

parallelizzare istruzioni che fanno lo stesso
caso su dati diversi
↑ una singola istruzione fa effetto su + dati

SIMD and Vector Architecture

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2022/2023

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

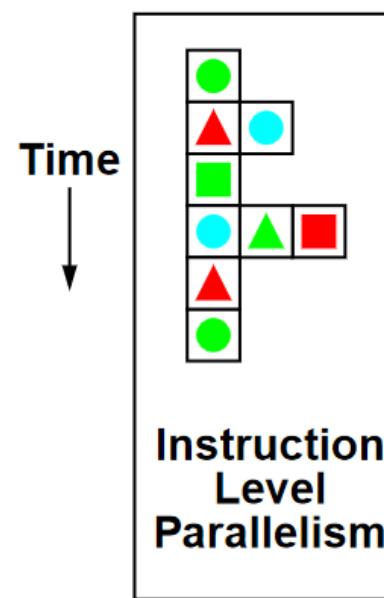
Parallelism

ILP: Parallelismo di istruzioni; eseguiamo più istruzioni in parallelo nel tempo.

TLP: flussi di istruzioni che sono diversi però su unità hw che possono eseguire concorrentemente.

Vector o SIMD architecture: eseguiamo contemporaneamente la stessa istruzione su multipli dati. È la forma di parallelismo meno flessibile,

- 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)



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

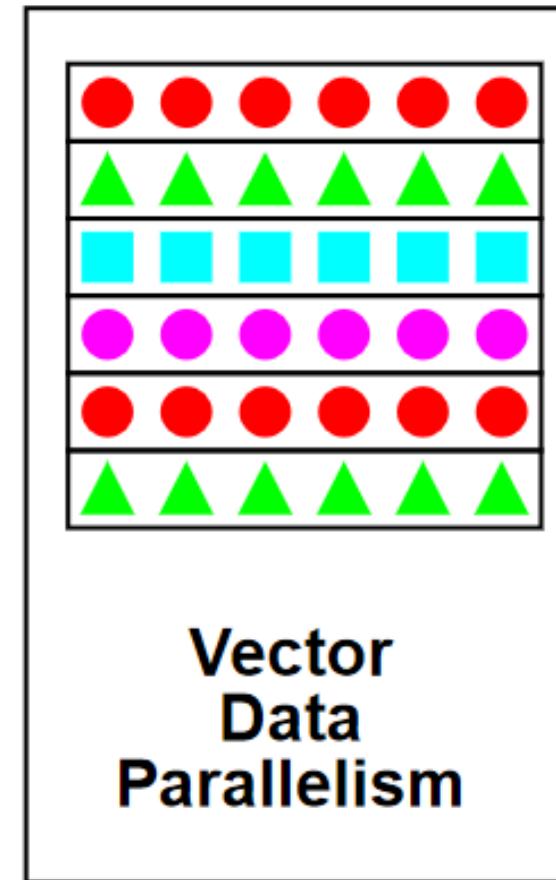
In questo caso a b e c sono vettori (array) e eseguiamo la stessa istruzione sui diversi elementi che li compongono. Non ce dipendenza di dato tra elementi. Posso tradurre il codice in :

$A[0] = B[0] + C[0]$
 $A[1] = B[1] + C[1]$
 $A[n-1] = \dots$

Si può srotolarlo il ciclo in più istruzioni tutte uguali. In questo caso tommaso lo potrebbe anticipare l'esecuzione di un istruzione per tenere sempre occupata la pipe.

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...



→ very long instruction word

VLIW: vector HW without vector ISA

Il compilatore deve dividere le istruzioni e mapparle sulle unità funzionali.

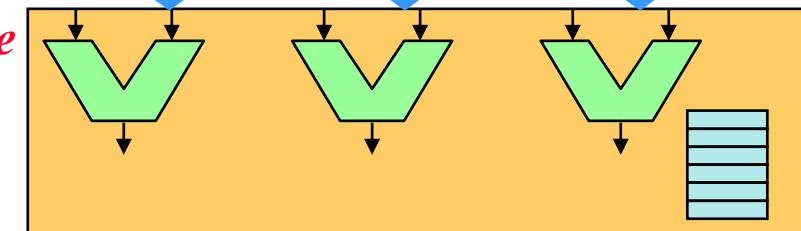
VLIW: Multiple independent operations packed together by the compiler

| | |
|-------|----|
| instr | op |

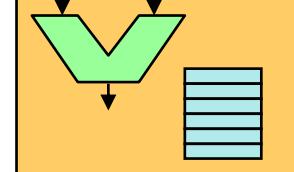
Compiler

| | | | |
|-------|------------|------------|------------|
| instr | op | op | op |
| instr | <i>nop</i> | op | op |
| instr | op | op | <i>nop</i> |
| instr | op | <i>nop</i> | op |
| instr | op | op | op |

execute
1 instr/cycle
3 ops/cycle



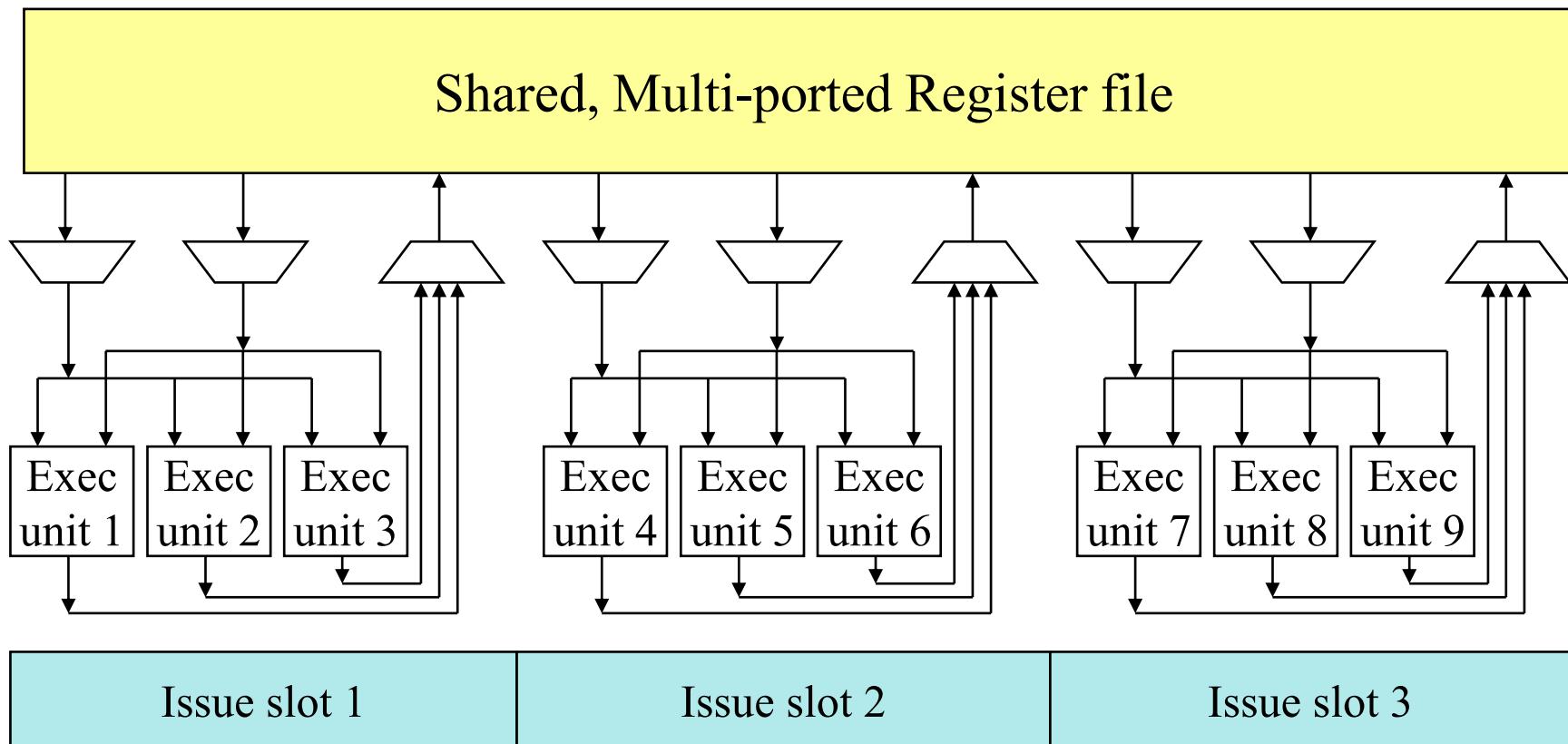
execute
1 instr/cycle



RISC CPU

3-issue VLIW

VLIW architecture: central Register File



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

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.

Rispetto all'architettura precedente in cui ciascuna replica dell'unità funzionale poteva eseguire istruzioni diverse, quindi la problematica era decidere quale istruzione va eseguita in un determinato ciclo su una certa unità funzionale. Adesso si ha un'architettura che ha sempre repliche di unità funzionali ma eseguono la stessa istruzione sulle varie unità. Quindi il mapping delle istruzioni sulle invita è più facile perché è la stessa istruzione che esegue. Non c'è più il problema di risolvere problemi legati a dipendenze.

Inoltre le architetture che supportano parallelismo a livello di dati, per poter funzionare i dati non devono avere dipendenze tra loro.

Le singole istruzioni sono applicate su multipli dati e ne, momento in cui si ha la stessa istruzione che opera su più dati si può decidere che questa stessa operazione venga gestita in modo temporale, quindi i multipli dati vengono e dati in modo seriale alla unità funzionale, quindi si condivide la stessa unità funzionale per più dati. Oppure si lavora nello spazio, cioè si hanno più unità funzionali a cui diamo nello stesso tempo la stessa istruzione.

Esistono due macro famiglie di architetture che consentono di eseguire la stessa istruzione su più dati, ovvero gli array processor (simd) e i vector processor che sono acceleratori in cui le operazioni eseguono su multipli dati nel tempo condividendo lo stesso hw. Mentre le architetture di tipo array operano su multipli dati nello stesso tempo ma su multipli unità funzionali.

Array vs. Vector Processors



Instruction Stream

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

ARRAY PROCESSOR

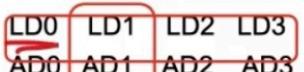


Caricare 4 elementi di A salvare i risultati
fare la somma dell'array con gli elementi del registro 1
farne la moltiplicazione
Salvare i valori nell'array
risultato

ARRAY PROCESSOR



Same op @ same time



Different ops @ same space

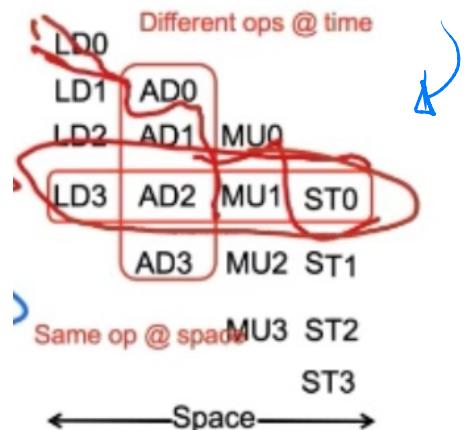
abbiamo più elementi architettonici
che possono svolgere la stessa
funzione, insieme concorrentemente
Sei le load (x es.) su 4 elementi
dell'array e poi add e poi ...

Nei processori vettoriali non si ha una replica nello spazio, ma si vanno a aprire le unità funzionali, ovvero c'è ne una che c'è gestisce le add, una che gestisce le load...

Sappiamo che si fanno operazioni in sequenza su dati che sono indipendenti tra loro. Quindi si sfrutta il pipelining. In entrambi i casi ci vogliono 4 cicli. Il lato positivo della struttura a pipeline è che non ci sono stalli perché i dati sono indipendenti.

Che seno ha avere un architettura che fa parallelismo a livello di dati ma esegue un dato alla volta su una singola unità funzionale? Il vantaggio sta nel sapere che si fa la stessa istruzione su multipli dati e questo ci consente di dire che non ci sono dipendenze tra i dati, per cui posso avere architetture più ottimizzate che usano un pipelining più massivo.

VECTOR PROCESSOR



Vector Processors (I)

La differenza sostanziale e che nelle architetture vettoriali l'operazione agisce su vettori e quindi si ha garanzia che con una singola istruzione si rappresenta l'idea di svolgere più operazioni nel tempo, dell' stesso tipo a partire da una certa locazione in memoria (quindi su vettori). Questo da garanzia che non ci avranno dipendenze di dato.

