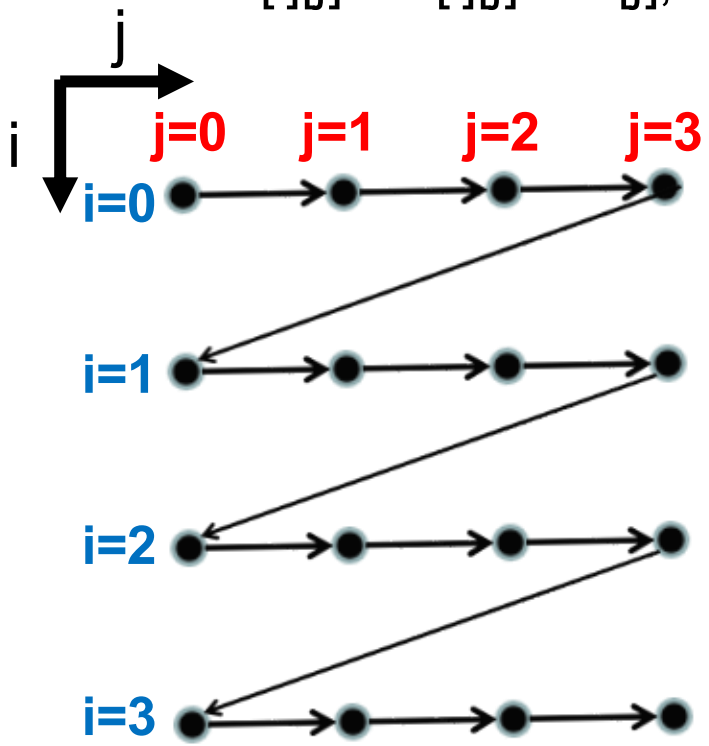
A

```
for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    A[i][j] = A[i][j] + C[j];
```



B

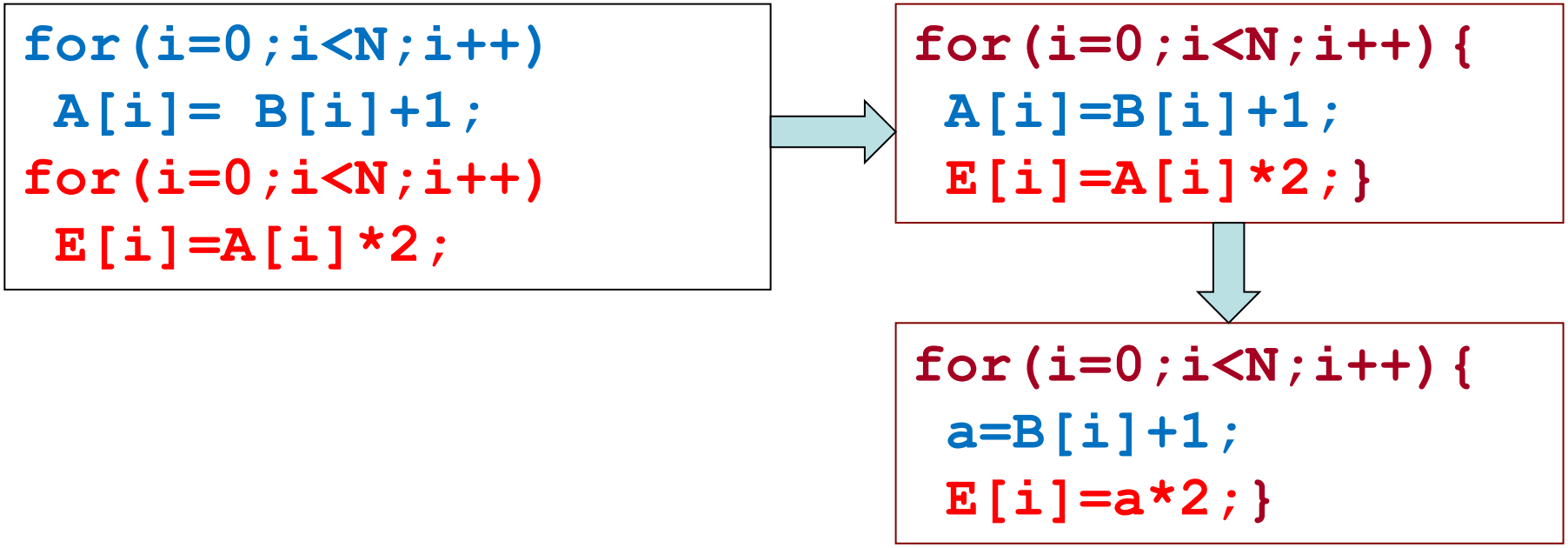
```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    A[i][j] = A[i][j] + C[j];
```



