SIMD and Vector Architecture

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024

Basate su:

Ch.4 Computer Architecture A Quantitative Approach, 6th Edition

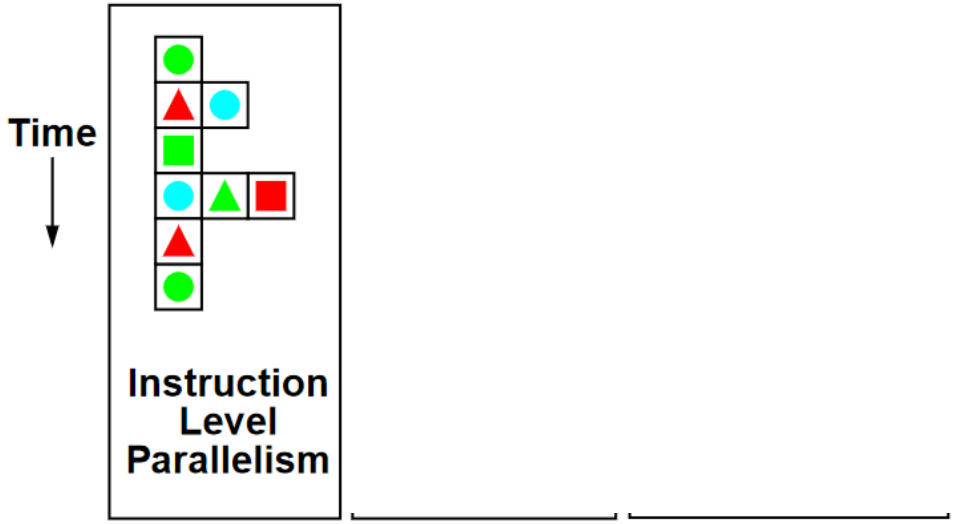
Mauro Olivieri, Sapienza, The EPI vector acceleration processor. ACM European Summer School, Sept. 2021

Matheus Cavalcante, Vector Processing in Modern ISAs

Parallelism

- Computer architects exploit **parallelism** to provide increases in computing performance through architectural choices.

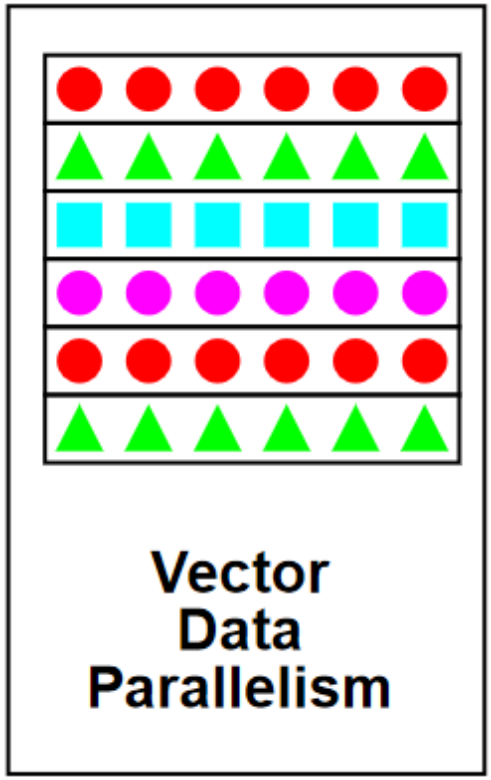
- Simplest (and ubiquitous) form of parallelism: pipelining
- Beyond pipelining:
  - Instruction level parallelism (ILP)
  - Thread level parallelism (TLP)
  - Data level parallelism (DLP)



Krste Asanovic, Vector Microprocessors.

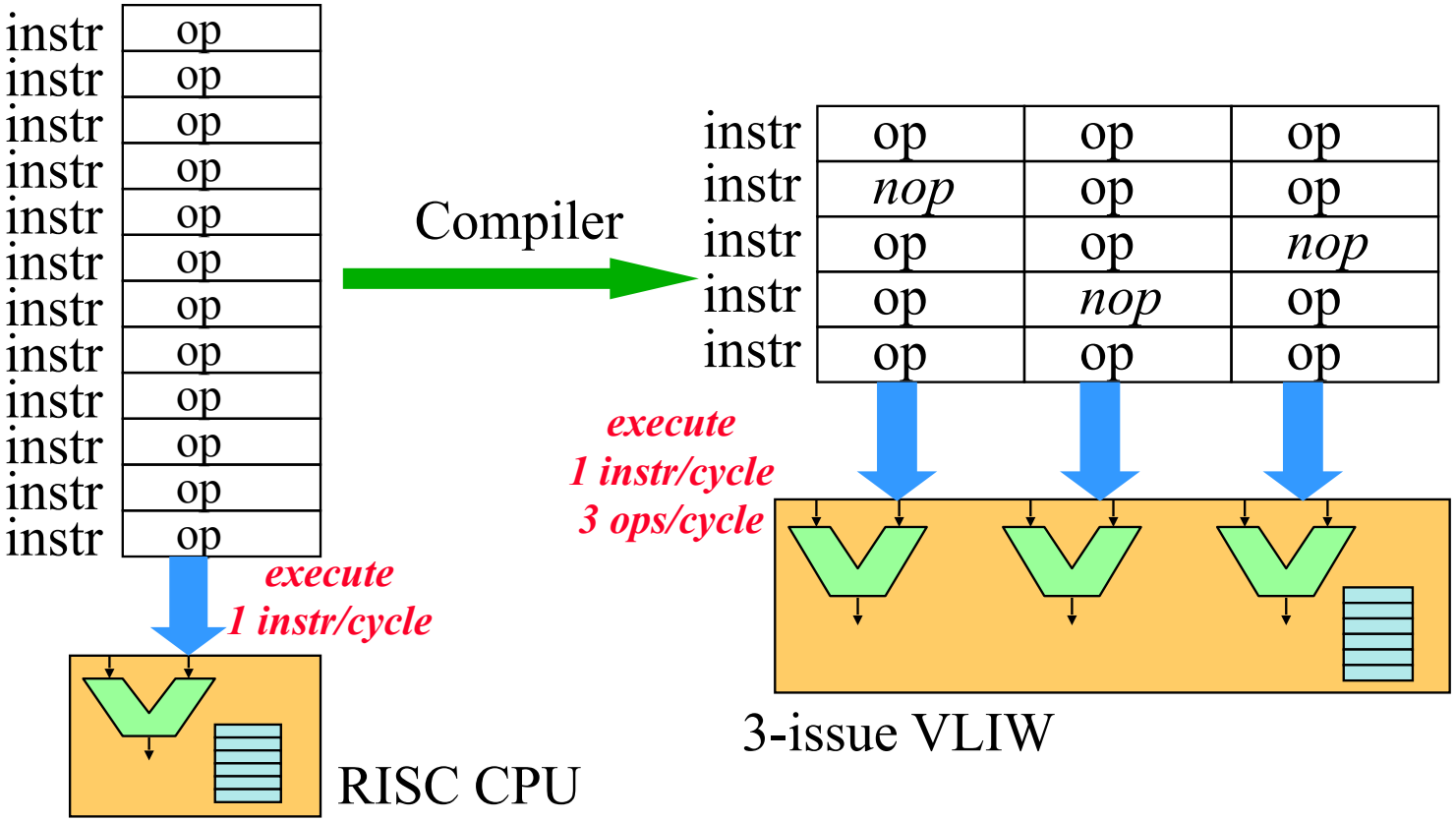
Data Parallelism

- The least flexible form of parallelism
  - Anything that can be expressed to exploit DLP, can also be written to exploit ILP or TLP
- The cheapest form of parallelism
  - A machine **that exploits DLP** only needs to fetch and decode a single instruction to describe a whole array of vector operations
  - Energy efficient!
    - If you can program it…

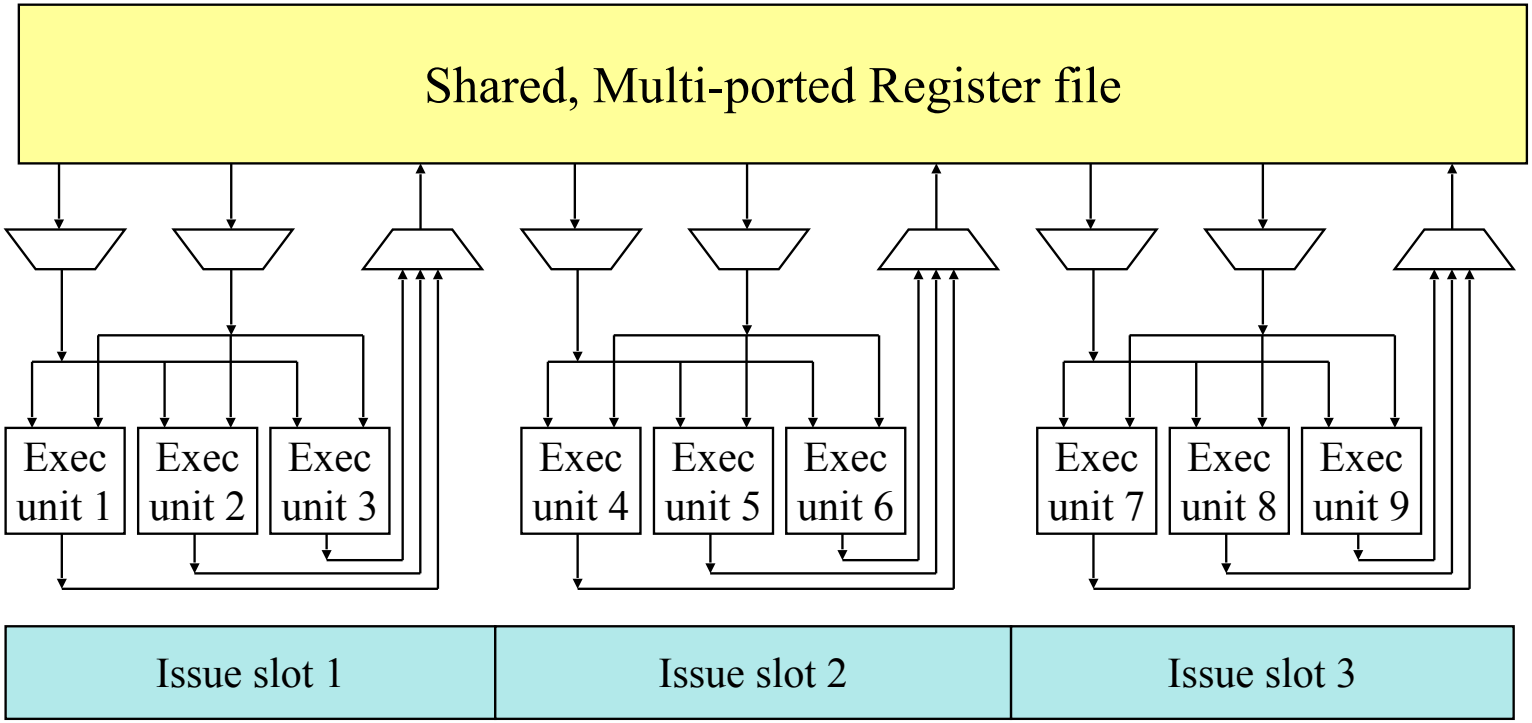


VLIW: vector HW without vector ISA

- VLIW: Multiple independent operations packed together by the compiler



VLIW architecture: central Register File



Q: ***How many ports does the register file need for n-issue?***

Why SIMD?