- 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;
```

Caso gpu: ha tanti core che fanno la stessa istruzione su multipli dati.

- 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

register file che invece di contenere variabili scalari contiene dei vettori

Inoltre c'è la necessità di lavorare su vettori che hanno dimensioni diverse, non necessariamente $n=32$, $n=64$ può essere che c'è qualcuno con $n=35$, $n=64$

c'è bisogno di un registro di config. che determina da quanti elementi è composto il vettore nel momento in cui viene eseguita un'istruzione vettoriale.

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

↗ stessa operazione nel tempo su elementi diversi dello stesso vettore.

stride - causante di sapere in anticipo quali sono gli indirizzi de memoria a cui fare load/store dei dati.

↓
facile fare prefetching

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

es. se si lavora con liste e' difficile usare le architetture vettoriali
e in generale x tutti gli algoritmi
con parallelismo irregolare

Vector Processor Disadvantages

- Works (only) if parallelism is regular (data parallelism)
 - ++ Vector operations
 - Very inefficient if parallelism is irregular
 - How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Fisher, “Very Long Instruction Word architectures and the ELI-512,” ISCA 1983.

History of Vector Machines

- Vector machines were one of the earlier ways to achieve high performance computing
 - In the eighties and nineties, **supercomputer = vector machine**
 - Dropped out of favor
 - Too difficult to program
 - Silicon is inexpensive
- Vector machines made a come back with the end of Moore's Law
 - As of today, the fastest supercomputer in the world is a vector machine!

TOP500 LIST - NOVEMBER 2020

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

R_{peak} values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

← 1-100 101-200 201-300 301-400 401-500 →

| Rank | System | Cores | R _{max} (TFlop/s) | R _{peak} (TFlop/s) | Power (kW) |
|------|--|-----------|-------------------------------|--------------------------------|---------------|
| 1 | Supercomputer Fugaku - Supercomputer Fugaku, A64FX 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan | 7,630,848 | 442,010.0 | 537,212.0 | 29,899 |

- Although 6 of the 10 fastest are GPU-based systems...

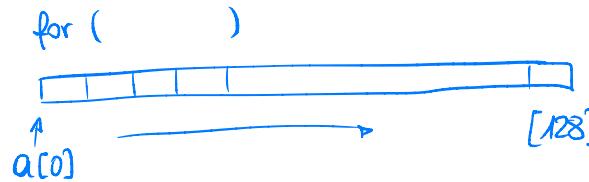
Vector Supercomputers

- Epitomized by Cray-1, 1976:
- Scalar Unit + Vector Extensions
- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

Vector Architectures

- Basic idea:
 - Read sets of data elements into “vector registers”
 - “vector registers” \approx 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

L'idea alla base della sua vettoriale è leggere gruppi di dati del vettore e salvarli all'interno di un vector register file.



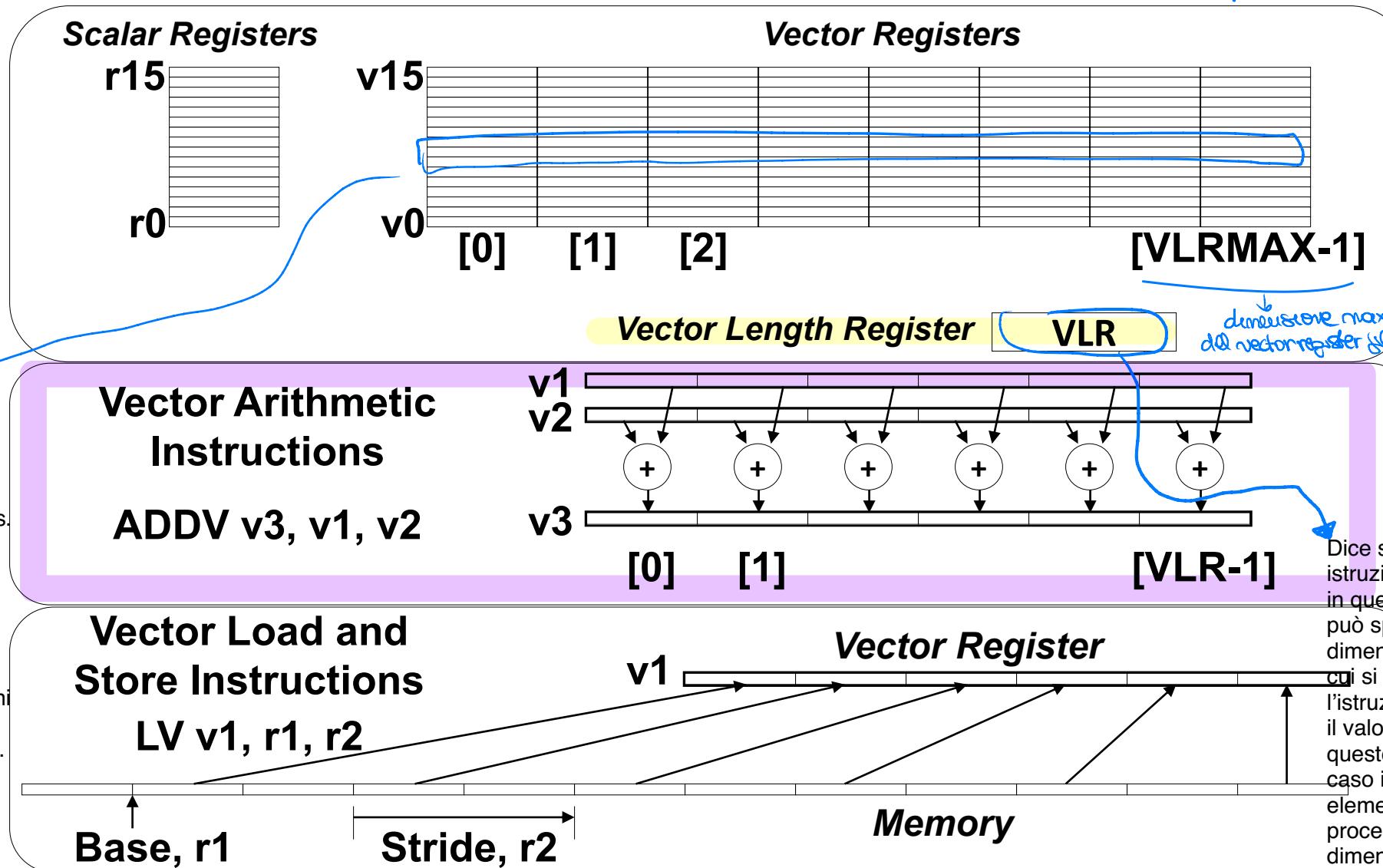
a. formato da 128 elementi
con l'isa vettoriale si prelevano
set di dati che si copiano con l'istruzione vector
load in un vector register file e poi si opera sulle
porzioni del vettore nel vector register file. Successiva-
mente si esegue un'altra istruzione vettoriale
sulla parte rimanente
dimensione del vettore che potrebbe
essere inferiore allo vector reg. file e potrebbe
quindi essere un avanzato da gestire con le
register che consente di dimensionare le dimensione
del vettore e in quel caso si può configurare il
vettore x avere una dimensione + conta del vec reg f.
Quindi si riesce con 2 iterazioni di un ciclo a processa-
re un intero vettore.
Dopo aver caricato dalla memoria i dati su un vector
register file, si esegue una singola istruzione su
molti per dati (elementi del vettore). Poi salvo i risultati in
memoria.

Vector Programming Model

l'architettura vettoriale
avrà sia un RF scalare
(composto da 32 registri
di tipo intero) sia un
vector RF,

$r_0 - r_{31}$
32 registri contenenti
una max dimensione di dati
es. RF da 1024 bit

Ci sarà un registro
di configurazione
che contiene l'
info di quanto è
il numero max di
elementi che possono
essere fisicamente
contenuti nel registro



Se ci asciuga di queste celle è l'equivalente di un registro scalare, quindi es. 32 bit, il registro $v0$ contiene $vlrmax$ elementi da 32 bit.

Il $vlrmax$ è un parametro architettura le, quindi si può comprare un processore con estensioni vettoriali del register file da 64 bit, o da 256 o 512. Se. Fugaku 64fx ha 512 bit , quindi nel vrf ogni registro è lungo 512 bit.

Dice su quanti elementi quel istruzione vettoriale opererà in quel momento. Quindi si può specializzare la dimensione degli elementi su cui si vuole parallelizzare l'istruzione andando a variare il valore che si scrive su questo registro (0-Rmax). Nel caso in cui la rimanenza Degli elementi del vettore da processare è più piccola della dimensione del registro.

Quando si fa una load vettoriale, l'indirizzo base è scalare, contenuto in un registro scalare, poi si ha un registro destinazione che è vettoriale e poi c'è un altro registro scalare che dice quanto deve essere incrementato l'indirizzo da un elemento all'elemento successivo all'interno di quel vettore.

Vlr restringe la porzione di register vector su cui opera l'istruzione, quindi può essere la lunghezza del vrf o una sua sotto sezione.

Stride: distanza in memoria lō

Vector Programming Model

Two main advantages with respect to the classical Scalar Processing:

1. Utilizes deeply pipelined ALUs to operate on multiple elements at the same time, without data hazards;
2. when it is possible, is able to exploit the data parallelism 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 ];
```