- Loop order of code version B is better: fewer cache misses

Loop Transformation: Fusion

- Loop Fusion: Replace two loops with same range by a single loop



- If cache capacity is less than N words, fusion halves cache misses from 4N/B to 2N/B (B is cache block size)

Loop Transformation: Tiling

```
for(j=0;j<4;j++)
  for (i=0;i<4;i++)
    s += A[j][i]*A[i][j];
```

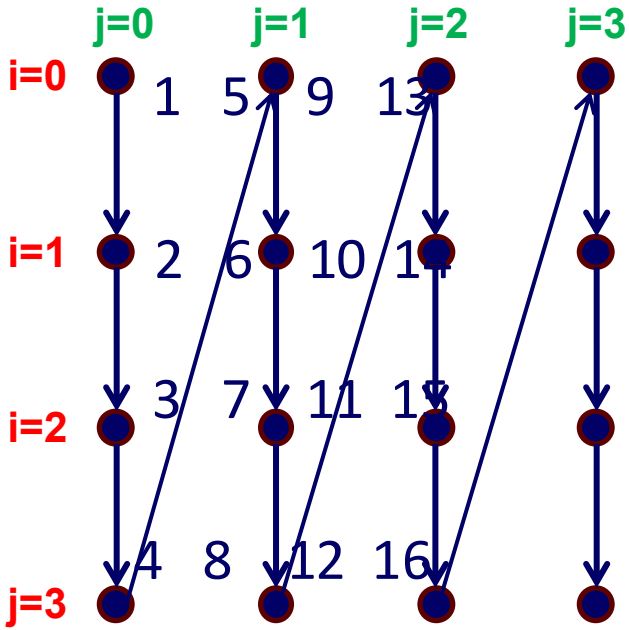
Loop Splitting ↓

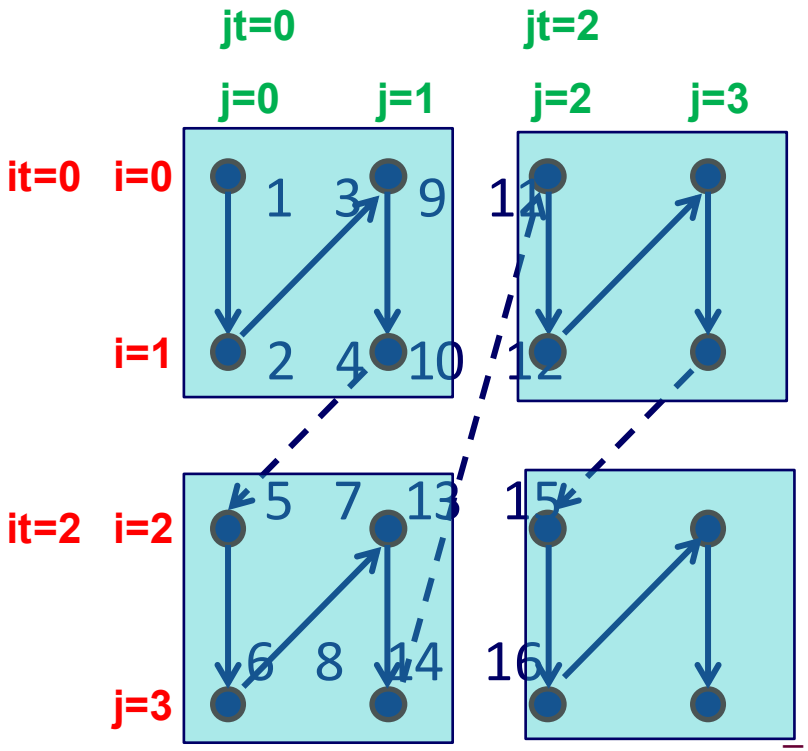
```
for(jt=0;j<4;jt+=2)
 for(j=jt;j<jt+2;j++)
  for (it=0;it<4;it+=2)
   for (i=it;i<it+2;i++)
    s += A[j][i]*A[i][j];
```

Loop Permutation ↓

```
for(jt=0;j<4;jt+=2)
 for (it=0;it<4;it+=2)
  for(j=jt;j<jt+2;j++)
   for (i=it;i<it+2;i++)
     s += A[j][i]*A[i][j];
```