- Single 32-bit Microprocessor
  - One 32-bit element per instruction
- 512-bit SIMD processor
  - **16** 32-bit elements per instruction
  - **32** 16-bit elements per instruction
  - **64** 8-bit elements per instruction. If the SIMD processor clock frequency is at 1 GHz, then the performance is 64 GOPS
- Increasing the SIMD size, the higher the performance
- SIMD is the basic for Vector Processor
  - N operations per instruction where the elements in the operations are independent from each other

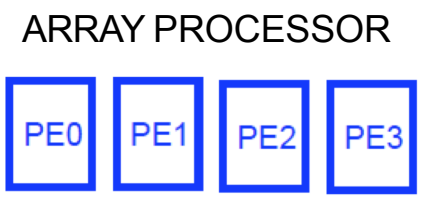
Exploiting Data Parallelism: SIMD

- Concurrency arises from performing **the same operation on different pieces of data**
  - **Single-Instruction Multiple-Data (SIMD)**

- Single instruction operates on multiple data elements
  - Multiplexed in time or in space

- Time-space duality:
  - **Array processor:** instruction operates on multiple data elements at the same time using different spaces
  - **Vector processor:** instruction operates on multiple data elements in consecutive data steps using the same space
  - In reality, hybrids between array and vector processors are more commonly found today.

Array vs. Vector Processors



Instruction Stream

```
ld      vr <- a[3:0]
add     vr <- vr, 1
mul     vr <- vr, 2
st  a[3:0] <- vr
```

Intel MMX, SSE, SSE2, AVX

- **MMX** (1997) adds 57 new **integer** instructions, **64-bit** registers
  - 8 of 8-bit, 4 of 16-bit, 2 of 32-bit, or 1 of 64-bit (reuse 8FP registers – FP and MMX cannot mix)
  - Number of elements is defined in the opcode, only consecutive memory load/store is allowed
  - Move, Add, Subtract, Shift, Logical, Mult, MACC, Compare, Pack/Unpack
- **SSE** (1999) adds 70 new instructions for single precision **FP**, 2 of 32-bit
- **SSE2** (2001) adds 144 new instructions for FP, 4 of 32-bit, **128-bit** registers
- **SSE3** (2004), SSE4 (2006), SSE5 (2007) add more new instructions
- **AVX** (2008) adds new instructions for **256-bit** registers
- **AVX2** (2013) adds more new instructions
- **AVX-512** (2015) adds new instructions for **512-bit** registers
- Issue:
  - Each register length has a new set of instructions
  - Many instructions for the 512-bit SIMD

Terminology

- ISA: Instruction Set Architecture
- GOPS: Giga Operations Per Second
- GFLOPS: Giga Floating-Point OPS
- **XRF**: Integer register file
- FRF: Floating-point register file
- **VRF**: Vector register file
- SIMD: Single Instruction Multiple Data
- MMX: Multi Media Extension
- SSE: Streaming SIMD Extension
- AVX: Advanced Vector Extension
- CSR: Control and Status Register
- **SEW**: Element Width (8-64)
- ELEN: Largest Element Width (32 or 64)
- **XLEN**: Scalar register length in bits (64)
- FLEN: FP register length in bits (16-64)
- **VLEN**: Vector register length in bits (128-512)
- **LMUL**: Register grouping multiple (1/8-8)
- EMUL: Effective LMUL
- **VLMAX**/MVL: Vector Length Max
- AVL/**VL**: Application Vector Length
- **Configurable**: parameters are fixed at built time, i.e. cache size
- **Extensible**: added instructions to ISA includes custom instructions to be added by customer
- **Standard extension**: the reserved codes in the ISA for special purposes, i.e. FP, DSP, …
- **Programmable**: parameters can be dynamically changed in the program

Vector Processors (I)

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i <= 49; i++)
  C[i] = (A[i] + B[i]) / 2;
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance in memory between two elements of a vector

Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element
- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking needed within a vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables prefetching of vectors into registers/cache/memory

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining & parallelization work really well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements

+ Highly regular memory access pattern

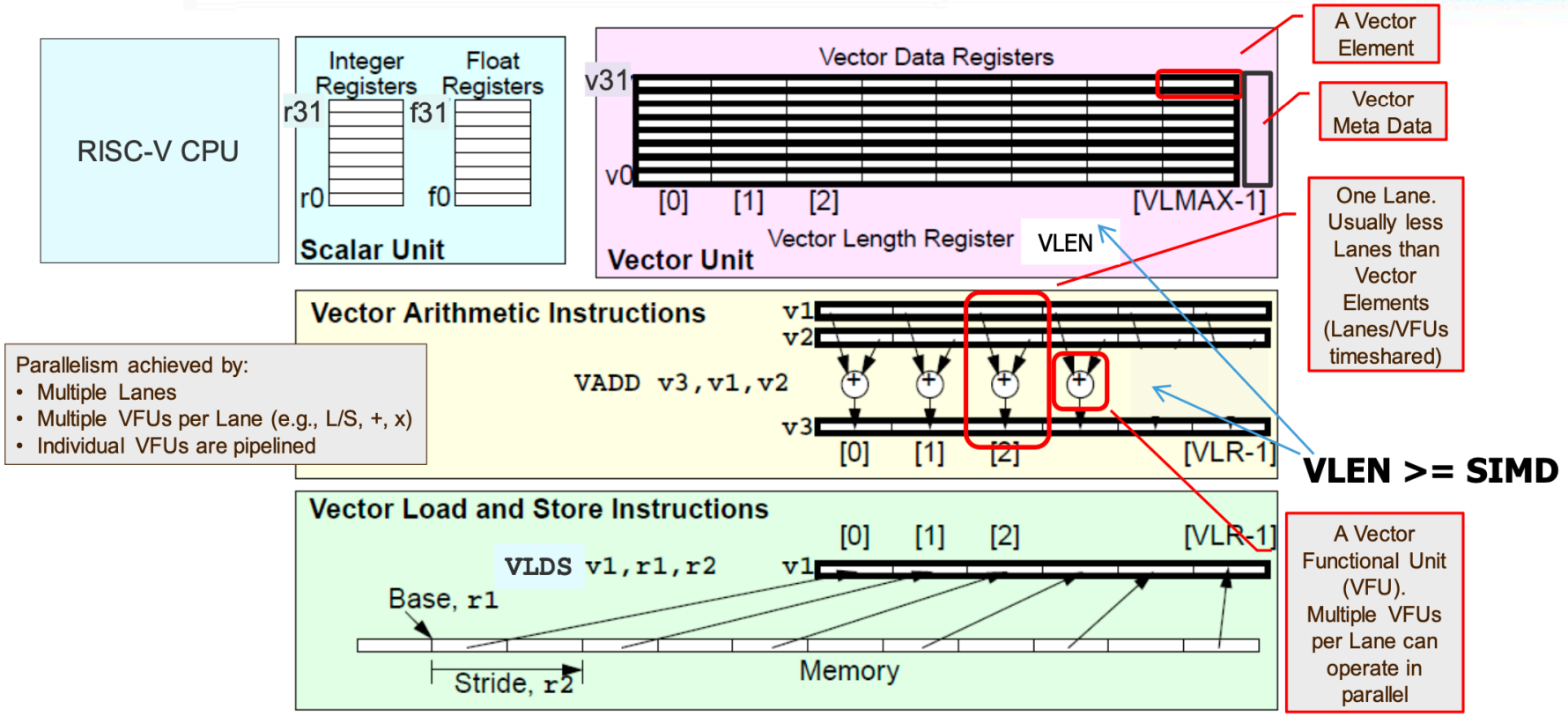
+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
    - “vector registers” ≈ large sequential register
  - Operate on data on those registers – single operation on many data
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Architectures - programming model



Newell, R., "RISC-V Vector Extension Proposal Snapshot," Microsemi, Microchip Technology Inc., 2018

Vector Programming Model

Two main advantages with respect to the classical Scalar Processing:

1. Utilizes deeply pipelind ALUs to operate on multiple elements at the same time, without data hazards;
2. when it is possible, is able to exploit the data paralellism of the on-going task, eliminating all the standard branch-control that would be massively present otherwise. Fewer instructions to express all the parallelism.

```
for (i=0;i<64;i++)
        C[i]=A[i]+B[i];
```

Scalar Code:

```
li x4, 64
loop:
fld f1, 0(x1)
fld f2, 0(x2)
fadd.d f3,f1,f2
fsd f3, 0(x3)
addi x1, 8
addi x2, 8
addi x3, 8
subi x4, 1
bnez x4, loop
```

9 instr x 64 times

Vector Code:

```
li x4, 64
setvl x4
vld v1, x1
vld v2, x2
vadd v3,v1,v2
vst v3, x3
```

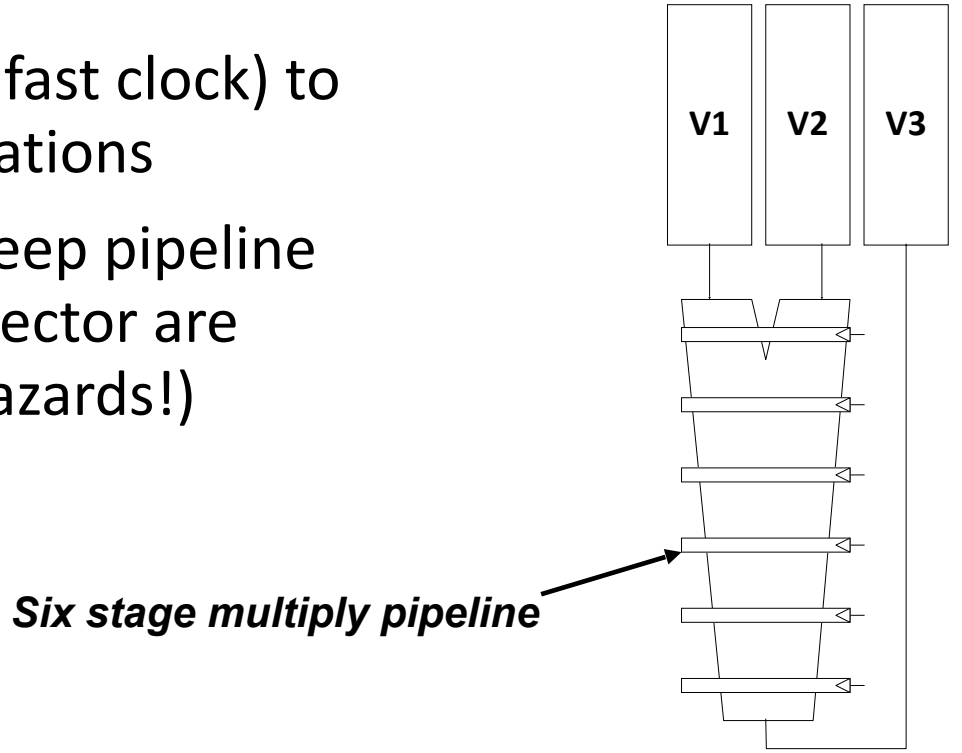
Only 6 instr.