Configura in un reg. il n° di iteraz.
de svolgere, se ffd x ogni elemento

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
```

Vector Code:

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

Only 6 instr.

9 instr x 64 times

Nell'architettura vettoriale, si configura la lunghezza del vettore, che viene salvata in un registro scalare la lunghezza del vettore (64), con un istruzione dedicata che è set vector length si specifica il vlr, poi si ha l'istruzione vector load degli elementi di a , di b . L'istruzione vector add tra 1 e 2 e la vector store degli elementi del vettore 3 a partire dall'indirizzo x3. Con queste 6 istruzioni su esegue l'equivalente di 9x64 istruzioni dell'architettura scalare. Quindi è più compatto e risparmi il dover fare fetch di ogni elemento.

Nel caso del codice vettoriale che è a metà tra un acceleratore specializzato per fare un determinato conto è l'architettura con Neumann classica, l'istruzione di load viene pagata una volta per tutti gli elementi del vettore. Per N elementi fa fetch di una sola istruzione.

Sia il costo delle istruzioni per leggere i dati che gestire i loop sono ammortizzati perché ora un'istruzione opera su molti più dati.

Com'è fatta la unità di esecuzione delle architetture vettoriali?

È fortemente pipeline, cioè fatta in modo che a ogni ciclo possa prendere un nuovo elemento del vettore e farci un'operazione. Quindi si hanno unità funzionali fortemente pipelined e che a ogni ciclo possono prendere un'istruzione. Questo significa che non ci sono mai dipendenze di dato.

Se ad esempio nel mio acccleratore vettoriale non voglio una sola unità funzionale ma ne voglio n, posso aumentare il parallelismo di dati, in questo caso si mantiene la semantica dei vettori, quindi il fatto di avere un vrf profondo e gestire vettori di dimensioni diverse, ma adesso si hanno più unità funzionali, quindi si possono spalmare gli elementi del vettore sulle multiple unità funzionali. Praticamente associeremo una porzione del vrf a ciascuna unità funzionale.

Quindi si mantiene il parallelismo temporale ma si aggiunge anche un parallelismo spaziale nell'esecuzione. Quindi si mantiene la pipeline ma ne abbiamo n in parallelo. (Le unità funzionali delle istruzioni vettoriali sono diverse da quelle per le istruzioni scalari, cioè la pipeline è diversa. Se non vi sono istruzioni vettoriali ma solo scalari la pipeline delle vettoriali rimane vuota).

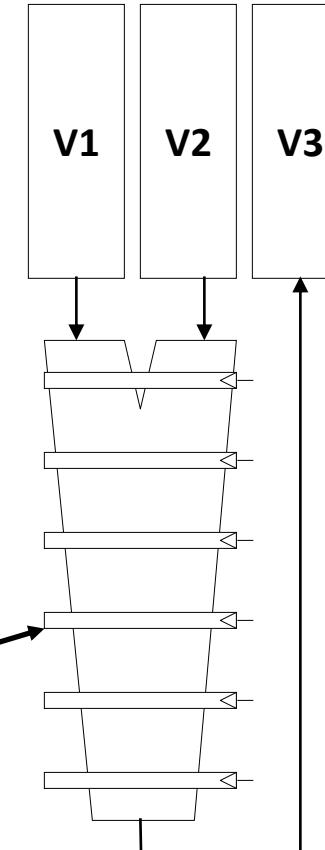
Vector Instruction Set Advantages

- 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

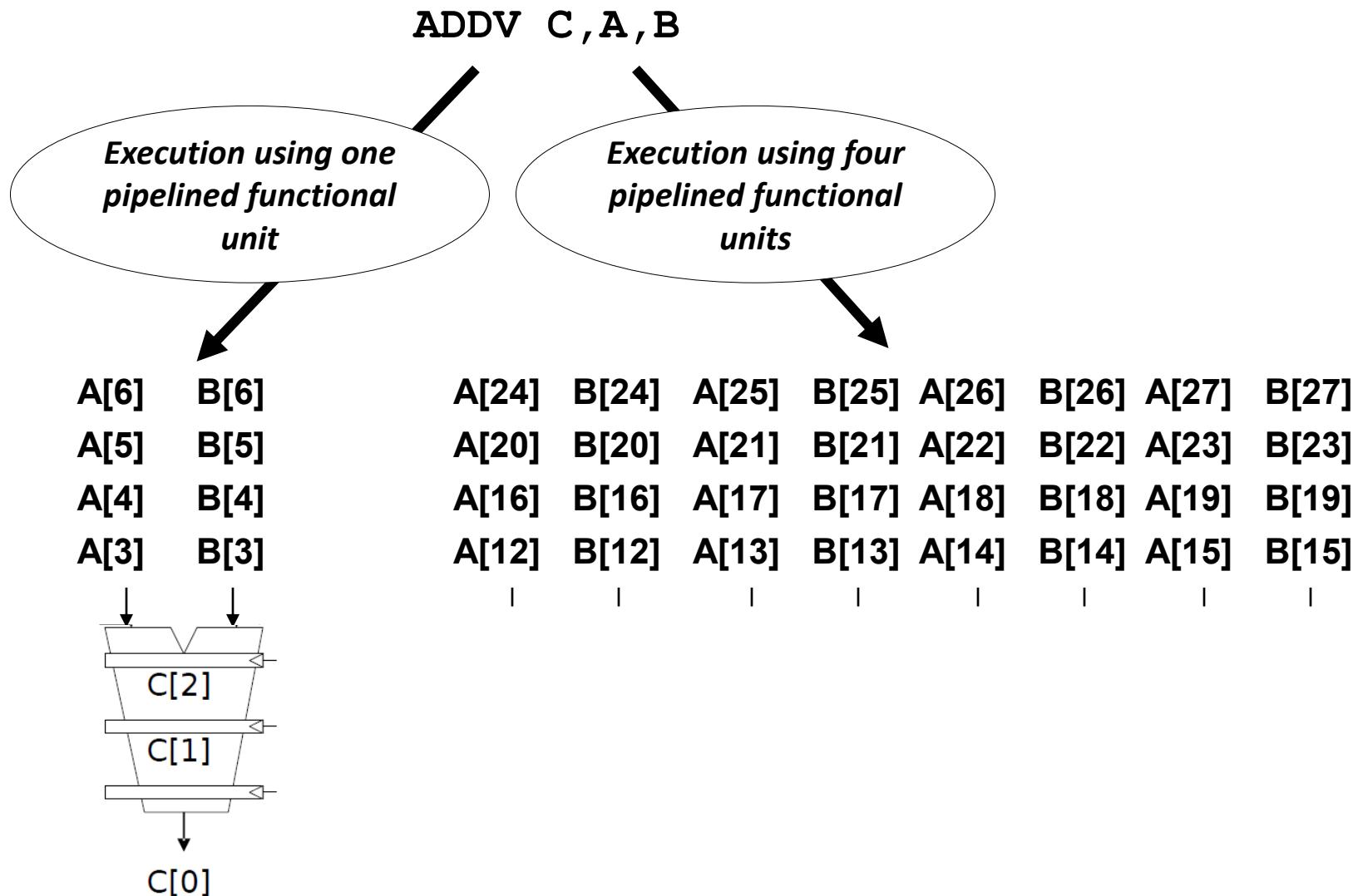
- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)

Six stage multiply pipeline



$$v_3 \leftarrow v_1 * v_2$$

Vector Instruction Execution



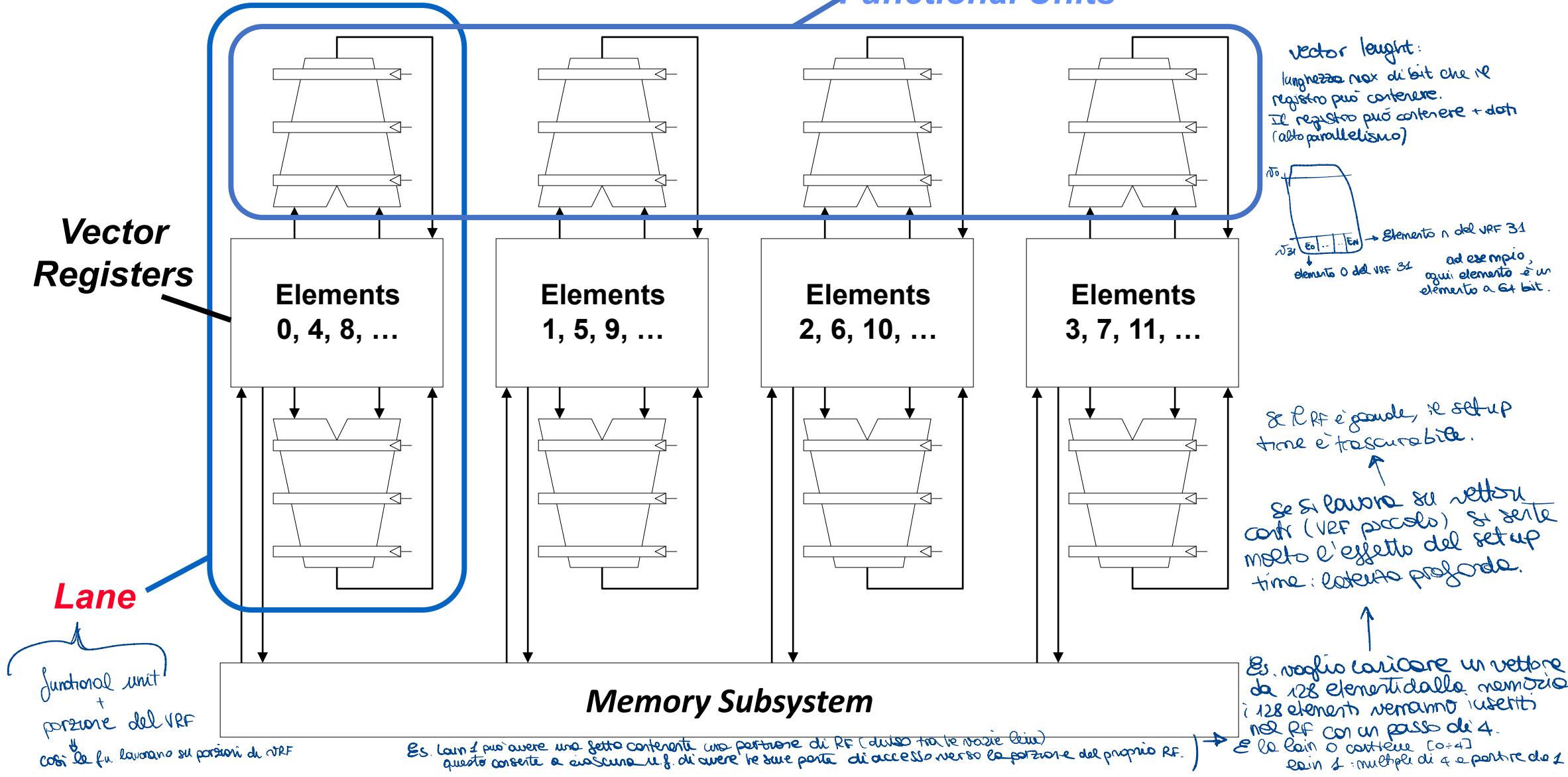
struttura hw che si aggiunge ai processori x sfruttare parallelismo a livello dati.

Vector Unit Structure

La parola lane indica il parallelismo, ad esempio in un processore vettoriale possono esserci 8 lane, ovvero si ha che in un singolo ciclo si possono fare 8 elementi del vettore. Più il parallelismo a livello di unità funzionali.

Il numero di functional unit è proporzionale al numero di lane che si hanno nell'acceleratore vettoriale.

Functional Units

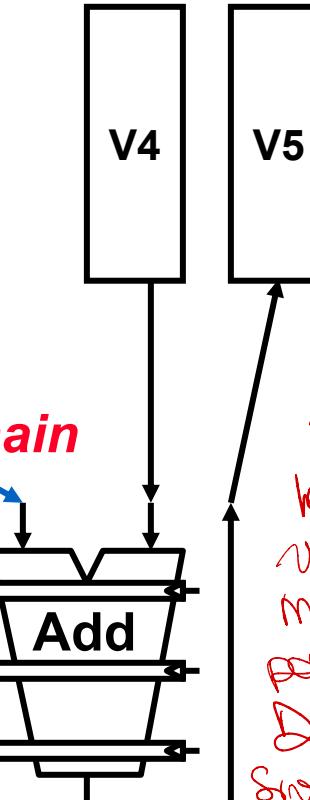
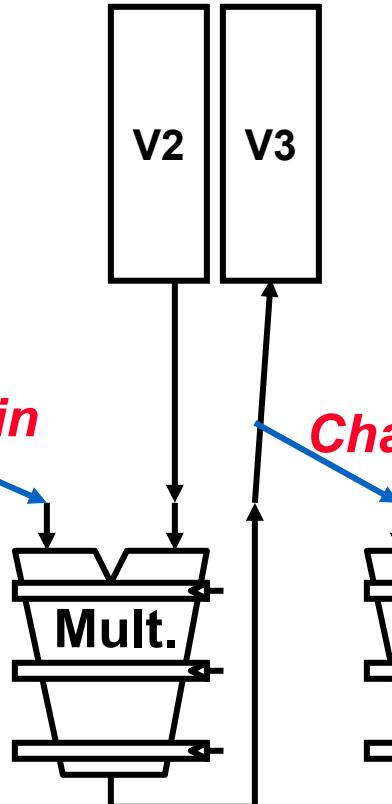
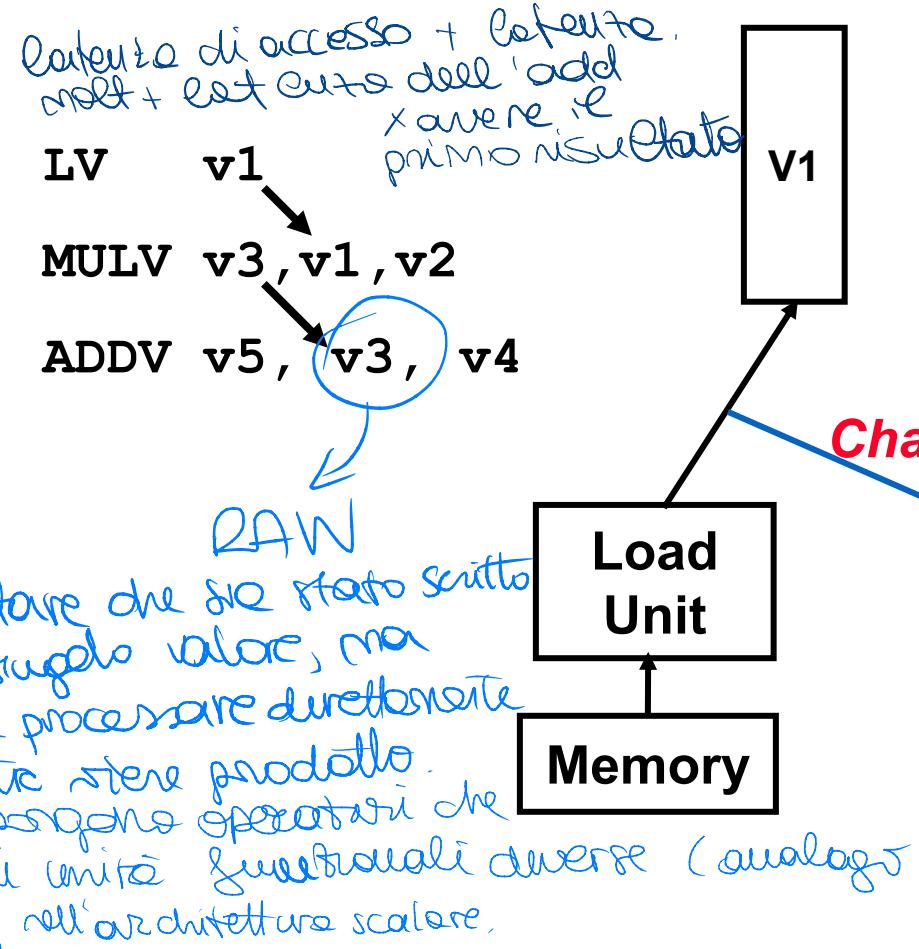


Altro concetto da introdurre è vector chaining: forwarding per le architetture scalari. Se sei ha una vector load nel registro V1 di v1 con stride 1 e poi una vector add in v3 di v1 è v2 (precedentemente caricato). Prima di iniziare a fare la somma devo una quella porzione di dato sia stata letta dalla memoria e salvata nel vector register file. Se il vrf è composto da 32 elementi, vuol dire che devo aspettare di leggere 32 variabili prima di iniziare a fare la somma. Però in realtà siccome l'acceleratore vettoriale legge in modo sequenziale, il dato 1 è pronto molto prima perché è già salvato nel vrf da prima di quando è terminato la lettura dell'ultimo elemento del vettore.

Vector Chaining

vld v1, x1, 1
vadd v3, v1, v2