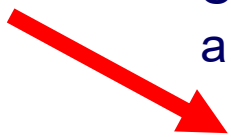
Tiling is a key transform to reduce cache misses





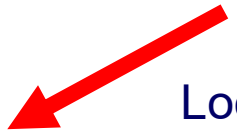
Tiled Matrix-Matrix Multiplication

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j]+= a[i][k]*b[k][j];
```



Split each loop into
a pair of equivalent loops

```
for (it=0; it<N; it+=T)
 for (i=it; i<it+T; i++)
  for (jt=0; jt<N; jt+=T)
   for (j=jt; j<jt+T; j++)
    for (kt=0; kt<N; kt+=T)
     for (k=kt; k<kt+T; k++)
      c[i][j]+= a[i][k]*b[k][j];
```

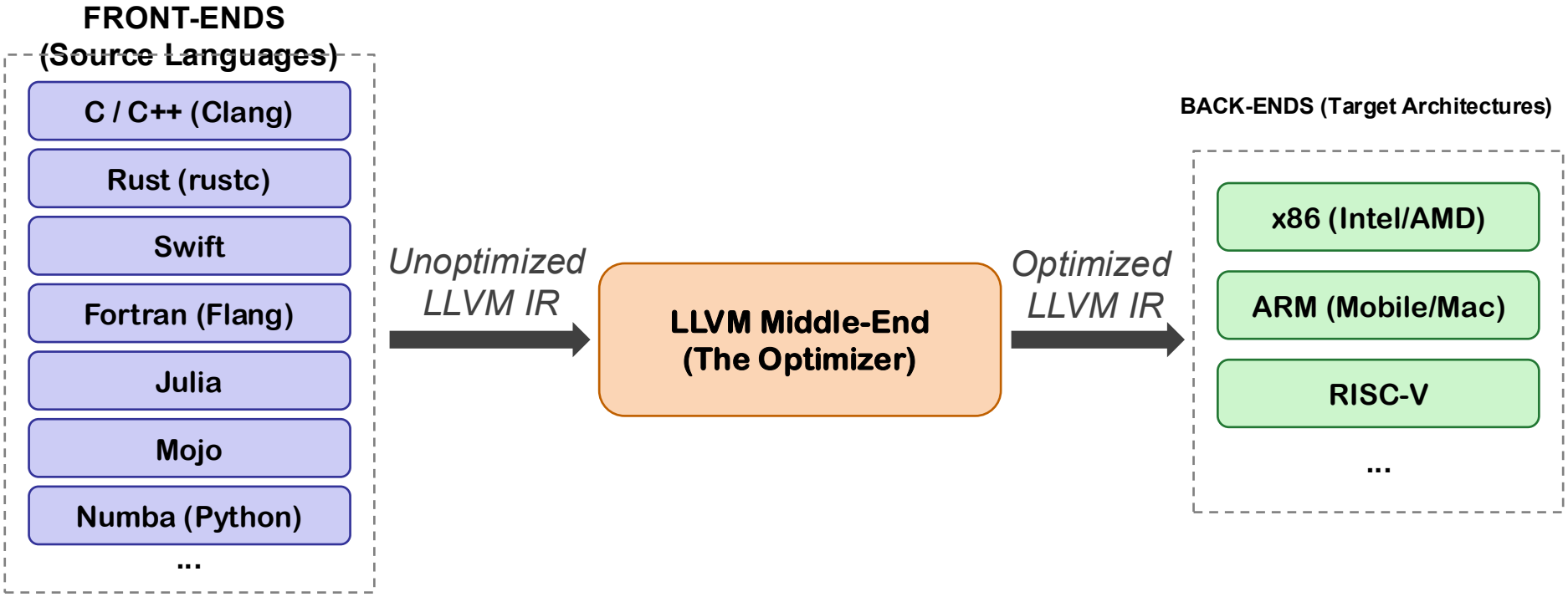


Loop Permutation