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same object code on more parallel pipelines or lanes

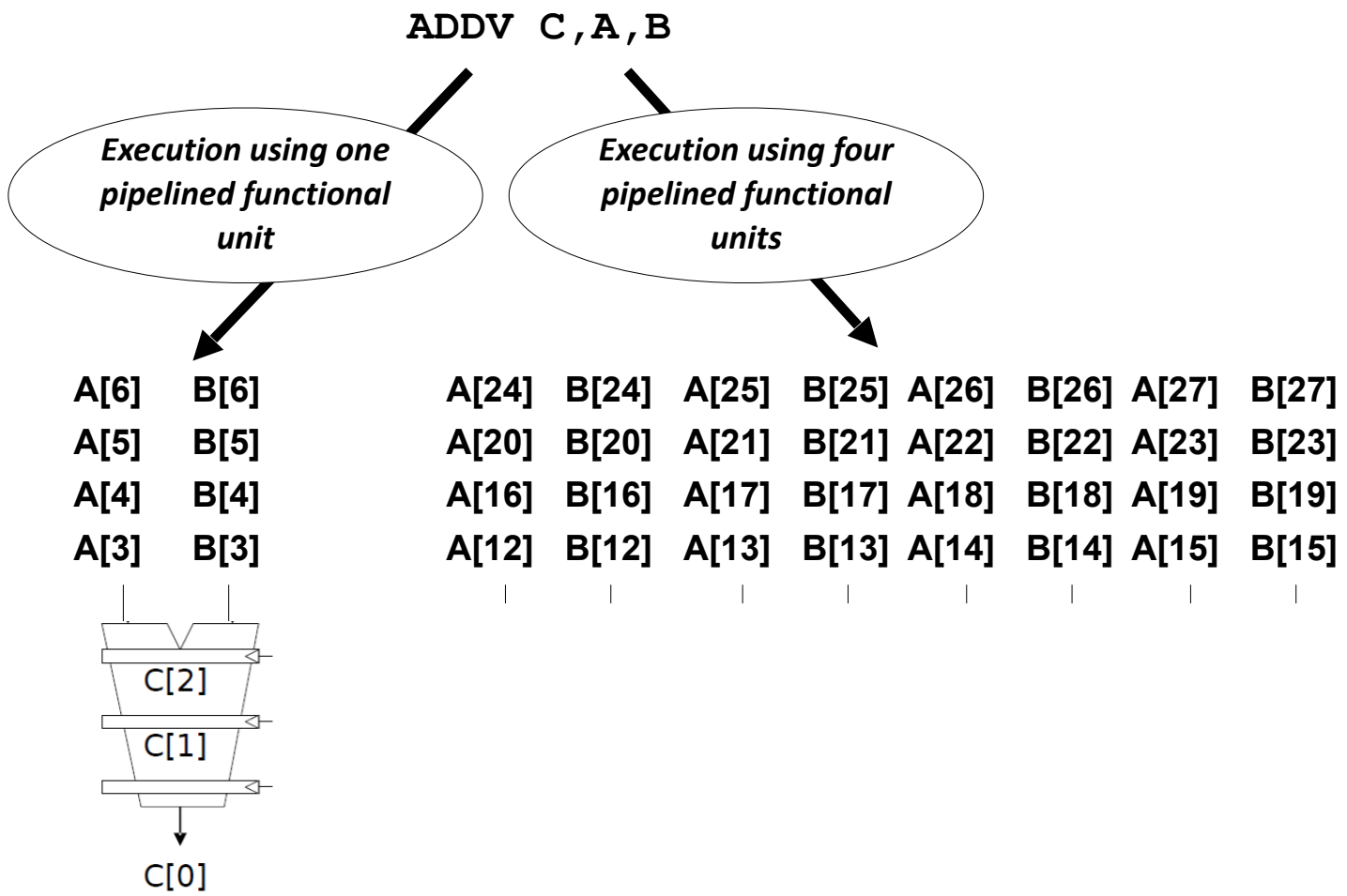
Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

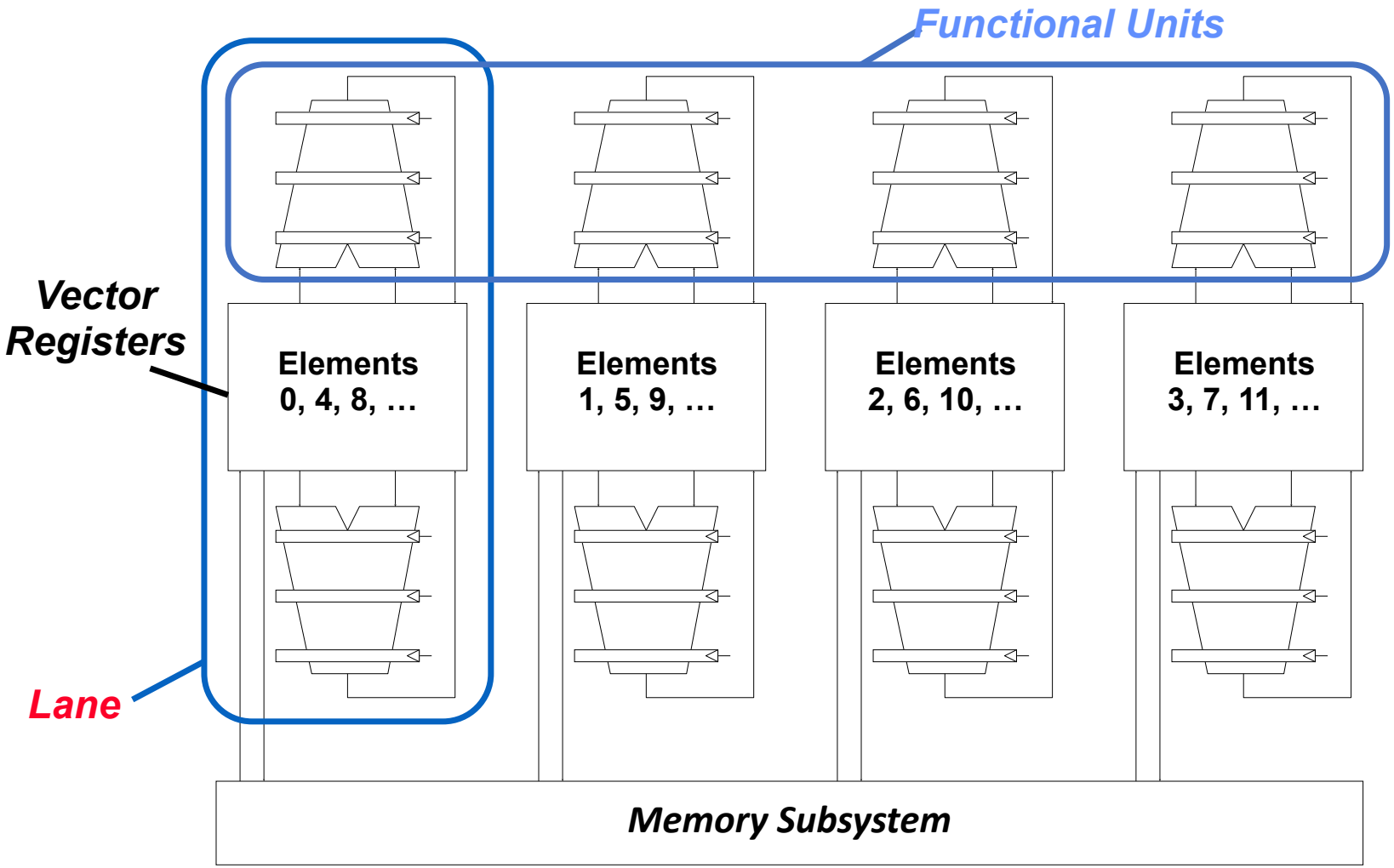



```
V3 <- v1 * v2
```

Vector Instruction Execution

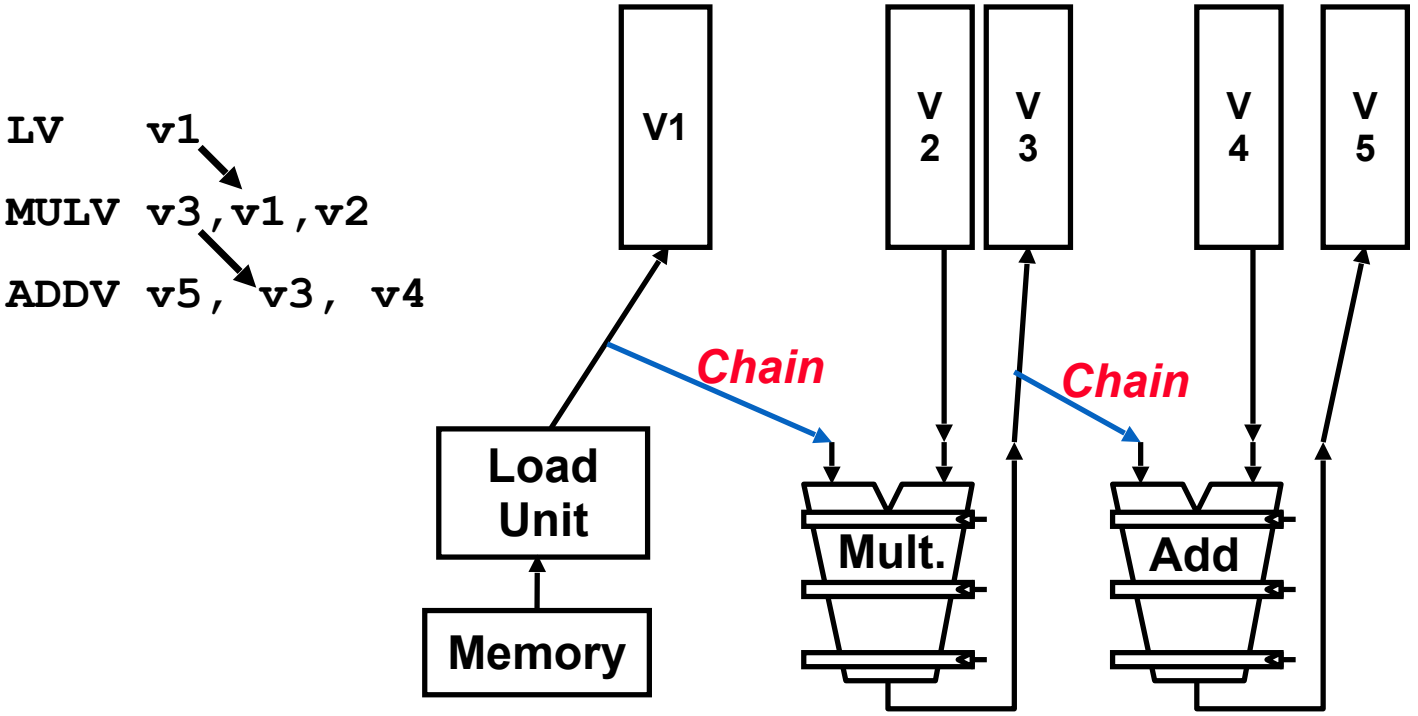


Vector Unit Structure



Vector Chaining

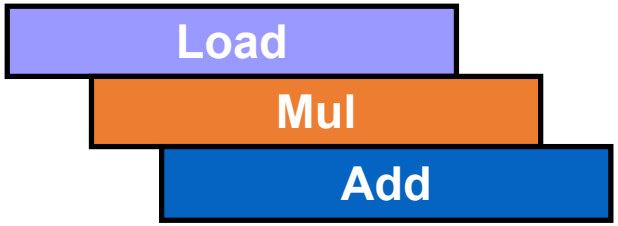
- Vector version of register bypassing
  - introduced with Cray-1