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



La dipendenza di dato non vuol dire che per forza evo aspettare che l'operazione sia stata svolta su tutti gli elementi del vettore prima di iniziare quella dopo, posso invece a tutti gli effetti non appena un dato è pronto anziché salvarlo nel vrf posso convogliarlo come ingresso per L'Unità funzionale sommatore che a quel punto inizierà a processare direttamente il dato letto dalla memoria.

SENZA VECTOR C. PRIMA
BISOGNAREBBE FINIRE LE
LOAD FINO ALL'ULTIMO
ELEMENTO DEL VETTORE

vector chaining:

consente di
fare forwarding tra
la load/store unit
vettoriale e l'unità
moltiplicatore senza
passare dal RF.

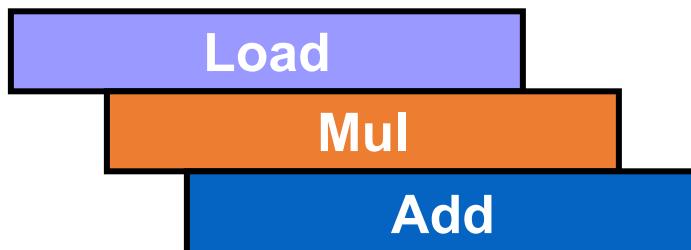
Dato che gli elementi succes-
sivi del vettore sono indipen-
di. Posso processare il 1° elemento
del vettore senza aspettare che il
dato letto dal vettore
sia stato letto. → Il dato
è trasferito come ingresso
nella moltiplicatrice.

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



Se si ha nella lain sia L'Unità che fa le load sia una che fa le mul ch è una che fa le somme, senza il chaining avrei dovuto aspettare che una singola istruzione processasse tutti gli elementi del vettore prima di passare alla successiva. Con il vector chaining posso sovrapporre queste fasi. Cioè non appena il primo dato dell'elemento zero è stato letto posso iniziare a processare l'elemento 0 dell'istruzione mul, mentre la load sta leggendo il andato dell'elemento 1. Sole a la mul produce il risultato del primo elemento posso iniziare a fare la add mentre la mul processa il dato due e la load il dato 3 ecc. quid i riesco a sovrapporre nello spazio le operazioni.

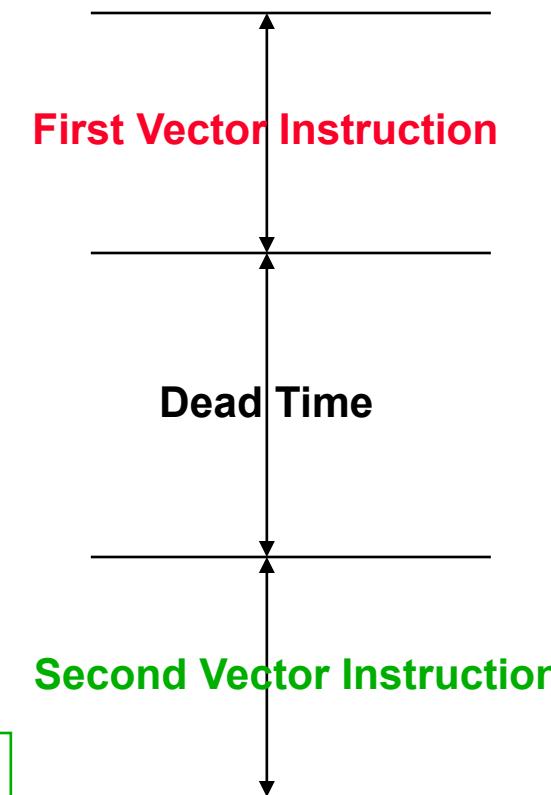
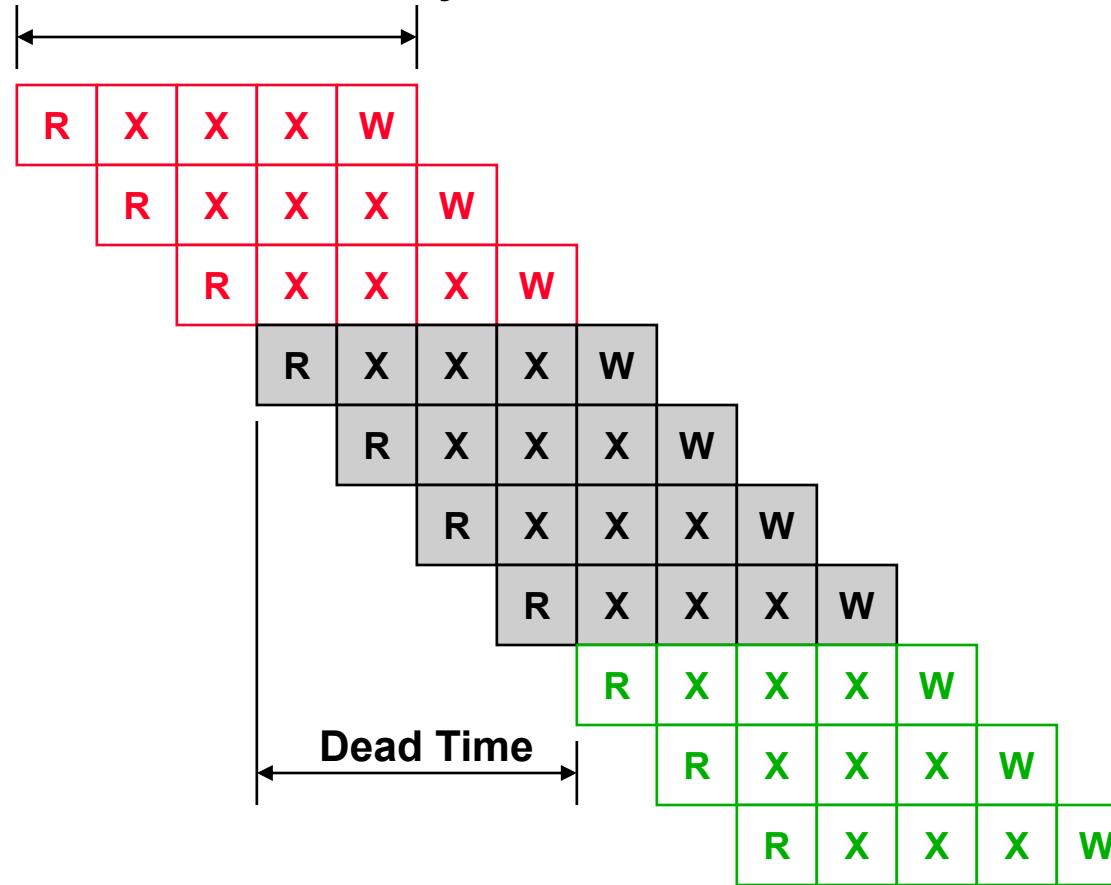
La logica di chaining si rompe quando ho un'unità strutturale, ovvero due istruzioni successive che insistono sulla stessa unità funzionale . Prima di iniziare la seconda load devo aver liberato l'unità dalla prima load. Pagherò il tempo di esecuzione di tutta L istruzione.

Vector Startup

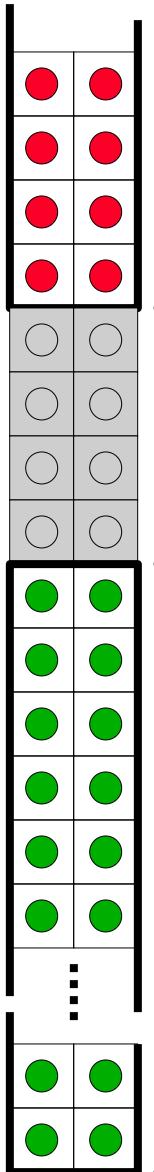
Start up time: Prima di veder l'efficacia della pipeline, l'elemento deve scorrere attraverso tutti gli stadi della pipeline, e quindi lo start up time è la latenza dell'unità funzionale. Sono i cicli che intercorrono da quando il primo elemento entra nella pipe e quando esce.

- 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)

Functional Unit Latency



Dead Time and Short Vectors



4 cycles dead time

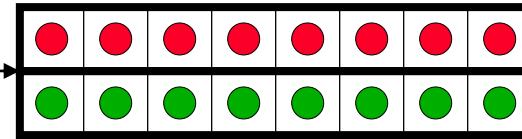
64 cycles active

Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94%
with 128 element vectors

No dead time →



T0, Eight lanes

No dead time

100% efficiency with 8 element
vectors

Tempo che serve per svuotare la pipe prima di eseguire una nuova istruzione sulla stessa unità funzionale, ad esempio se. Si fanno due vector load in serie , dovrò aspettare su aver terminato la load sul vettore 1 prima di iniziare quella sul vettore 2.

Le architetture simd sfruttano parallelismo micro architettura le a livello spaziale, mentre quelle vettoriali sfruttano un approccio di pipelining , cioè più unità funzionali che nel tempo processano elementi diversi del vettore.

Mentre nelle architetture simd si ha un'unità funzionale che ha un determinato parallelismo, che è capace di eseguire ad esempio in un SIMD a 256 bit 4 operazioni floating su delle double world, l'unità funzionale è capace di processare parallelamente 4 elementi dell'array.

Nel caso della logica vettoriale, ciascuna unità funzionale lavora in sequenza su un elemento dell'array.

Però se la micro architettura dell'architettura vettoriale ha più unità funzionali, queste possono lavorare in parallelo e si può avere che alcuni elementi si trovano nello stadio load e altri in add.

Oltre a questo le unità vettoriali mettono a disposizione repliche delle unità funzionali per lavorare in parallelo.

Lein: replica delle unità funzionali del sistema, e anche di porzioni del vrf. Oltre ad avere unità vettoriali e simd si portano dietro un rf dedicato per contenere dati ad alto parallelismo

Question

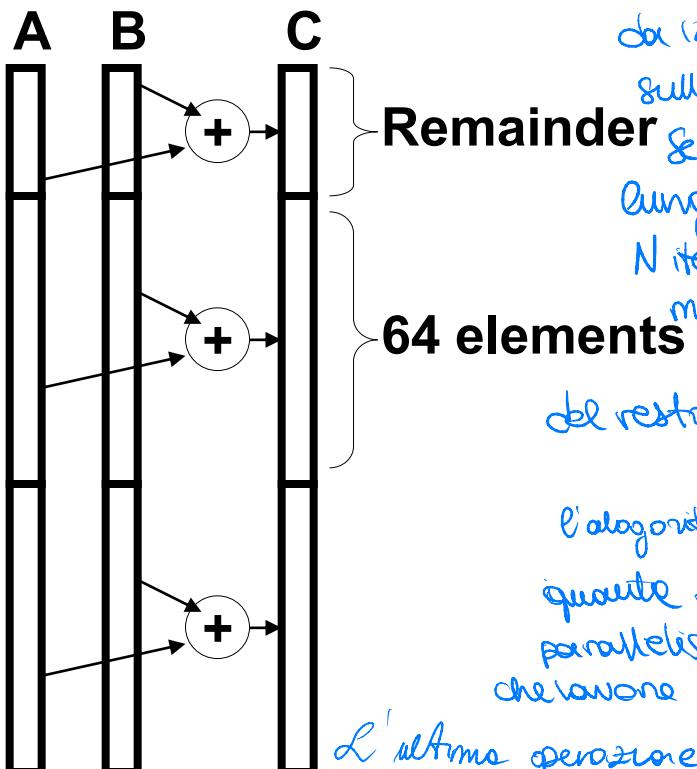