```
for (it=0; it<N; it+=T)
 for (jt=0; jt<N; jt+=T)
  for (kt=0; kt<N; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; k++)
      c[i][j]+= a[i][k]*b[k][j];
```

LLVM Overview

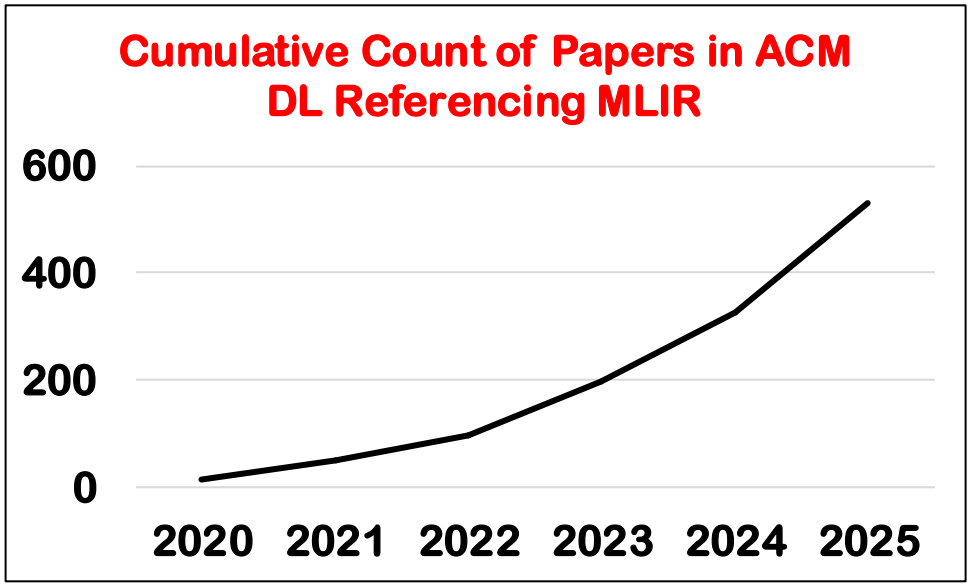
- LLVM: Low Level Virtual Machine
  - Initially developed by Chris Lattner for his M.S. thesis at U. Illinois
  - Currently the most widely used compiler framework
- Abstract low-level IR (Intermediate Representation) at the machine instruction level
  - Many source languages translate to unoptimized LLVM IR (front-end)
  - Common LLVM middle-end optimizations can be used to create optimized LLVM
  - Target-specific back-ends transform the optimized LLVM to assembly code

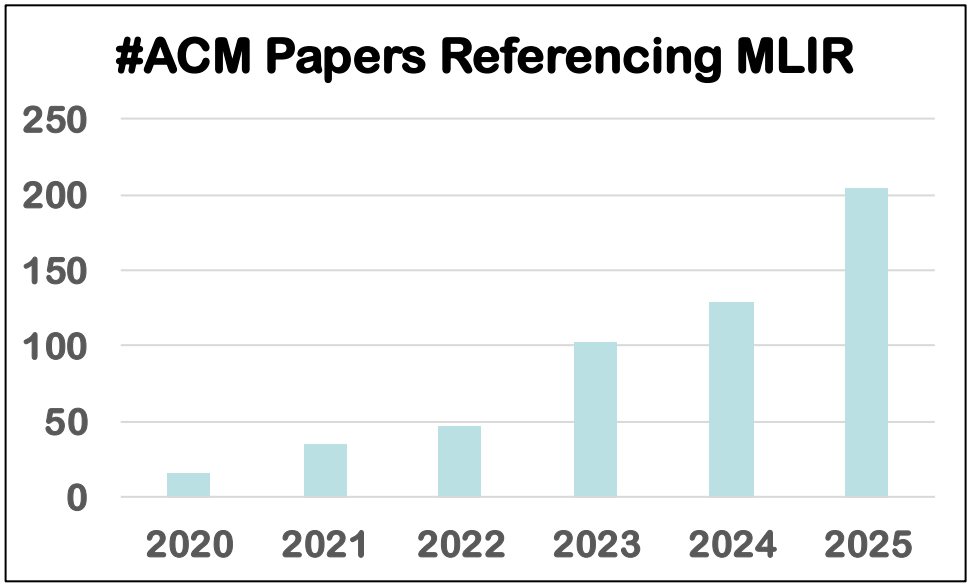


MLIR: History

- Created by Google (2018–2019)
  - Led by Chris Lattner, creator of LLVM (Low Level Virtual Machine)
  - MLIR = Multi-Level Intermediate Representations
  - Open-sourced under the LLVM project in 2019
- Motivation: Explosion of domain-specific compilers for ML
  - Diversifying hardware (GPUs, TPUs, NPUs, custom accelerators)
  - LLVM is widely used but too low-level for many key optimizations
    - Loop-level transformations like tiling and fusion critical for tensor computations
  - Need for reusable compiler infrastructure for higher level optimizations
- Multiple IRs for transformations at different levels of abstraction
  - Formalized via Dialects
    - Dialects define operations and types
    - Transformation passes for optimization and lowering to lower-level dialects
- Rapidly increasing adoption across industry and research

Exponential Rise of Interest in MLIR





LLVM Instructions