Vector Chaining Advantage

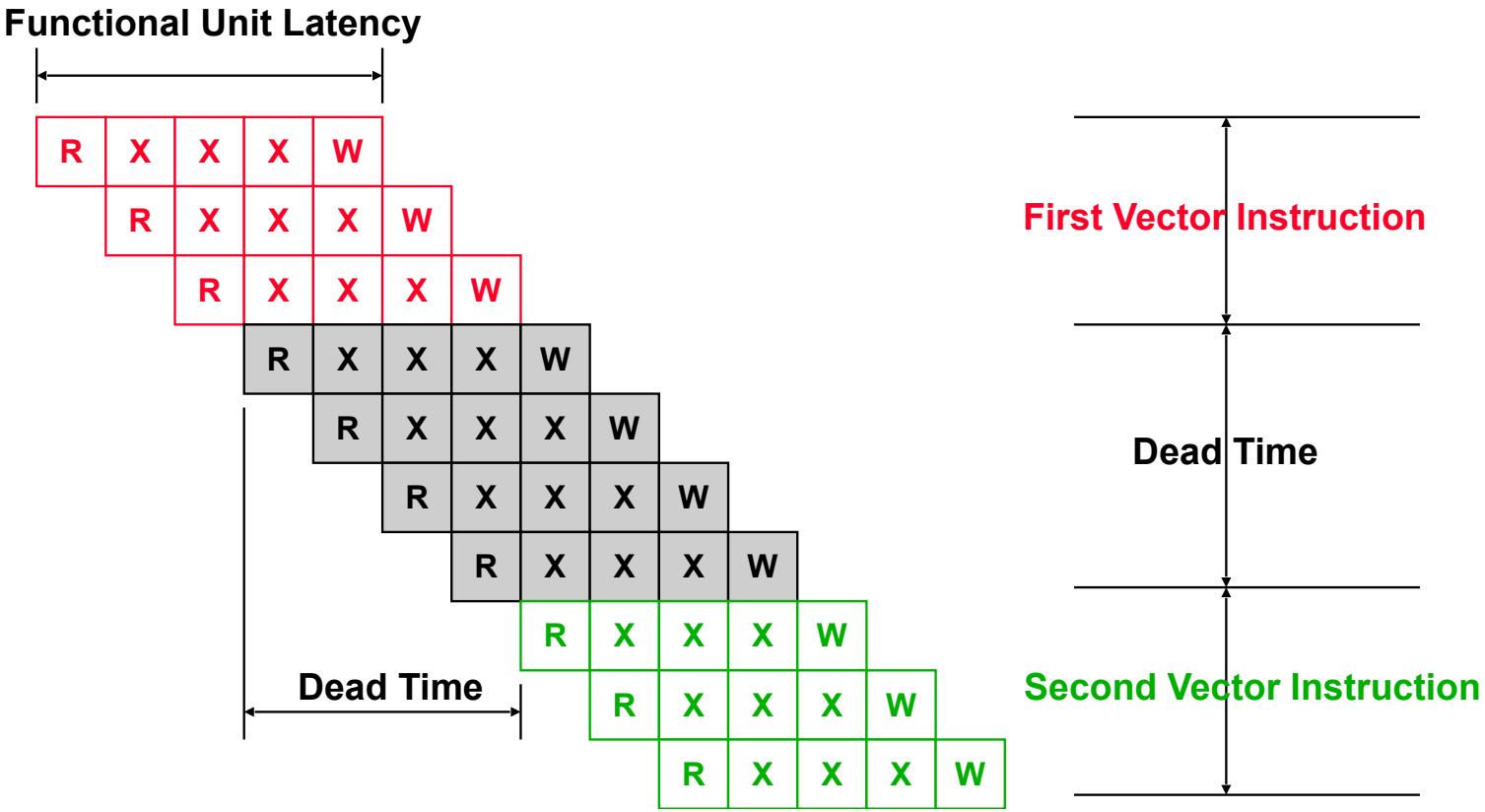
Without chaining, must wait for last element of result to be written before starting dependent instruction

With chaining, can start dependent instruction as soon as first result appears



Vector Startup

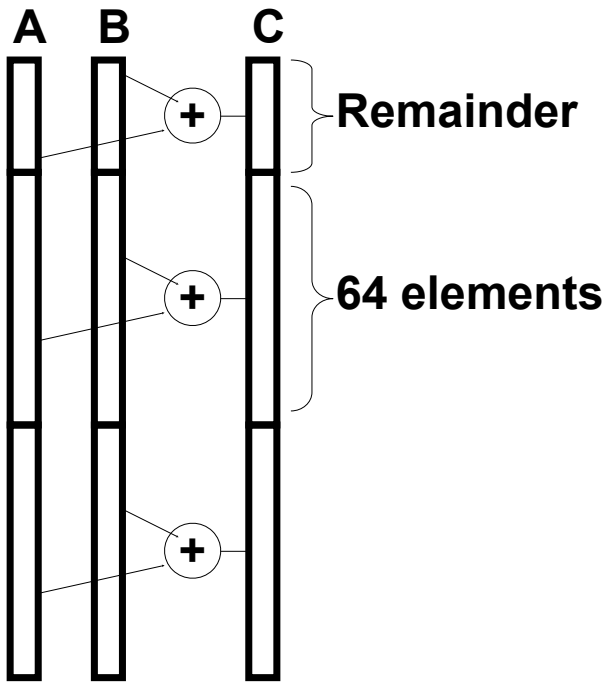
- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline)



Question

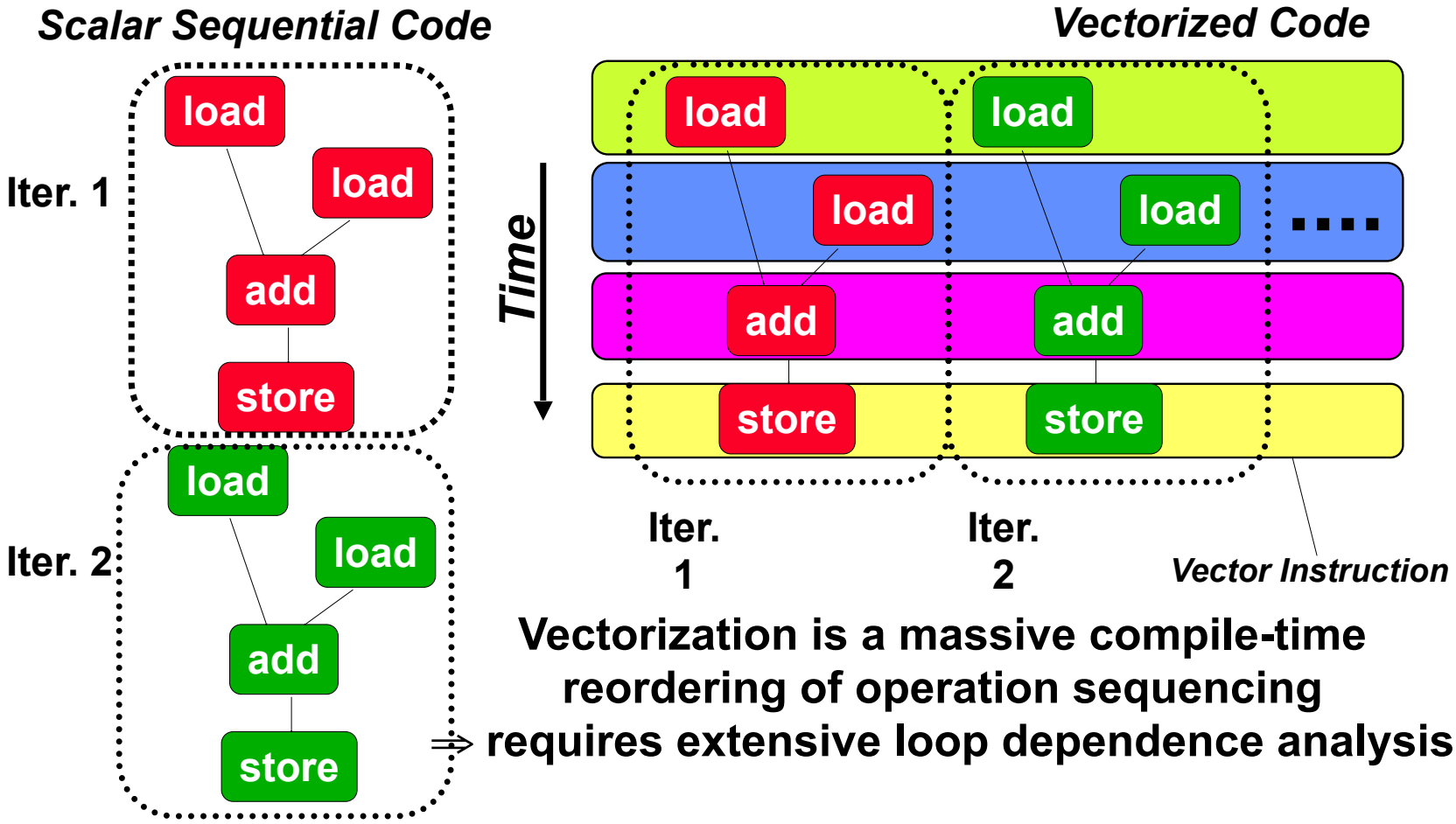
- What if # data elements > # elements in a vector register?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

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



Automatic Code Vectorization

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



Vectorization is a massive compile-time reordering of operation sequencing ⇒ requires extensive loop dependence analysis

Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - » Contiguous block of information in memory
    - » Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - » Harder to optimize memory system for all possible strides
    - » Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - » Vector equivalent of register indirect
    - » Good for sparse arrays of data
    - » Increases number of programs that vectorize

Vector Scatter/Gather

- Want to vectorize loops with indirect accesses:
  - `for (i=0; i<N; i++)`
  - `    A[i] = B[i] + C[D[i]]`
- Indexed load instruction (Gather)
  - `LV vD, rD          # Load indices in D vector`
  - `LVI vC, rC, vD     # Load indirect from rC base`
  - `LV vB, rB          # Load B vector`
  - `ADDV.D vA, vB, vC  # Do add`
  - `SV vA, rA          # Store result`