Se due programmi hanno array di lunghezza diversa,
il codice è scritto x

- 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**

Inoltre si paga una carica latente all'inizio, che peserà di più se la rimanenza degli elementi del vettore è inferiore alla dimensione totale

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



Esempio:

VRF: può contenere 128 elementi

Se si lavora su una struttura dati a 1024 bit, con 8 iterazioni da 128 bit si conclude il calcolo sull'array.

Se l'array ad esempio è lungo 1400 elementi, si faranno N iterazioni a dimensione max (128 bit) e poi un'ultima iterazione delle dimensioni del resto del vettore da processare.



l'algoritmo lavora x strisci del vettore
quante + possibili operazioni a max
parallelismo - e poi un'ultima iterazione
che lavora su una partizione + piccola.

L'ultima operazione potrebbe essere un po' inefficiente:
Si potrebbe avere un certo numero di loop, ma lo numero
potrebbe essere inferiore al n° di loop.

Automatic Code Vectorization

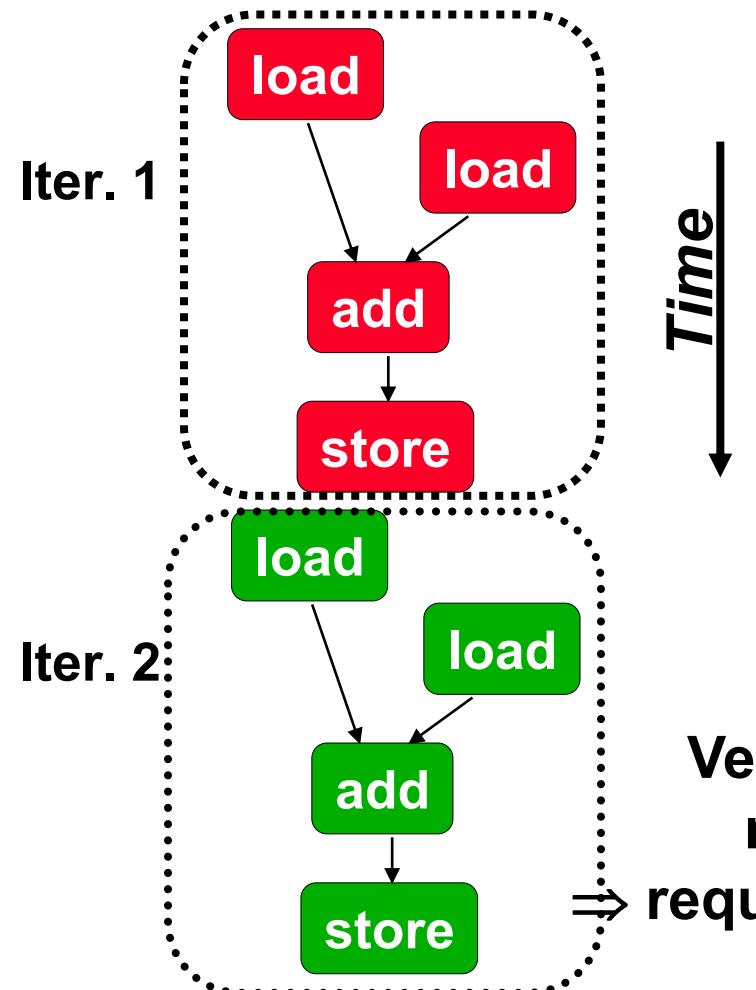
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Quando si vuole che il compilatore che vettorizzi il codice C, non è facile. Se si immagina un ciclo for senza dipendenze di dato, per noi è facile riconoscerlo, per il compilatore no.

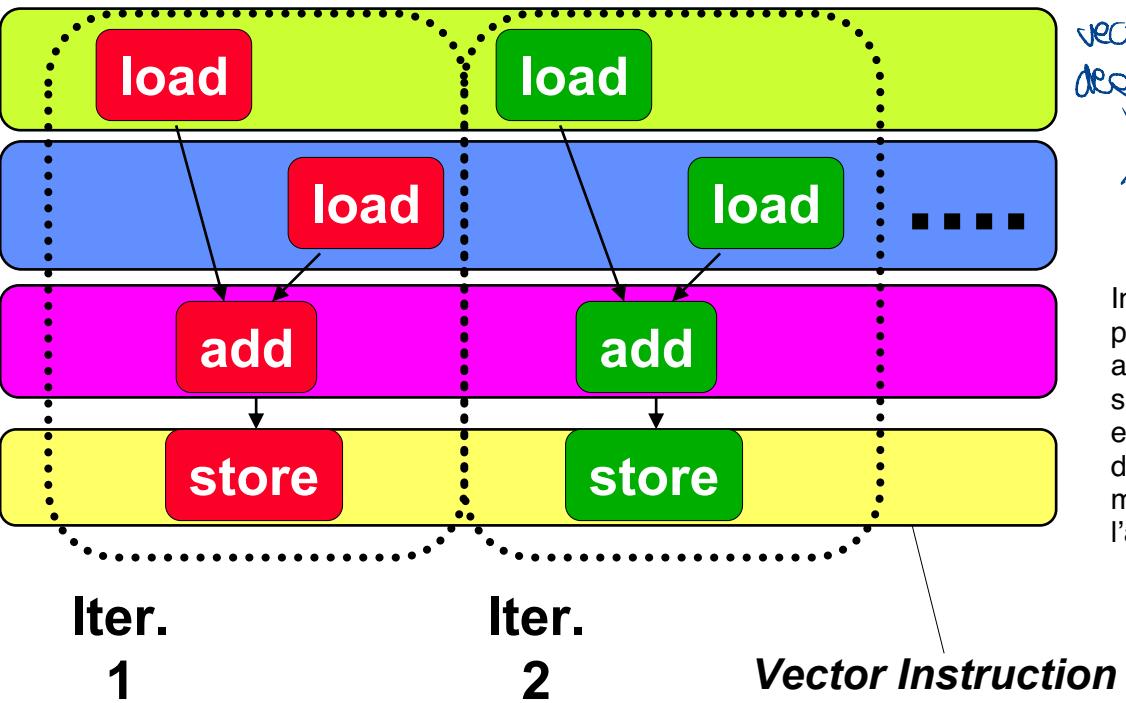
Nel caso di ciclo for in esempio si una load add store ripetuti N volte. L'algoritmo di tommasulo, quindi pipeline out of order sono in grado di fare riordinamenti dinamico delle istruzioni e sfruttare il parallelismo della pipe.

Nel nostro caso si vuole proprio trasformare in istruzioni vettoriali.

Scalar Sequential Code



Vectorized Code



In un qualche modo il programmatore riesce ad aggiungere del codice che specifica che il loop ad esempio e privi di dipendenze, in questo modo è più facilitata l'autovettorizzazione.

**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

distanza = 0
tra elementi del vettore

- Unit stride

Indirizzamento lineare, con stride unitario, cioè vengono caricate
delle memorie elementi contigui del vettore.

» Contiguous block of information in memory

» Fastest: always possible to optimize this

- Non-unit (constant) stride

Stride non unitario, esiste una distanza fisica tra il primo

» Harder to optimize memory system for all possible strides

» Prime number of data banks makes it easier to support different strides at full bandwidth

elementi sparsi

- Indexed (gather-scatter)

Indirizzamento tramite indice, a cui viene aggiunto un array di indici da sommare all'indice base, cui poi fare accesso in memoria con

» Vector equivalent of register indirect

» Good for sparse arrays of data

» Increases number of programs that vectorize

accesso sparso in memoria in cui usiamo a tal fine un vettore di indirizzi.

ESEMPIO

Vector Scatter/Gather

- Want to vectorize loops with indirect accesses:

- `for (i=0; i<N; i++)`

- `A[i] = B[i] + C[D[i]]`

// Somma tra il singolo elemento di B e l'elemento di C al quale accediamo tramite D, vettore di indice.

- Indexed load instruction (Gather)

- `LV vD, rD`

Load indices in D vector

- `LVI vC, rC, vD`

Load indirect from rC base de un registro base del vett.D.

- `LV vB, rB`

Load B vector

- `ADDV.D vA, vB, vC`

Do add

- `SV vA, rA`

Store result

Salvo il memoria questi elementi nel vettore C.

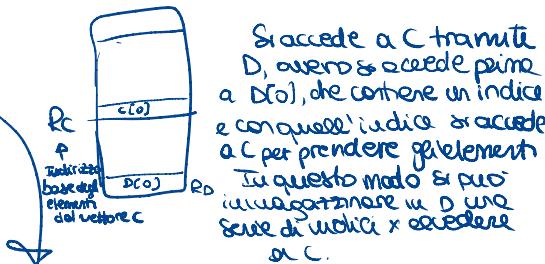
Nota: le liste hanno dipendenze fra gli elementi del vettore

No architetture vettoriali.

parte da un vettore
deuso, ordinato.
scrive in memoria
in modo non continuo

legge in modo sparso dalla
memoria e disporre i dati
in modo ordinato in un vettore

D: vettore di indici



Si accede a C tramite D, ovvero si accede prima a D[i0], che contiene un indice e con quell'indice si accede a C per prendere gli elementi. In questo modo si puo' memorizzare in D una serie di indici x accedere a C.

Successivamente viene fatta la lista degli elementi di B, poi viene fatta la vector ADD tra B e C e il risultato in A e poi si fa la vector store di A, a partire dall'indirizzo nel vettore base rA.

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];
```