```
<result> = <opcode> <type> <operands> [<attributes>]
```

```
%sum = add nsw i32 %a, %b
```

- Low-level instructions at the level of machine assembly
  - <result>: A named SSA (Static Single Assignment) value
    - Example: %sum (all variable names must start with “%”)
  - <opcode>: The operation to be executed
    - Examples: add, load, call, icmp
  - <type>: Operands have their type explicitly specified
    - Example: i32, float, ptr
  - <operands>: One or more input values, each with a type or implied type
    - %a and %b in the above example
  - [<attributes>]: Optional modifiers that refine semantics
    - Examples: nuw, nsw, align, fast
    - nsw: “no signed wrap” [says not to worry about possible overflows]

SSA (Static Single Assignment)

```
int nonssa(int a, int b)
{ int x;
  x = a+b;
  x = x*x;
  return x;
}
```

```
int ssa(int a, int b)
{ int x1,x2;
  x1 = a+b;
  x2 = x1*x1;
  return x2;
}
```

- Both functions return the same result $(a+b)^2$
  - Function **nonssa** reuses variable **x** to replace an older value with a newer value in the second assignment statement
  - Function **ssa** uses different variable names for the two assignments
- All statements in LLVM (and MLIR) use SSA values
  - Each variable can have only one (static) assignment
    - But can be dynamically assigned many times (e.g., in a loop program)
- SSA form is very beneficial for analysis/optimization
  - Examples: Dead code elimination, CSE (Common Subexpression Elimination), Loop optimizations, Register allocation, …

Common Subexpression Elimination

- If an expression is recomputed and its operands are unchanged, compute it once and reuse the result

```
x = a*b + c;
…
…
y = a*b - d;
```

$\Longrightarrow$

```
t = a*b;
x = t + c;
…
…
y = t - d;
```

- Optimization only valid only if a and b are not modified between the two uses
- SSA form makes such reuse easy to spot: every value is defined exactly once

Control Flow with LLVM SSA

- What happens with a conditional expression?

  y = if (x > 0) then x else 0

Non-SSA: y is assigned on both paths

```
entry:
  cmp = (x > 0)
  br cmp, then, else
then:
  y = x            ; definition 1
  br merge
else:
  y = 0            ; definition 2
  br merge
merge:
  ... use y
```

SSA: One definition, merge via "phi" node