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

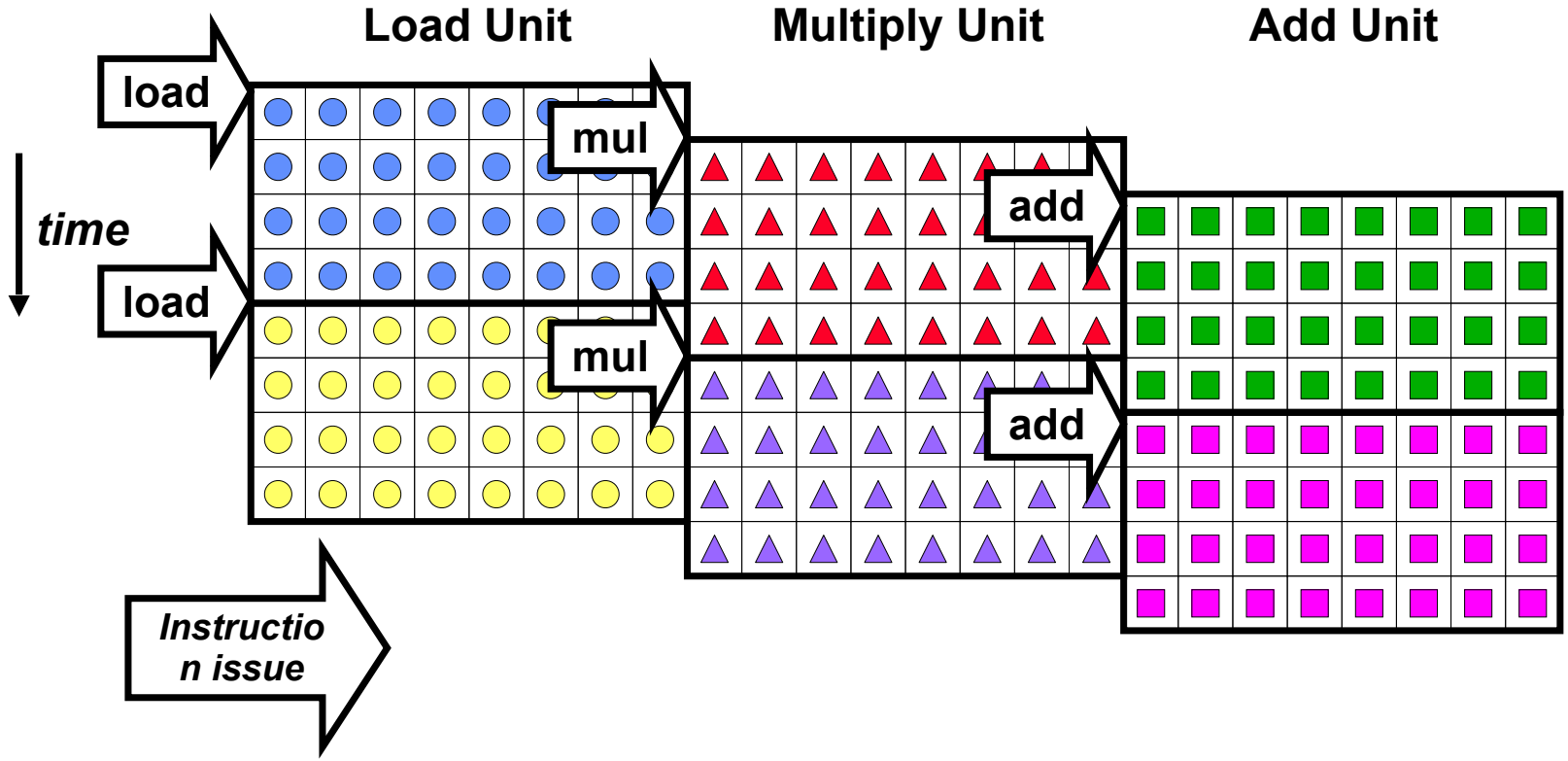
Code example:

```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

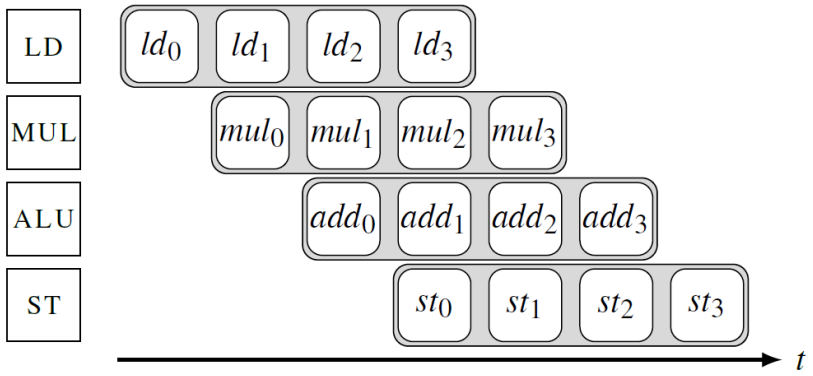
- **example machine has 32 elements per vector register and 8 lanes**



Complete 24 operations/cycle while issuing 1 short instruction/cycle

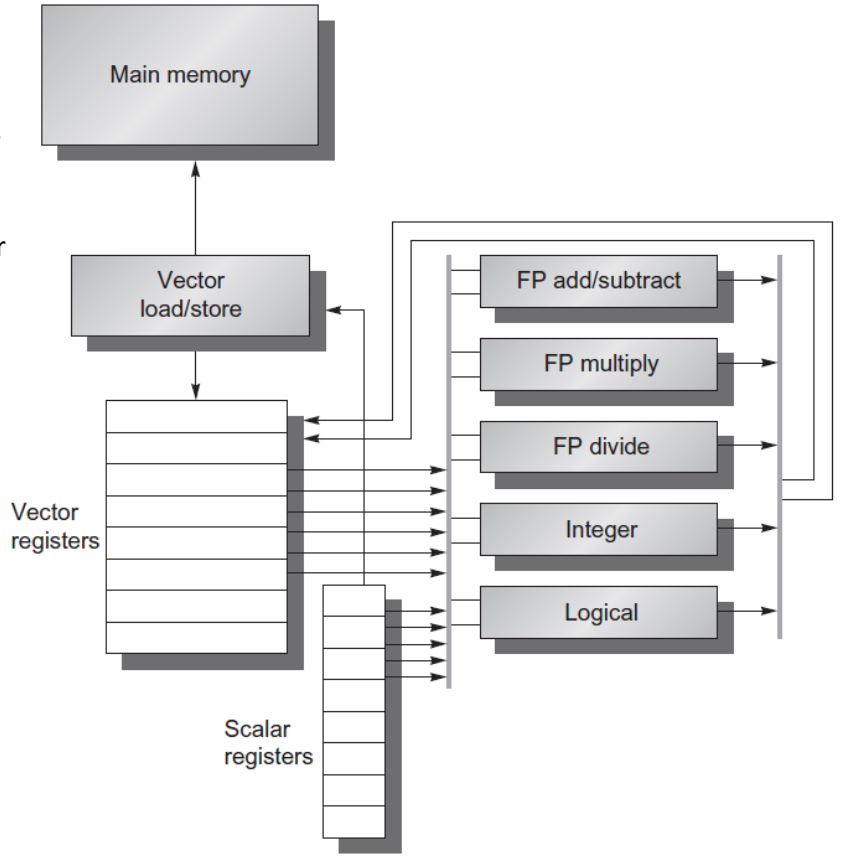
Vector-SIMD Execution Model

- Renaissance of (Cray-like) vector processing during the last few years
- Quest for the energy efficiency and high performance promised by vector architectures
- Main ISAs now include a vector processing extension
  - ARM SVE (Scalable Vector Extension)
  - RISC-V Vector Extension



RV64V (Vector extension)

- Loosely based on Cray-1
- 32 *-bit vector registers (* could be 32,64, …)
  - Vector data type is property of the vector register and not of the instruction. Same opcode for multiple datatypes
  - Can accommodate multiple vector lengths
  - Before executing the vector instruction, the program needs to configure the vector register width and data type.
  - Dynamic Register Typing: Vector registers can be disabled, can be joined in groups.
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
  - Inputs can be vector or scalar.
    - Encoded in the opcode (.vv, .vs, .sv)
    - Scalar value read at issue time
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers (the one in RV64G)
  - 31 general-purpose registers
  - 32 floating-point registers



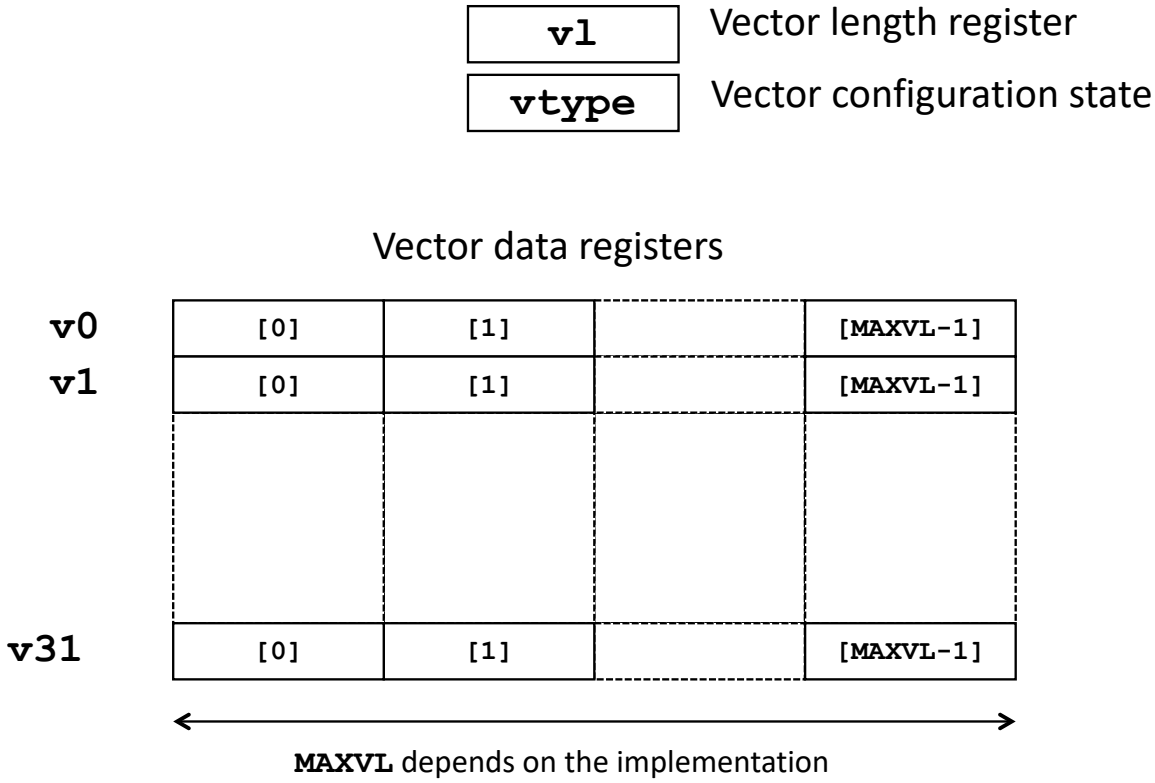
RISC-V Vector Standard

- Being added as a standard extension to the RISC-V ISA
- An updated form of Cray-style vectors for modern microprocessors
- The official RV vector extension is RVV1 and available at https://github.com/riscvarchive/riscv-v-spec

Quick summary of RISC-V Vector ISA

- 32 vector registers, **`v0-v31`**
- The vector type **`vtype`** CSR provides the default type to interpret the elements of the vector register
  - Standard Element Width (SEW):
    - 8, 16, 32, 64-bit…
  - Length multiplier (LMUL):
    - Multiple vector registers can be grouped together, so that a single vector instruction operates on multiple registers
    - LMUL can be 1, 2, 4, 8 or a fraction
- Vector length register **`vl`** controls the number of elements executed by each instruction
- Instructions can be predicated by a mask
- Vector memory operations include unit-stride, constant-stride and scatter-gather

Vector Unit State



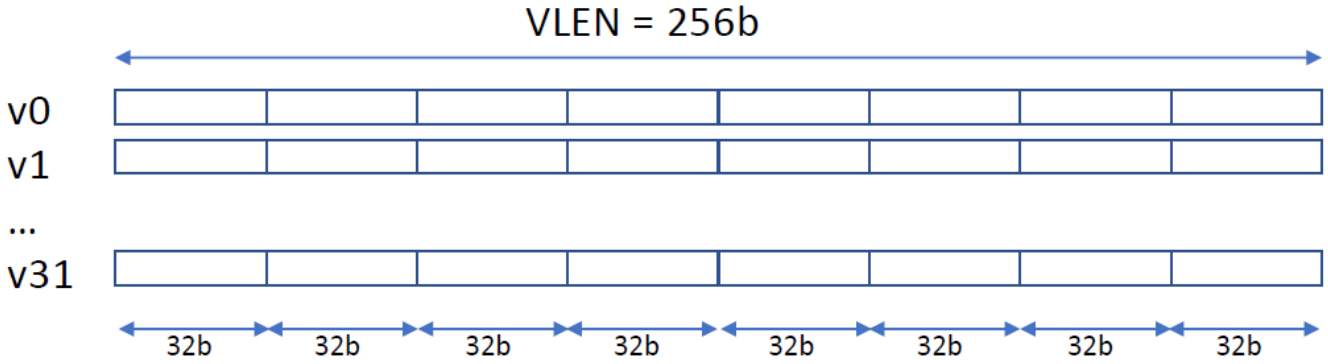
Register State: 32 registers of VLEN bits

- 32 register names: v0 through v31
- Each register is VLEN bits wide
  - VLEN is chosen by implementation, must be power of 2
    - See spec for additional restrictions in relation to ELEN and SLEN
- Some control registers
  - VL = active vector length
  - SEW = standard element width, hosted in vsew [2:0]
  - LMUL = grouping multiplier

SEW determines number of elements per vector

- SEW = Standard Element Width
- Dynamically settable through *‘vsew [2:0]’*
- Each vector register viewed as VLEN/SEW elements, each SEW bits wide
- Polymorphic instruction
- vadd can be an i8/i16/i32/… add depending on SEW
- Set up along with VL (vsetvli t0, a0, e32)

Example: VLEN=256b, vsew=‘010, SEW=32b, elements = VLEN/SEW = 8



Vector configuration

- Can be done through the **`vsetvli`** instruction
- Vector length depends on the element width and length multiplier
  - It should be possible to write assembly code without knowing MAXVL

- **`vsetvli rd, rs1, vtypei`**
  - **`vl`** is set to **`min(MAXVL, rs1)`**, the value is also copied to **`rd`**
  - **`vtype`** is set to **`vtypei`**

  - Example, setting the **`vl`** to 16 elements of 32 bits (if **`MAXVL >= 16`**)
    - **`li t0, 16`**
    - **`vsetvli t1, t0, e32`**

Example: vector-vector add

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

Register **a0** holds **N**

Register **a1** holds **@A[0]**

Register **a2** holds **@B[0]**

Register **a3** holds **@C[0]**