Sì effettua l'istruzione solo se $A > 0 \Rightarrow$
c'è dipendenza. Elementi diversi del
vettore vedranno possibili diverse
di codice: $A = B$ se nessuno
degli $A > 0$

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

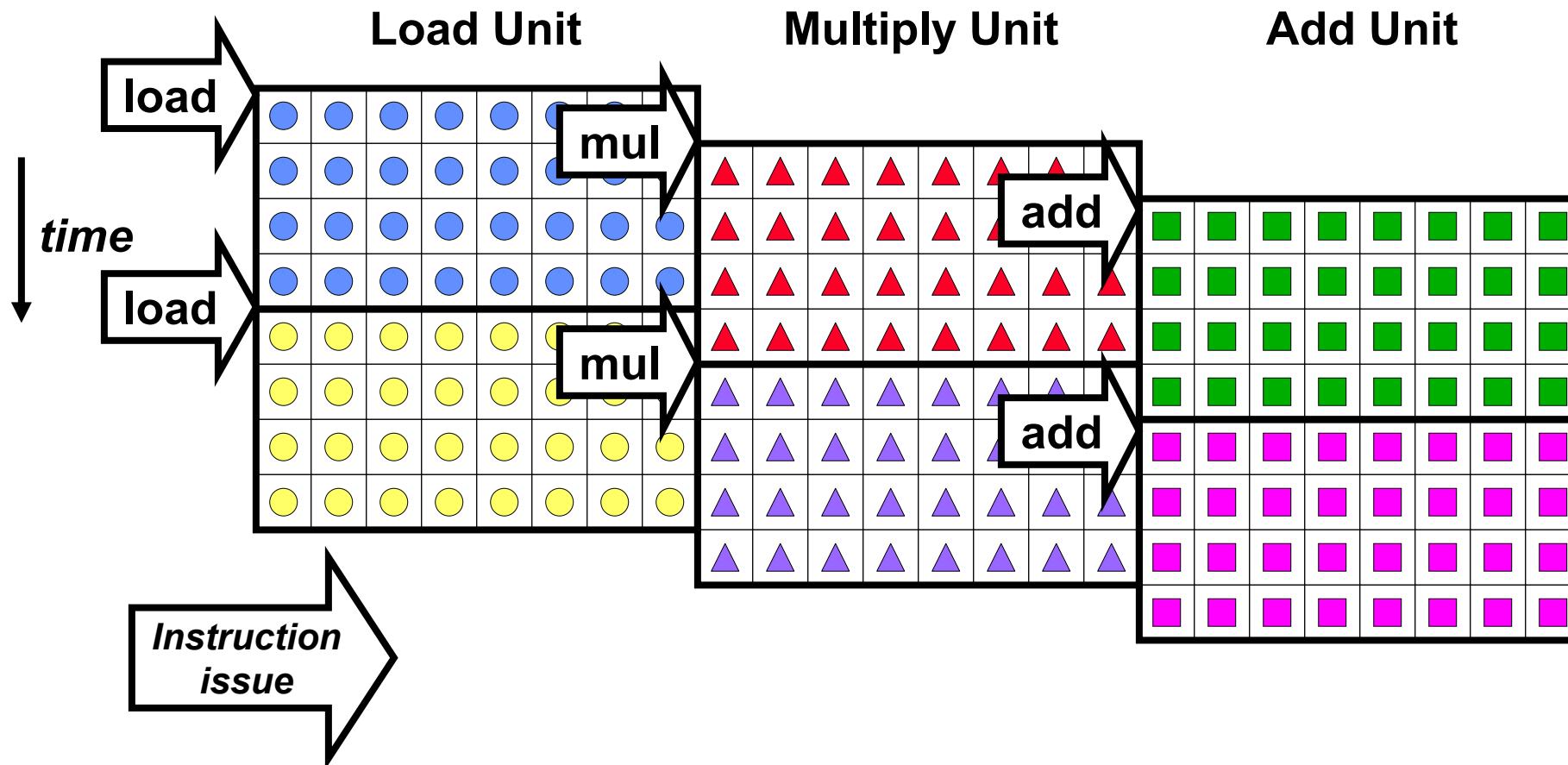
Code example:

```
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

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

Multimedia (SIMD) Extensions

- MMX follows a packed-SIMD execution model
 - *À la* array processing (yet much more limited)
 - Other examples include ARM NEON, Intel MMX/SSE/AVX, RISC-V P (DSP) Extension
- Multimedia extension instructions
 - Single instruction acts on multiple pieces of data at once
 - Common application: graphics
 - Perform short arithmetic operations (also called *packed arithmetic*)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

| | | | | |
|-------------|-------------|-------------|-------------|------|
| a_3 | a_2 | a_1 | a_0 | \$s0 |
| b_3 | b_2 | b_1 | b_0 | \$s1 |
| $a_3 + b_3$ | $a_2 + b_2$ | $a_1 + b_1$ | $a_0 + b_0$ | \$s2 |



Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements **simultaneously**

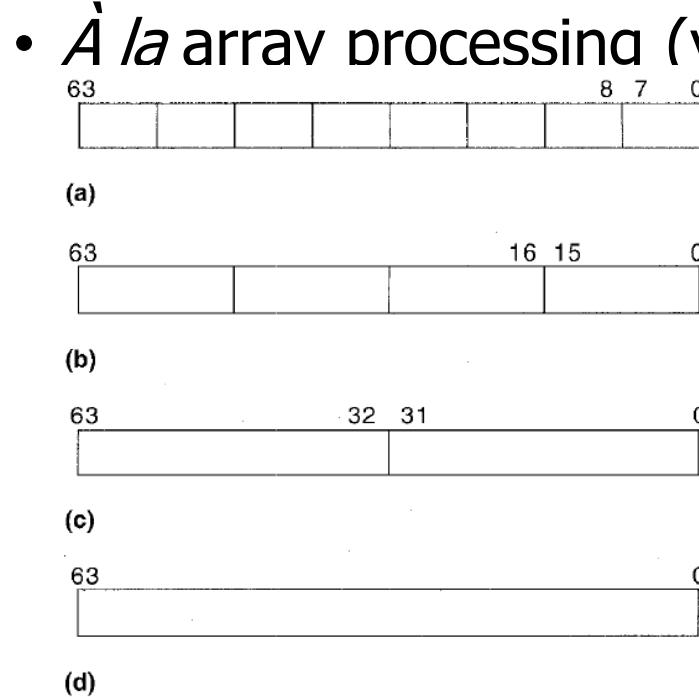


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

a (graphics) operations in mind
No VLEN register

Opcode determines data type: 8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride is always equal to 1.

Peleg and Weiser, “[MMX Technology Extension to the Intel Architecture](#),” IEEE Micro, 1996.

MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image 1 on top of the background in image 2

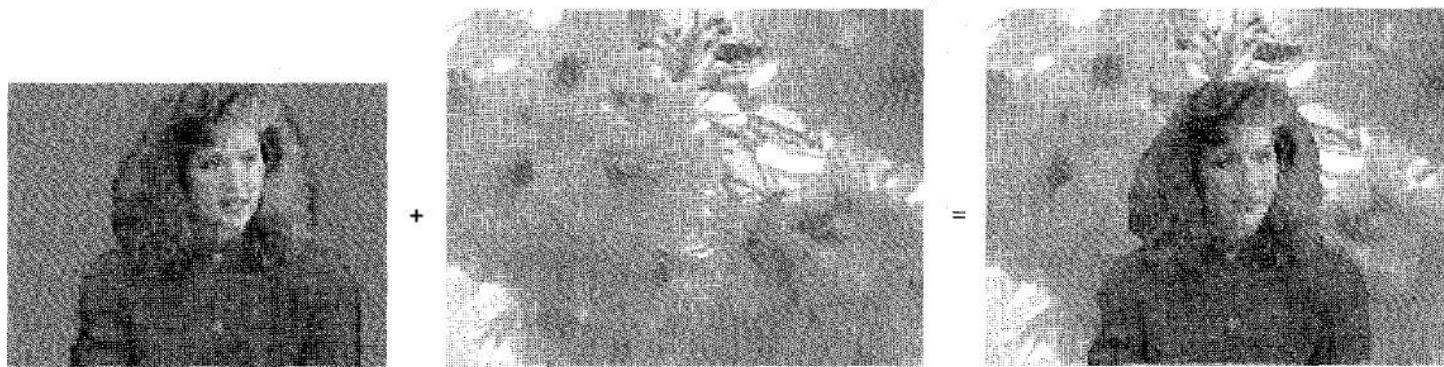


Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

| | | | | | | | | |
|-----|----------|----------|---------|---------|----------|----------|---------|---------|
| MM1 | Blue | Blue | Blue | Blue | Blue | Blue | Blue | Blue |
| MM3 | X7!=blue | X6!=blue | X5=blue | X4=blue | X3!=blue | X2!=blue | X1=blue | X0=blue |
| MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF |

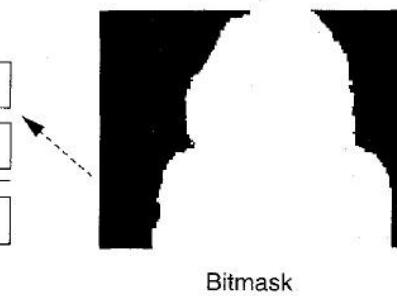


Figure 9. Generating the selection bit mask.

Peleg and Weiser, “MMX Technology Extension to the Intel Architecture,” IEEE Micro, 1996.

MMX Example: Image Overlaying (II)

Y = Blossom image

X = Woman's image

PAND MM4, MM1

| | | | | | | | | | | | | | | | | | |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| MM4 | Y ₇ | Y ₆ | Y ₅ | Y ₄ | Y ₃ | Y ₂ | Y ₁ | Y ₀ | MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF |
| MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | MM3 | X ₇ | X ₆ | X ₅ | X ₄ | X ₃ | X ₂ | X ₁ | X ₀ |
| MM4 | 0x0000 | 0x0000 | Y ₅ | Y ₄ | 0x0000 | 0x0000 | Y ₁ | Y ₀ | MM1 | X ₇ | X ₆ | 0x0000 | 0x0000 | X ₃ | X ₂ | 0x0000 | 0x0000 |

PANDN MM1, MM3

PANDN MM1, MM3

POR MM4, MM1

MM4 X₇ X₆ Y₅ Y₄ X₃ X₂ Y₁ Y₀



Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq mm3, mem1 /* Load eight pixels from  
                   woman's image  
Movq mm4, mem2 /* Load eight pixels from the  
                   blossom image  
Pcmpeqb mm1, mm3  
Pand mm4, mm1  
Pandn mm1, mm3  
Por mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.



arm

SVE Fundamentals

CODES@OEHI Hackathon 2019

olly.perks@arm.com
29th October 2019

What makes it a Scalable Vector Extension?

- **There is no preferred vector length**