```
entry:
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, %then, %else
then:
  br label %merge
else:
  br label %merge
merge:
  %y = phi i32 [ %x, %then ],
               [ 0, %else ]
```

- SSA allows only one definition per value
  - Two assignments to y are illegal
- Special merge instruction (called a phi node) conditionally assigns appropriate value depending on how control flowed

Overview of MLIR

- Modular infrastructure for building compilers
- Builds on key ideas from LLVM and overcomes its limitations
  - MLIR retains SSA form, strong typing, use of attributes, etc. from LLVM
  - MLIR extends LLVM's fixed instruction set to customizable operations and types
  - MLIR extends the control flow model to allow structure/nesting

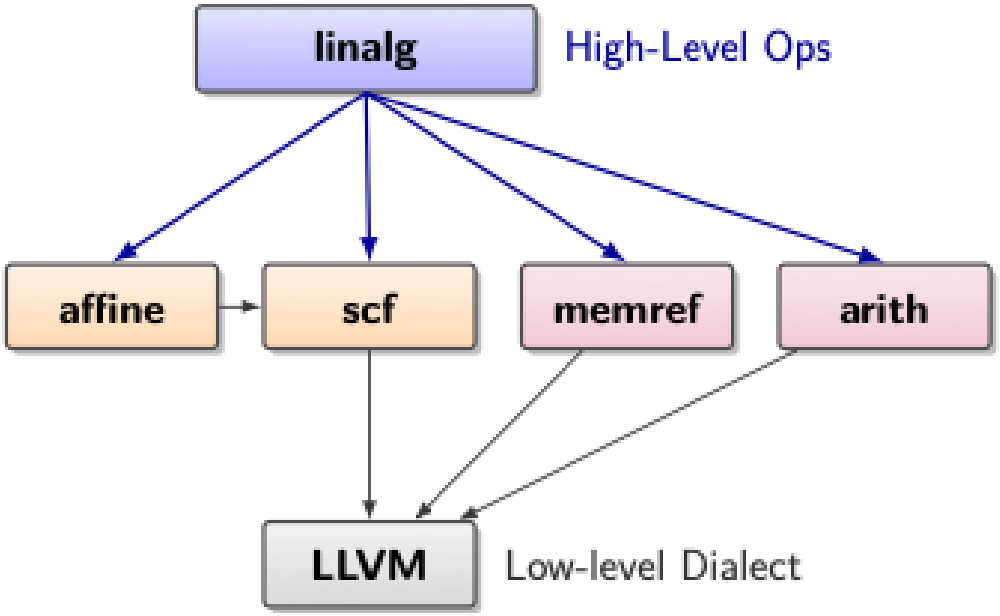
Feature	LLVM IR	MLIR
Instruction set	Fixed	Extensible via dialects
Control flow	CFG only	Structured + CFG
Nested regions	No	Yes
Types	Fixed, low-level	Extensible, high-level
Abstraction levels	One	Many
Custom ops	No	Yes

Overview of MLIR

- Key idea: multiple intermediate representations and progressive lowering from a high-level form through multiple steps
- Dialects define operations and types
  - No “built-in” types in MLIR
  - Predefined **arith** dialect defines various scalar types typically defined in programming languages
- An MLIR program can mix operations from multiple dialects
  - Transformation passes progressively lower the representation
    - Higher-level dialect’s operations rewritten using operations of lower-level dialects
  - Finally, a lower-level dialect (e.g., LLVM dialect) lowers program to machine code of the target hardware platform

Examples of Core MLIR Dialects

- MLIR provides several pre-defined dialects
  - linalg: Represents high-level linear algebra operations (like matmul, conv, or generic structured ops)
  - affine: Provides powerful abstractions for loops and memory accesses that enable polyhedral optimizations (like tiling and fusion).
  - scf (Structured Control Flow): Captures control flow logic using high-level regions (like for, if, and while) rather than low-level branch instructions.
  - arith: Defines the standard set of basic integer and floating-point arithmetic operations.
  - memref: Models mutable memory buffers with support for complex layouts
  - LLVM: An MLIR dialect that can directly interface with LLVM to generate code for various hardware target platforms



MLIR Program Structure

- MLIR Program: IR of a source program in some DSL, to be transformed + lowered
- Key "structural" concepts for MLIR: Operation, Region, Block
  - Nested recursive structure of MLIR program with these three units
- Operation: Fundamental unit of MLIR; everything is an "operation"
  - Examples at different levels of granularity
    - a function (func.func), a loop (scf.for), a constant (arith.constant)
    - dialect-specific operations, e.g., linalg.matmul, gpu.launch
- Region: Scoped container for blocks; has one or more blocks
  - Operations may have zero, one, or multiple regions within it
    - arith.addi has 0, func.func has 1, scf.if has multiple regions ("then" and "else" parts)
- Block: Sequence of operations
  - Has arguments (SSA values) and terminator op
  - Blocks are the primary mechanism to implement arbitrary control flow

```
%results:2 = "d.operation"() ({
  // Regions belong to Ops and can have multiple blocks.
  ^block(%argument: !d.type):
    %value = "nested.operation"() ({
      // Ops can contain nested regions.
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:
    "d.terminator"() [^block(%arg0 : !d.type)] : () -> ()
}) : () -> (!d.type, !d.other_type)
```

Region

Block

Region

Source: Mehdi Amini

Control Flow with MLIR SSA

- Same example: LLVM SSA vs. MLIR SSA

  y = if (x > 0) then x else 0

LLVM SSA: "phi" node ”PULL”

```
entry:
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, %then, %else
then:
  br label %merge
else:
  br label %merge
merge:
  %y = phi i32 [ %x, %then ],
               [ 0, %else ]
```

MLIR: Block arguments replace phi

```
%c0  = arith.constant 0 : i32
%cmp = arith.cmpi sgt,%x,%c0:i32
cf.cond_br %cmp, ^then, ^else

^then:
  cf.br ^merge(%x : i32)
^else:
  cf.br ^merge(%c0 : i32)
^merge(%y: i32):
```

- Block arguments in MLIR replace phi nodes in LLVM
  - Block entry at merge point declares parameter; each branch passes values