```
stripmined_loop:
  vsetvli t0, a0, e64 # t0 holds amount done
  vle64.v v0, (a1)    # Load strip of vector A
  vle64.v v1, (a2)    # Load strip of vector B
  vadd.vv v2,v0,v1    # Add vectors
  vse64.v v2, 0(a3)   # Store strip of vector C
  slli t1,t0,3        # Multiply t0 by 8 to get bytes
  add a1,a1,t1        # Bump pointers
  add a2,a2,t1
  add a3,a3,t1
  sub a0,a0,t0        # Subtract amount done
  bnez a0, stripmined_loop
```

Example: vector-vector floating-point add

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

Register **a0** holds **N**

Register **a1** holds **@A[0]**

Register **a2** holds **@B[0]**

Register **a3** holds **@C[0]**

```
stripmined_loop:
  vsetvli t0, a0, e64 # t0 holds amount done
  vle64.v v0, (a1)    # Load strip of vector A
  vle64.v v1, (a2)    # Load strip of vector B
  vfadd.vv v2,v0,v1   # Add vectors
  vse64.v v2, 0(a3)   # Store strip of vector C
  slli t1,t0,3        # Multiply t0 by 8 to get bytes
  add a1,a1,t1        # Bump pointers
  add a2,a2,t1
  add a3,a3,t1
  sub a0,a0,t0        # Subtract amount done
  bnez a0, stripmined_loop
```

Vectorization:

Example

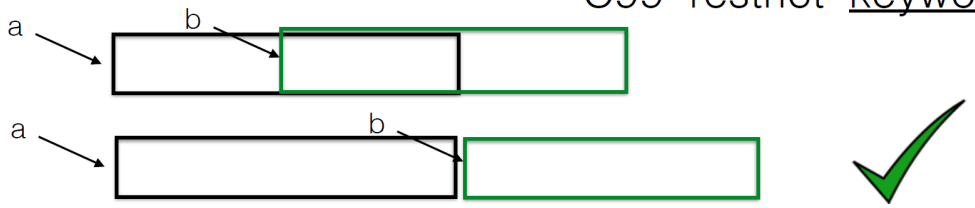
```
void add (float *c, float *a, float *b)
{
    for (int i=0; i <= N; i++)
       c[i]=a[i]+b[i];
}
```

Is not legal to automatically vectorise this loop in C/C++ (without more information)

So, using a compiler switch for auto-vectorisation won't help

#1 Give compiler hints

C99 "restrict" keyword




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

During each execution of a function body in which a restricted pointer P is declared, if some object that is accessible through P is modified, then all accesses to that object in that block must occur through P, otherwise the behaviour is undefined

#2 ignore vector dependencies

OpenMP 4.0 pragmas

Option 1:

```
void add (float *c, float *a, float *b)
{
 #pragma omp simd
    for (int i=0; i <= N; i++)
       c[i]=a[i]+b[i];
}
```

Indicates that the loop can be transformed into a SIMD loop
(i.e. the loop can be executed concurrently using SIMD instructions)

Option 2:

```
#pragma omp declare simd
void add (float *c, float *a, float *b)
{
       *c=*a+*b;
}
```

"declare simd" can be applied to a function to enable
SIMD instructions at the function level from a SIMD loop

#3 code explicitly for vectors

ivdep pragma # Intel specific

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

IVDEP (Ignore Vector DEPendencies) compiler hint.
Tells compiler “Assume there are no loop-carried dependencies”

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder

- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

The rise of SIMD

- SIMD is good for applying identical computations across many data elements
  - E.g., `for (i = 0; i < 100; i++) { A[i] = B[i] + C[i]; }`
  - Data-level parallelism
- SIMD is energy efficient
  - Less control logic per functional unit
  - Less instruction fetch and decode energy
- SIMD computations tend to be bandwidth-efficient and latency-tolerant
- Easy examples:
  - Dense linear algebra
  - Computer graphics
  - Machine learning
  - Digital signal processing

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - AVX-512 (2017)
    - Eight 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations

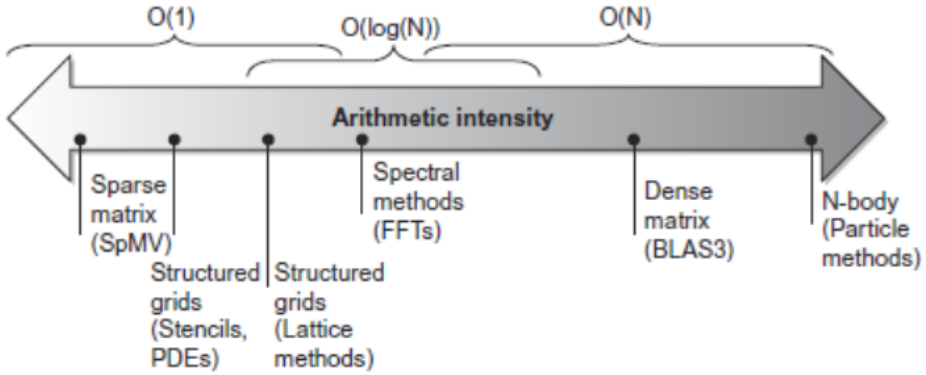
Example SIMD Code

- Example DXPY:

```
        fld             f0,a            # Load scalar a
        splat.4D        f0,f0           # Make 4 copies of a
        addix28,x5,#256 # Last address to load
Loop:   fld.4D          f1,0(x5)        # Load X[i] ... X[i+3]
        fmul.4D         f1,f1,f0        # a x X[i] ... a x X[i+3]
        fld.4D          f2,0(x6)        # Load Y[i] ... Y[i+3]
        fadd.4D         f2,f2,f1        # a x X[i]+Y[i]...
                                                # a x X[i+3]+Y[i+3]
        fsd.4D          f2,0(x6)        # Store Y[i]... Y[i+3]
        addix5,x5,#32   # Increment index to X
        addix6,x6,#32   # Increment index to Y
        bne             x28,x5,Loop     # Check if done
```

Roofline Performance Model

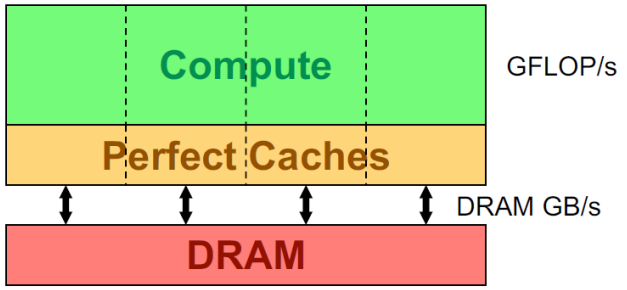
- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



Roofline (DRAM)

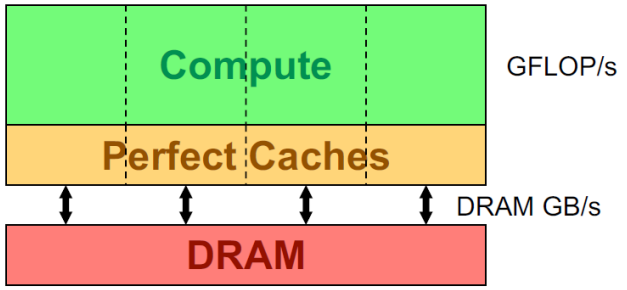
- Any given loop nest will perform:
  - Computation (e.g. FLOPs)
  - Communication (e.g. moving data to/from DRAM)
- With perfect overlap of communication and computation...
  - Run time is determined by whichever is greater

$$\text{Time} = \max \begin{cases} \#\text{FLOPs} / \text{Peak GFLOP/s} \\ \#\text{Bytes} / \text{Peak GB/s} \end{cases}$$



Roofline (DRAM)

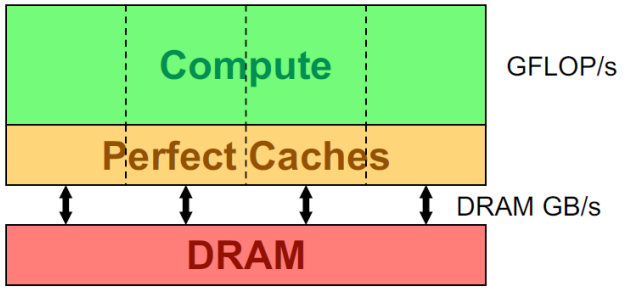
- Any given loop nest will perform:
  - Computation (e.g. FLOPs)
  - Communication (e.g. moving data to/from DRAM)
- With perfect overlap of communication and computation...
  - Run time is determined by whichever is greater



$$\frac{\text{Time}}{\#\text{FLOPs}} = \max \begin{cases} 1 \,/\, \text{Peak GFLOP/s} \\ \#\text{Bytes} \,/\, \#\text{FLOPs} \,/\, \text{Peak GB/s} \end{cases}$$

Roofline (DRAM)

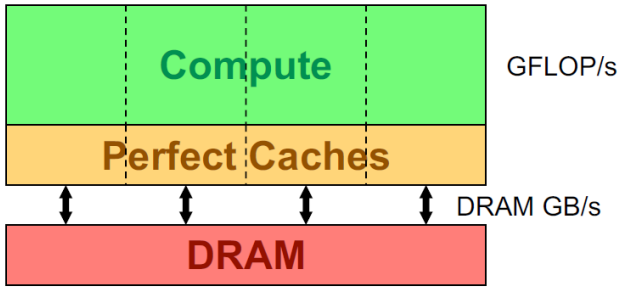
- Any given loop nest will perform:
  - Computation (e.g. FLOPs)
  - Communication (e.g. moving data to/from DRAM)
- With perfect overlap of communication and computation...
  - Run time is determined by whichever is greater



$$\frac{\#\text{FLOPs}}{\text{Time}} = \min \begin{cases} \text{Peak GFLOP/s} \\ (\#\text{FLOPs} / \#\text{Bytes}) * \text{Peak GB/s} \end{cases}$$

Roofline (DRAM)

- Any given loop nest will perform:
  - Computation (e.g. FLOPs)
  - Communication (e.g. moving data to/from DRAM)
- With perfect overlap of communication and computation...
  - Run time is determined by whichever is greater



$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases}$$

AI (Arithmetic Intensity) = FLOPs / Bytes (as presented to DRAM)

Aritmetic Intensity

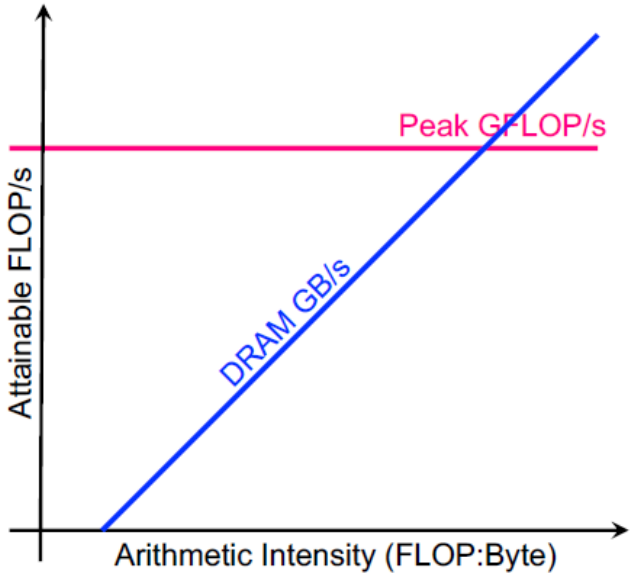
- Measure of data locality (data reuse)
- Ratio of **Total Flops** performed to **Total Bytes** moved
- For the DRAM Roofline…
  - Total Bytes to/from DRAM
  - Includes all cache and prefetcher effects
  - Can be very different from total loads/stores (bytes requested)
  - Equal to ratio of sustained GFLOP/s to sustained GB/s (time cancels)

Roofline Model

$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases}$$

AI (Arithmetic Intensity) = FLOPs / Bytes (moved to/from DRAM)

- Plot Roofline bound using Arithmetic Intensity as the x-axis
- **Log-log scale** makes it easy to doodle, extrapolate performance along Moore's Law, etc…

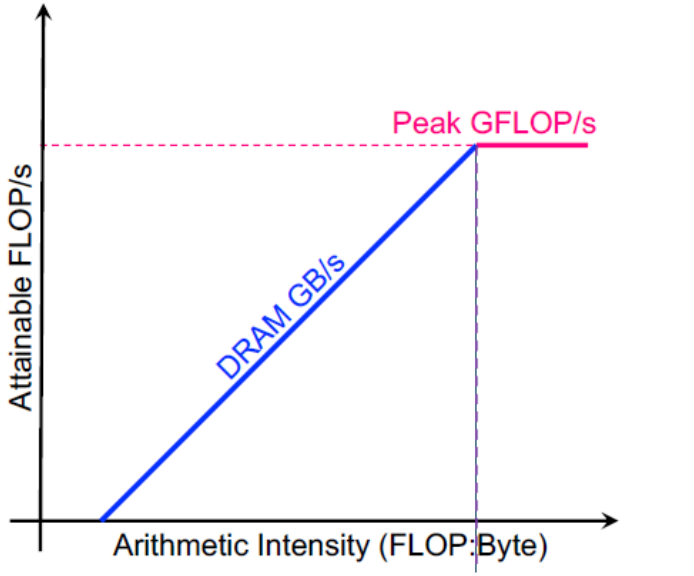


Roofline Model

$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases}$$

AI (Arithmetic Intensity) = FLOPs / Bytes (moved to/from DRAM)

- Plot Roofline bound using Arithmetic Intensity as the x-axis
- **Log-log scale** makes it easy to doodle, extrapolate performance along Moore's Law, etc…



Transition @ AI == Peak GFLOP/s / Peak GB/s == 'Machine Balance'

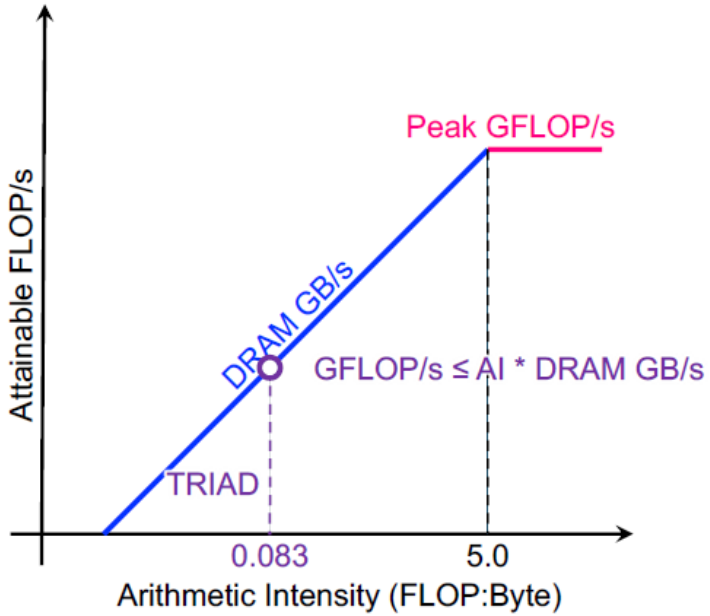
Roofline Examples #1

- Typical machine balance is 5-10 FLOPs per byte…
  - 40-80 FLOPs per double to exploit compute capability
  - Artifact of technology and money
  - **Unlikely to improve**

- Consider STREAM Triad…