 - The vector length (VL) is a hardware choice, 128-2048b, in increments of 128b
 - A Vector Length Agnostic (VLA) programming adjusts dynamically to the available VL
- **SVE addresses traditional barriers to auto-vectorization**
 - Software-managed speculative vectorization of uncounted loops
 - Extract more data-level parallelism (DLP) from existing C/C++/Fortran source code
- **SVE is a new approach to vectorization, not an iteration on existing ISAs (e.g. NEON)**
 - SVE is a separate, optional extension with a new set of instruction encodings
 - Initial focus is HPC and general-purpose server, not media/image processing

SVE vs Traditional ISA

How do we compute data which has ten chunks of 8-bytes?

Aarch64 (scalar)

- ❑ Ten iterations over an 8-byte register



NEON (128-bit vector engine)

- ❑ Four iterations over a 16-byte register + two iterations of a drain loop over a 8-byte register



SVE (VLA vector engine)

- ❑ Three iterations over a 32-byte **VLA register** with an adjustable **predicate**

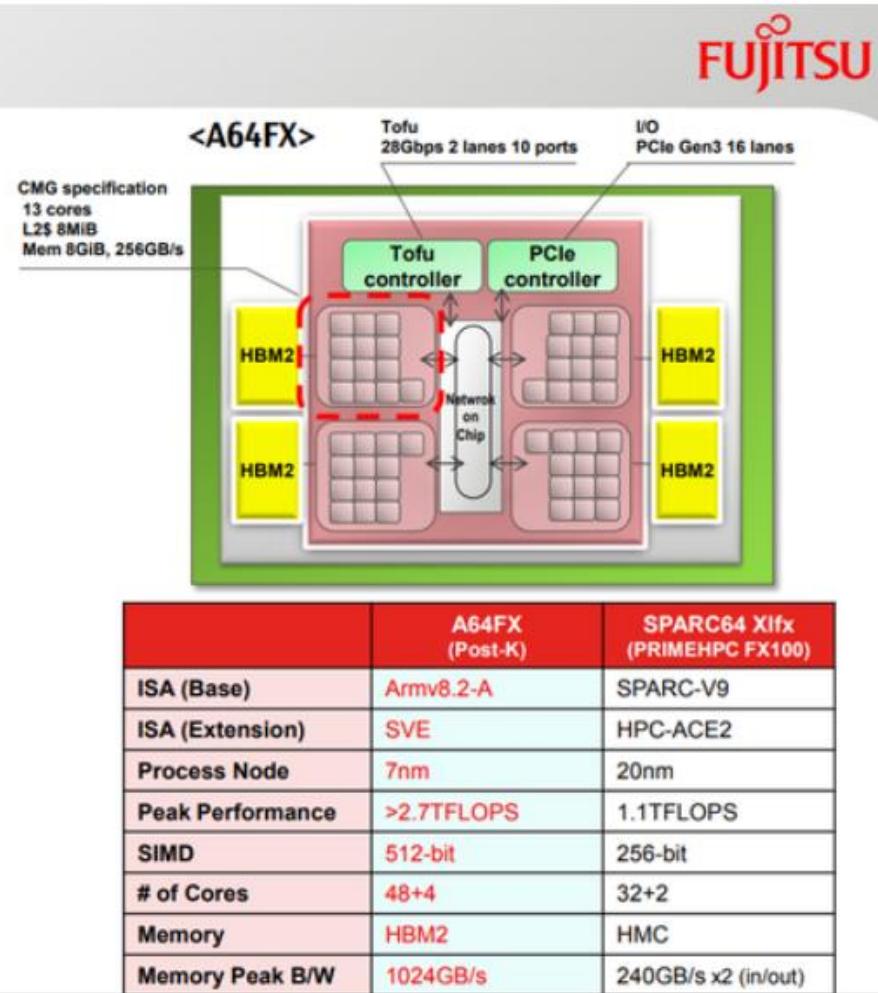


Post-K Supercomputer goes Arm with SVE

A64FX Chip Overview

■ Architecture Features

- Armv8.2-A (AArch64 only)
 - SVE 512-bit wide SIMD
 - 48 computing cores + 4 assistant cores*
- *All the cores are identical
- HBM2 32GiB
 - Tofu 6D Mesh/Torus 28Gbps x 2 lanes x 10 ports
 - PCIe Gen3 16 lanes



■ 7nm FinFET

- 8,786M transistors
- 594 package signal pins

■ Peak Performance (Efficiency)

- >2.7TFLOPS (>90%@DGEMM)
- Memory B/W 1024GB/s (>80%@Stream Triad)

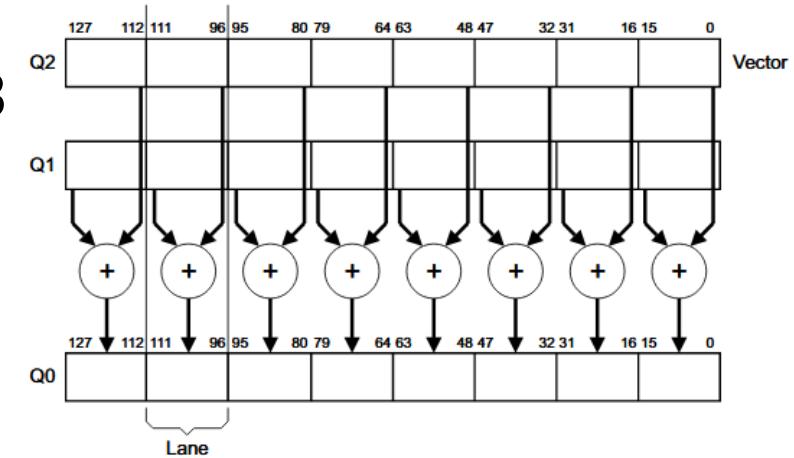
| | A64FX (Post-K) | SPARC64 XIfx (PRIMEHPC FX100) |
|------------------|----------------|-------------------------------|
| ISA (Base) | Armv8.2-A | SPARC-V9 |
| ISA (Extension) | SVE | HPC-ACE2 |
| Process Node | 7nm | 20nm |
| Peak Performance | >2.7TFLOPS | 1.1TFLOPS |
| SIMD | 512-bit | 256-bit |
| # of Cores | 48+4 | 32+2 |
| Memory | HBM2 | HMC |
| Memory Peak B/W | 1024GB/s | 240GB/s x2 (in/out) |



slides from Fujitsu

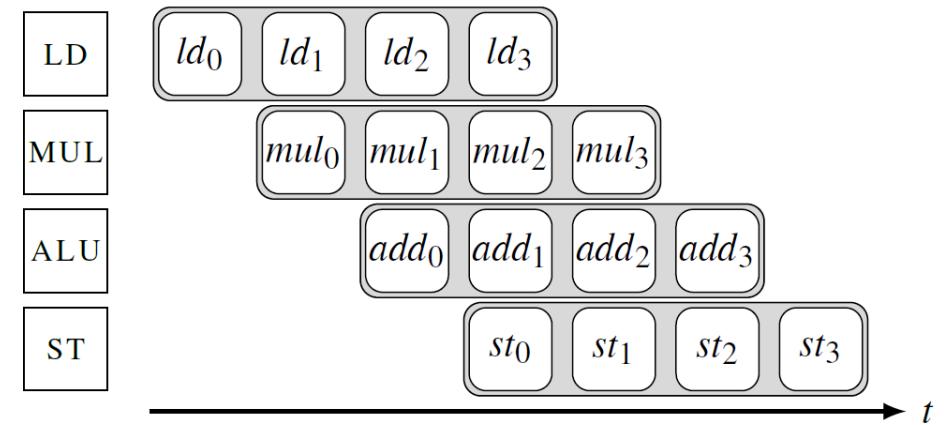
Packed-SIMD is no Vector-SIMD

- Limited instruction set
 - No strided load/store or scatter/gather
- Limited vector register length
 - Requires superscalar dispatch to keep functional units busy
 - Loop unrolling to hide latencies
- Vector length is set in stone
 - Very short vectors, e.g., ARM NEON has a VLEN of 128
 - Cray 1 used vectors of 1024 bits, in 1976
 - VLEN is encoded in the instruction itself
 - **VADD.I16 Q0, Q1, Q2**
 - **Scalability problem**: a wider VLEN requires a new ISA.
 - Widest extension to date: **Intel AVX-512**



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



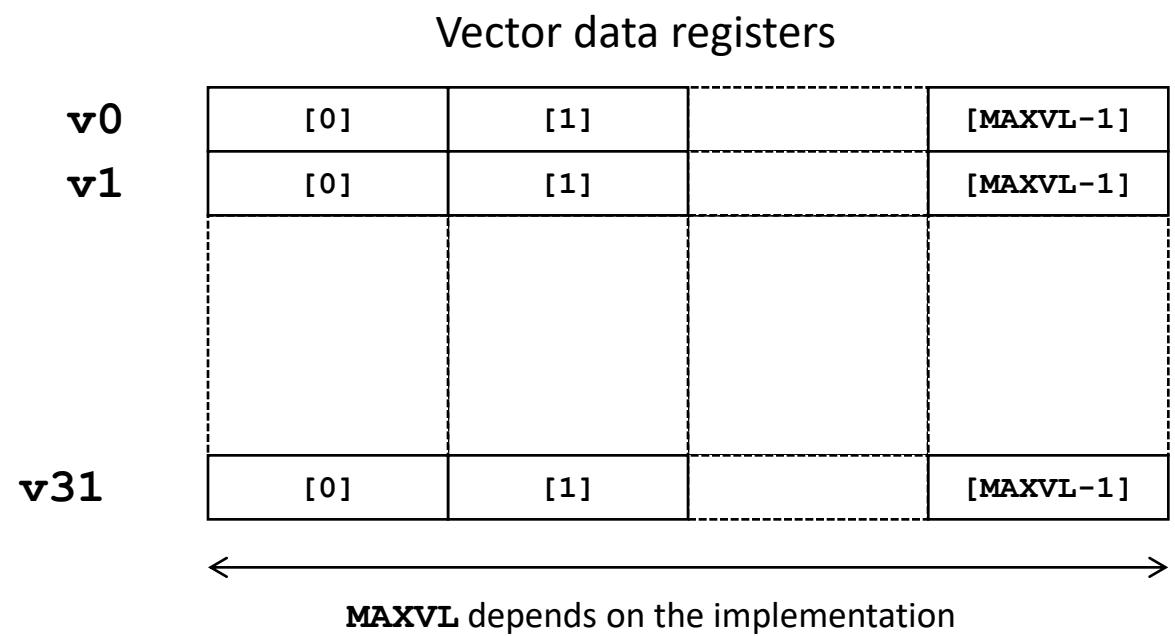
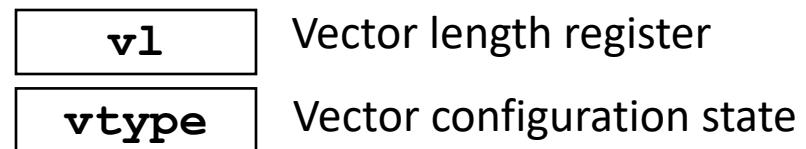
RISC-V Vector Standard

- Being added as a standard extension to the RISC-V ISA
- An updated form of Cray-style vectors for modern microprocessors
- Still a WIP, so details might (and are) changing before being standardized
- The latest version of the spec is kept at <https://github.com/riscv/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
- Vector length register **v1** 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



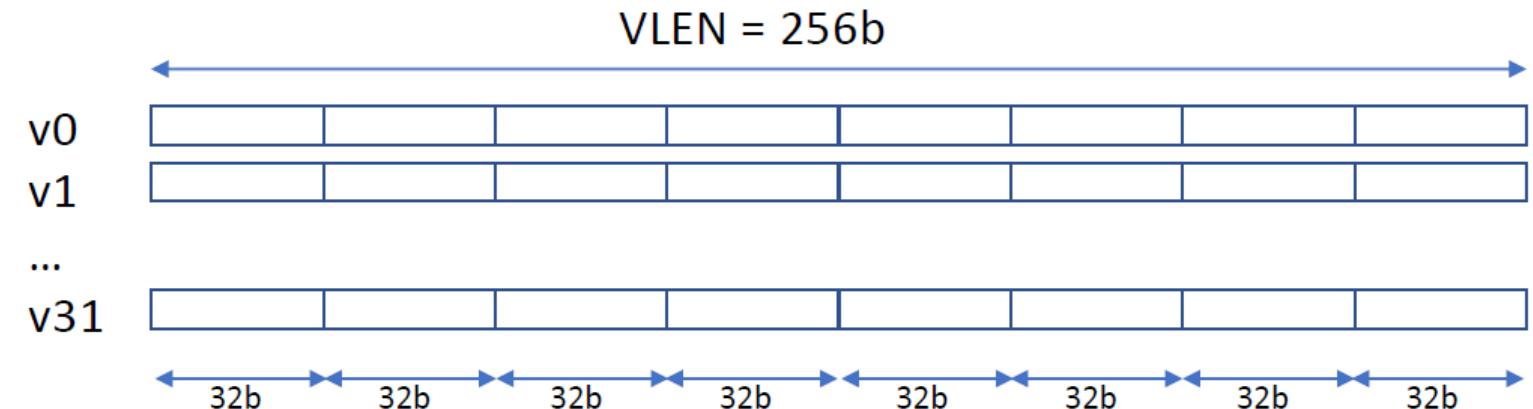
Register State: 32 registers of VLEN bits

- 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

81