```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

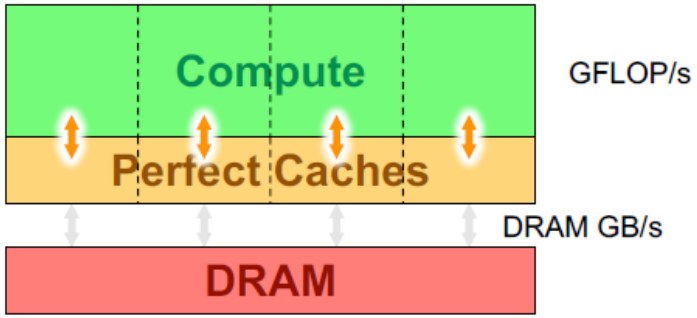
  - 2 FLOPs per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.083 FLOPs per byte == Memory bound**



Roofline Examples #2

- Conversely, 7-point constant coefficient stencil…
  - 7 FLOPs
  - 8 memory references (7 reads, 1 store) per point
  - **AI = 7 / (8*8) = 0.11 FLOPs per byte**

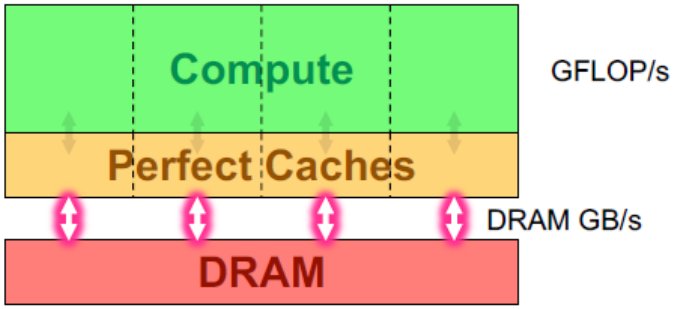
    (measured at the L1)




```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
 new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                + old[k  ][j  ][i-1]
                + old[k  ][j  ][i+1]
                + old[k  ][j-1][i  ]
                + old[k  ][j+1][i  ]
                + old[k-1][j  ][i  ]
                + old[k+1][j  ][i  ]
}}}
```

Roofline Examples #2

- Conversely, 7-point constant coefficient stencil…
  - 7 FLOPs
  - 8 memory references (7 reads, 1 store) per point
  - **Ideally, cache will filter all but 1 read and 1 write per point**

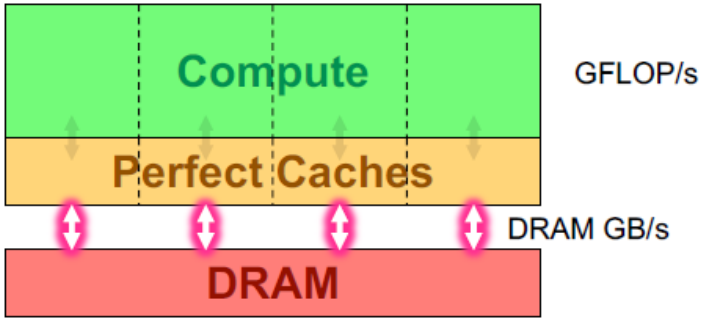



```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
 new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                + old[k  ][j  ][i-1]
                + old[k  ][j  ][i+1]
                + old[k  ][j-1][i  ]
                + old[k  ][j+1][i  ]
                + old[k-1][j  ][i  ]
                + old[k+1][j  ][i  ]
}}}
```

Roofline Examples #2

- Conversely, 7-point constant coefficient stencil…
  - 7 FLOPs
  - 8 memory references (7 reads, 1 store) per point
  - Ideally, cache will filter all but 1 read and 1 write per point
  - ➢ **7 / (8+8) = 0.44 FLOPs per byte (DRAM)**

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                 + old[k  ][j  ][i-1]
                 + old[k  ][j  ][i+1]
                 + old[k  ][j-1][i  ]
                 + old[k  ][j+1][i  ]
                 + old[k-1][j  ][i  ]
                 + old[k+1][j  ][i  ];
}}}
```

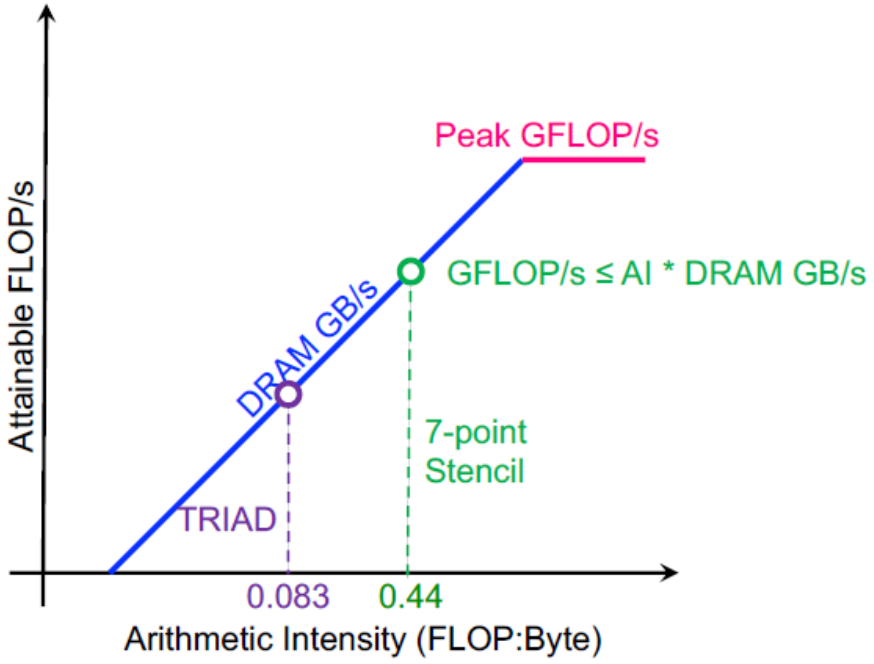


Roofline Examples #2

- Conversely, 7-point constant coefficient stencil…
  - 7 FLOPs
  - 8 memory references (7 reads, 1 store) per point
  - Ideally, cache will filter all but 1 read and 1 write per point
  - ➢ **7 / (8+8) = 0.44 FLOPs per byte (DRAM)**

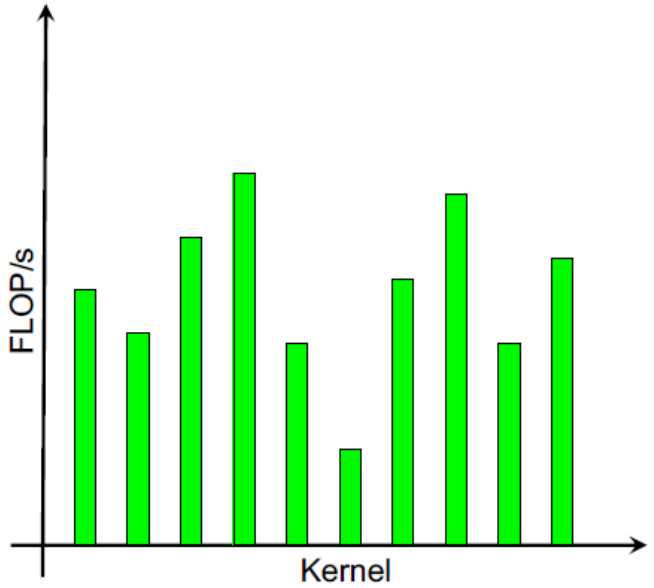
    == memory bound, but 5x the FLOP rate as TRIAD

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                     + old[k  ][j  ][i-1]
                     + old[k  ][j  ][i+1]
                     + old[k  ][j-1][i  ]
                     + old[k  ][j+1][i  ]
                     + old[k-1][j  ][i  ]
                     + old[k+1][j  ][i  ];
}}}
```



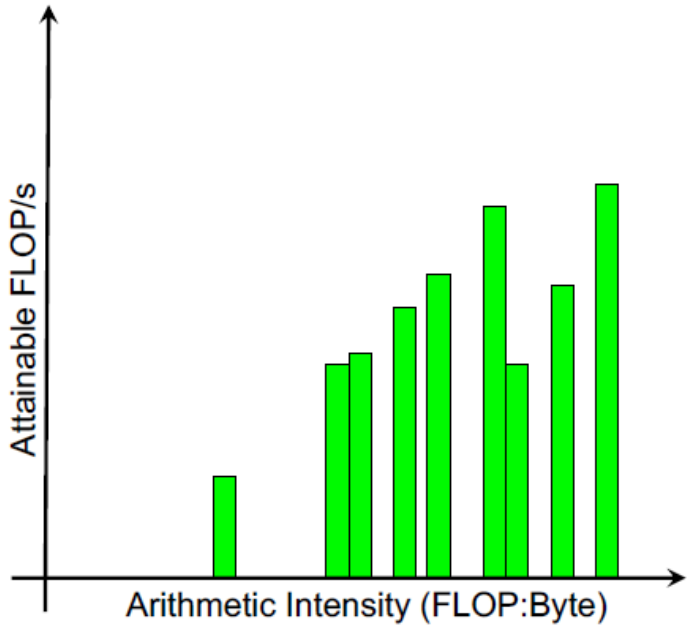
What is “Good” Performance?

- Think back to our mix of loop nests (benchmarks)…



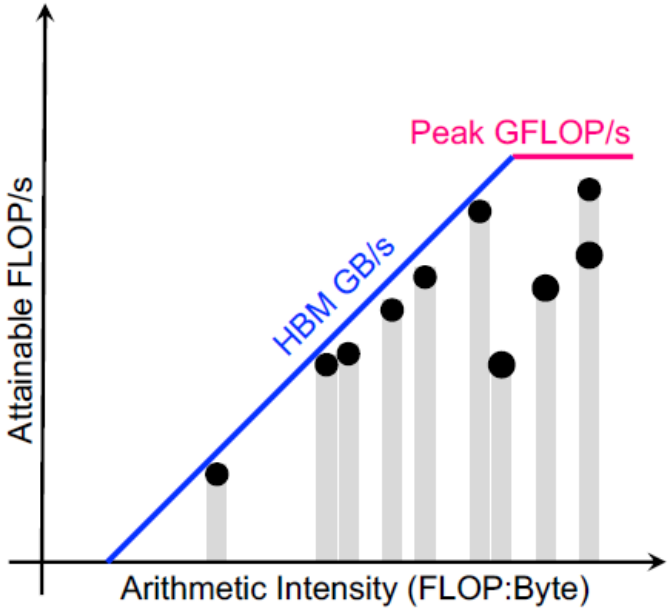
What is “Good” Performance?

- Think back to our mix of loop nests (benchmarks)
- We can sort kernels by their arithmetic intensity…



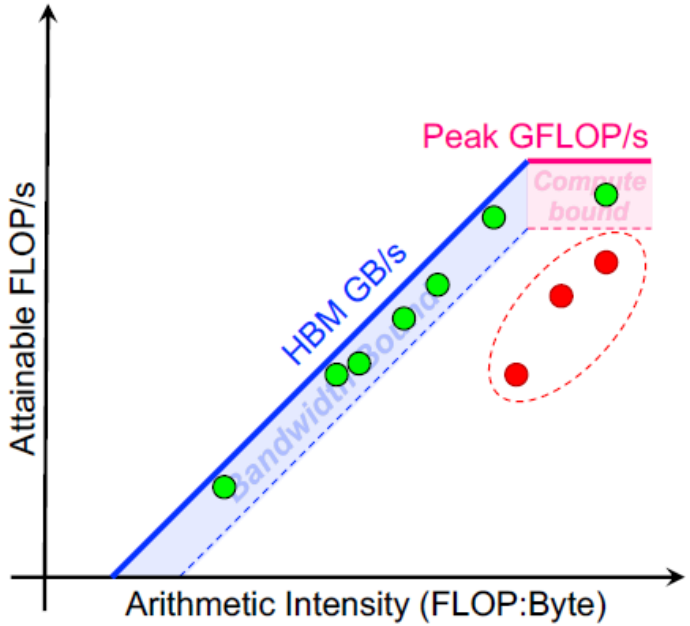
What is “Good” Performance?

- Think back to our mix of loop nests (benchmarks)
- We can sort kernels by their arithmetic intensity…
- … and compare performance relative to machine capabilities



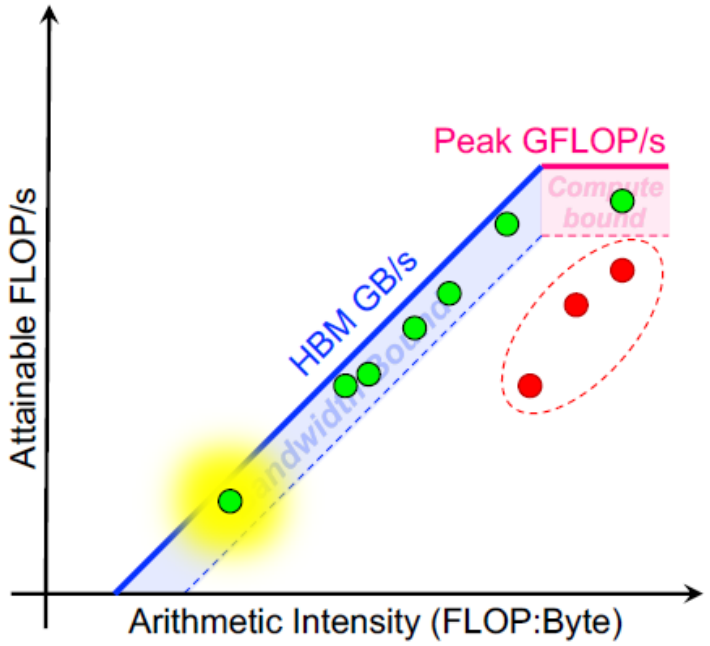
What is “Good” Performance?

- Kernels near the roofline are making **good use** of computational resources



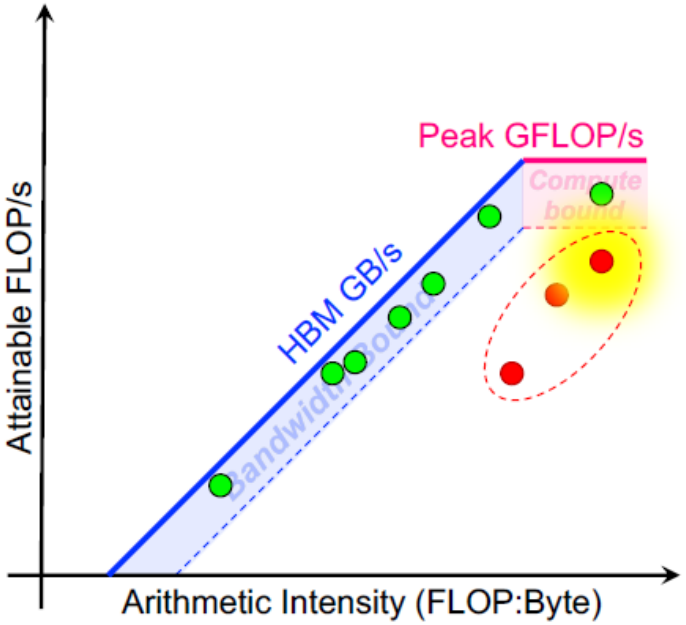
What is “Good” Performance?

- Kernels near the roofline are making **good use** of computational resources
  - kernels can have **low performance** (GFLOP/s), but make **good use** (%STREAM) of a machine



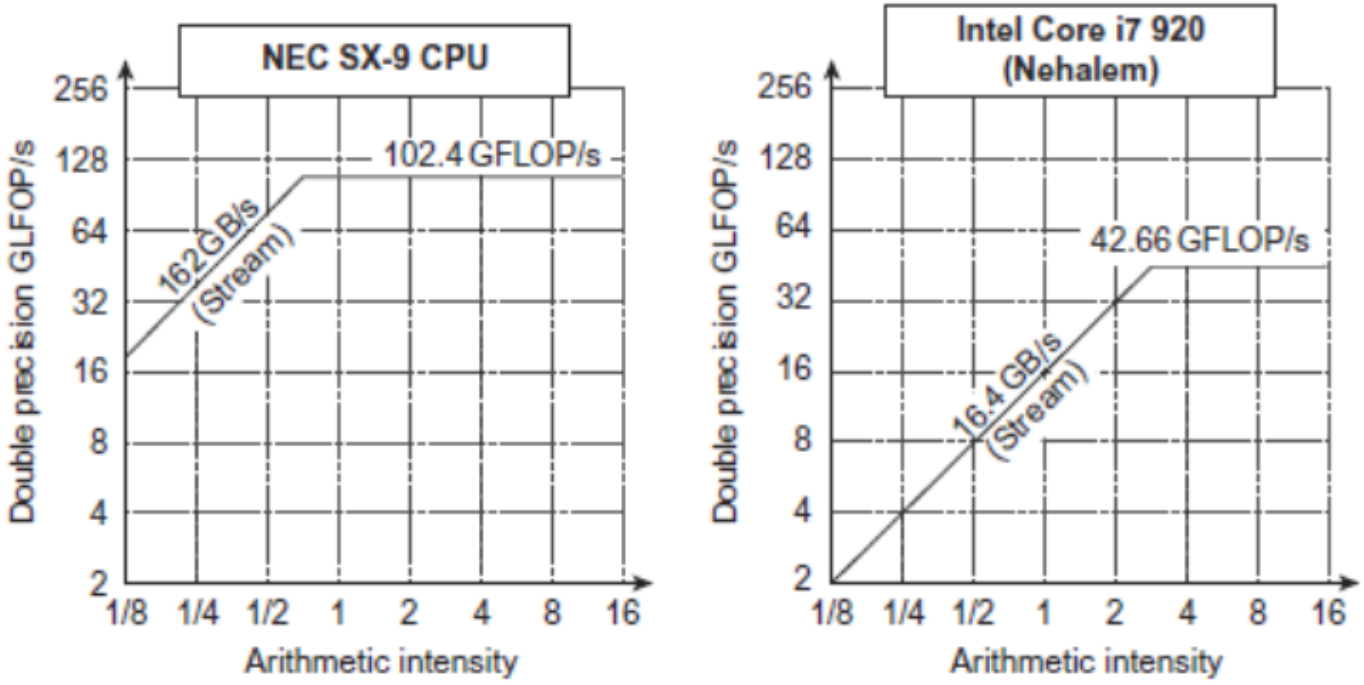
What is “Good” Performance?

- Kernels near the roofline are making **good use** of computational resources
  - kernels can have **low performance** (GFLOP/s), but make **good use** (%STREAM) of a machine
  - kernels can have **high performance** (GFLOP/s), but still make **poor use** of a machine (%peak)



Examples

- Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)



Intel core specs