```
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**

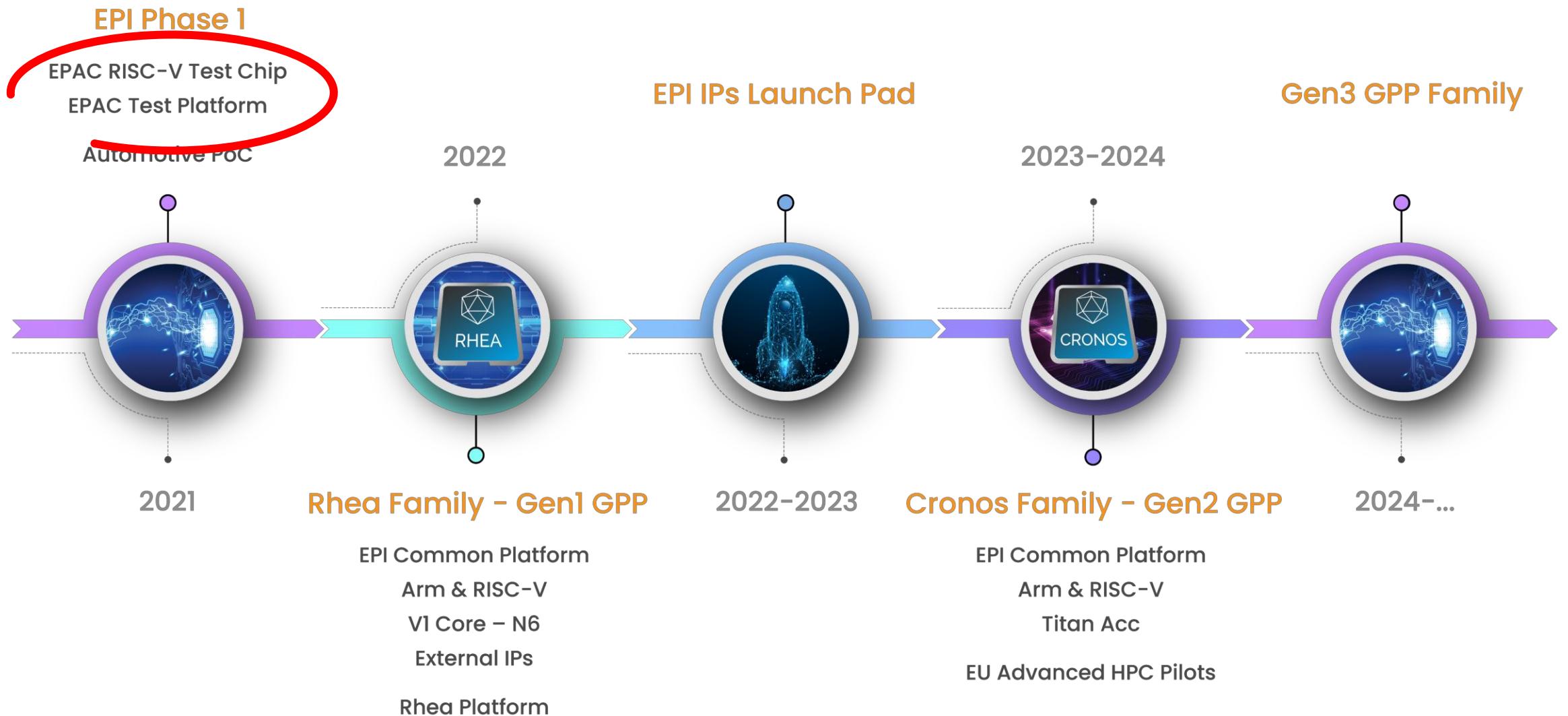
Register **a1** holds **@A[0]**

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

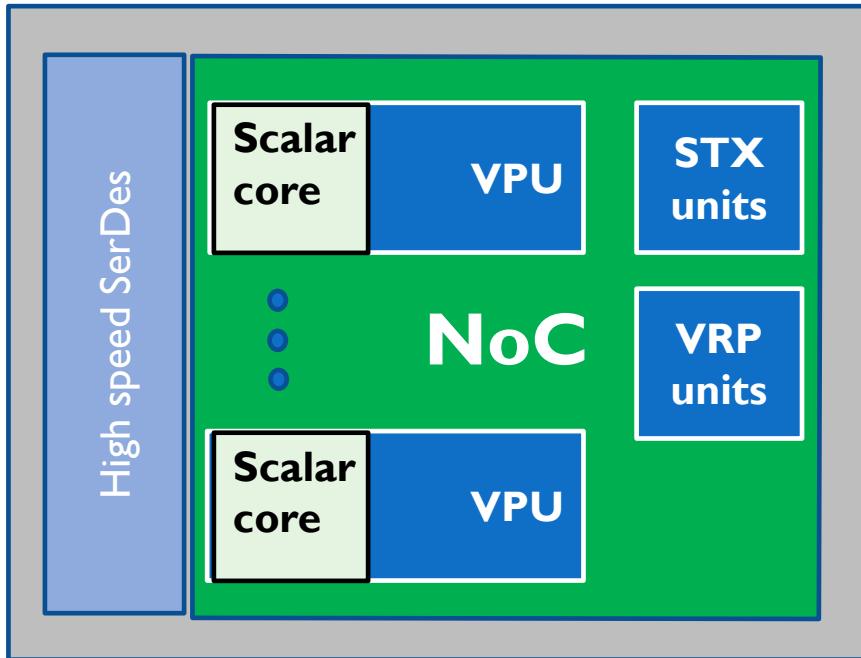
Register **a3** holds **@C[0]**

```
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
```

THE EUROPEAN PROCESSOR INITIATIVE ROADMAP:

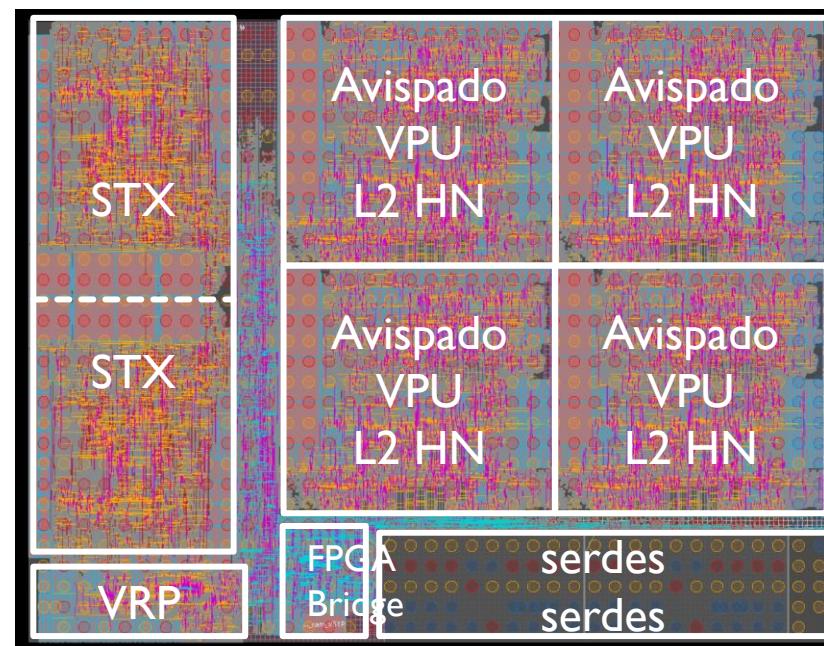


EPI ACCELERATOR (EPAC) DEMONSTRATOR TEST CHIP



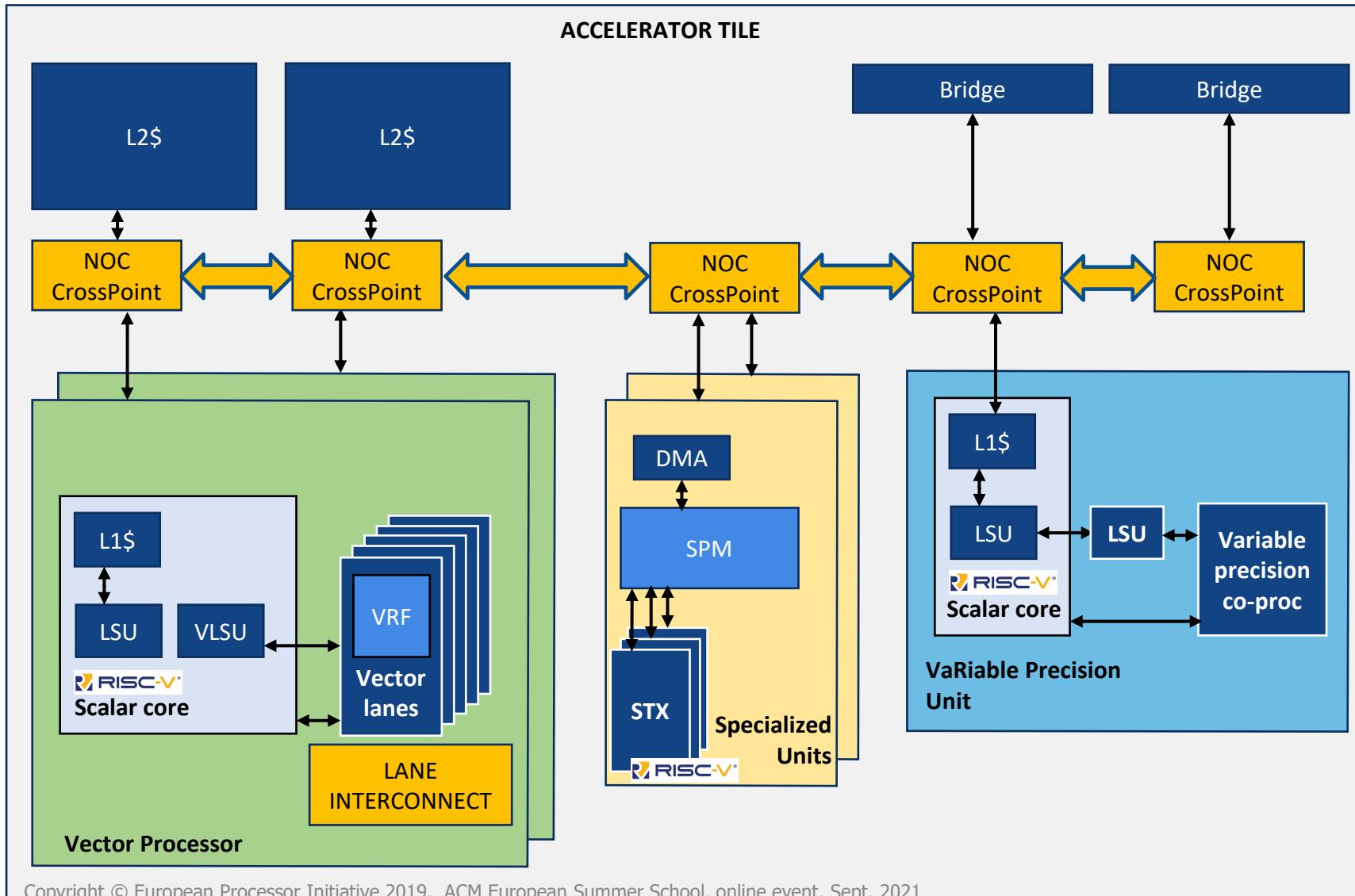
Architecture concept:

- **RISC-V** vector processos (scalar core & VPU) + "STX" accelerators + "VRP" accelerators
- On-chip L1, L2 + off-chip HBM + DDR PHY
- Targets 16 DP GFLOPS **per core** (vector processor only)



- 22 nm FDSOI
- Taped out in April 2021
- Total area:
5943 X 4593 um²
(27.297 mm²)

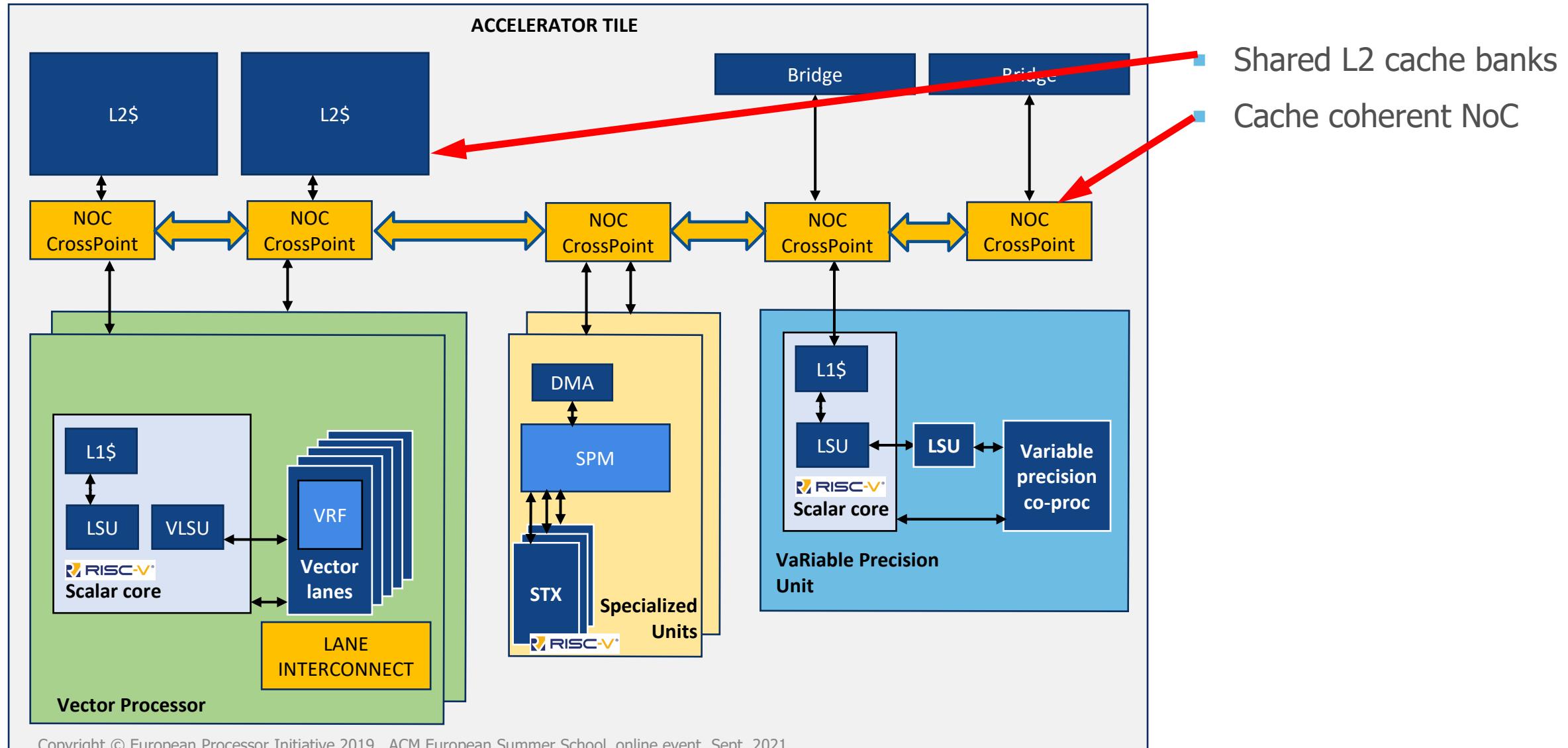
EPAC ARCHITECTURE VIEW



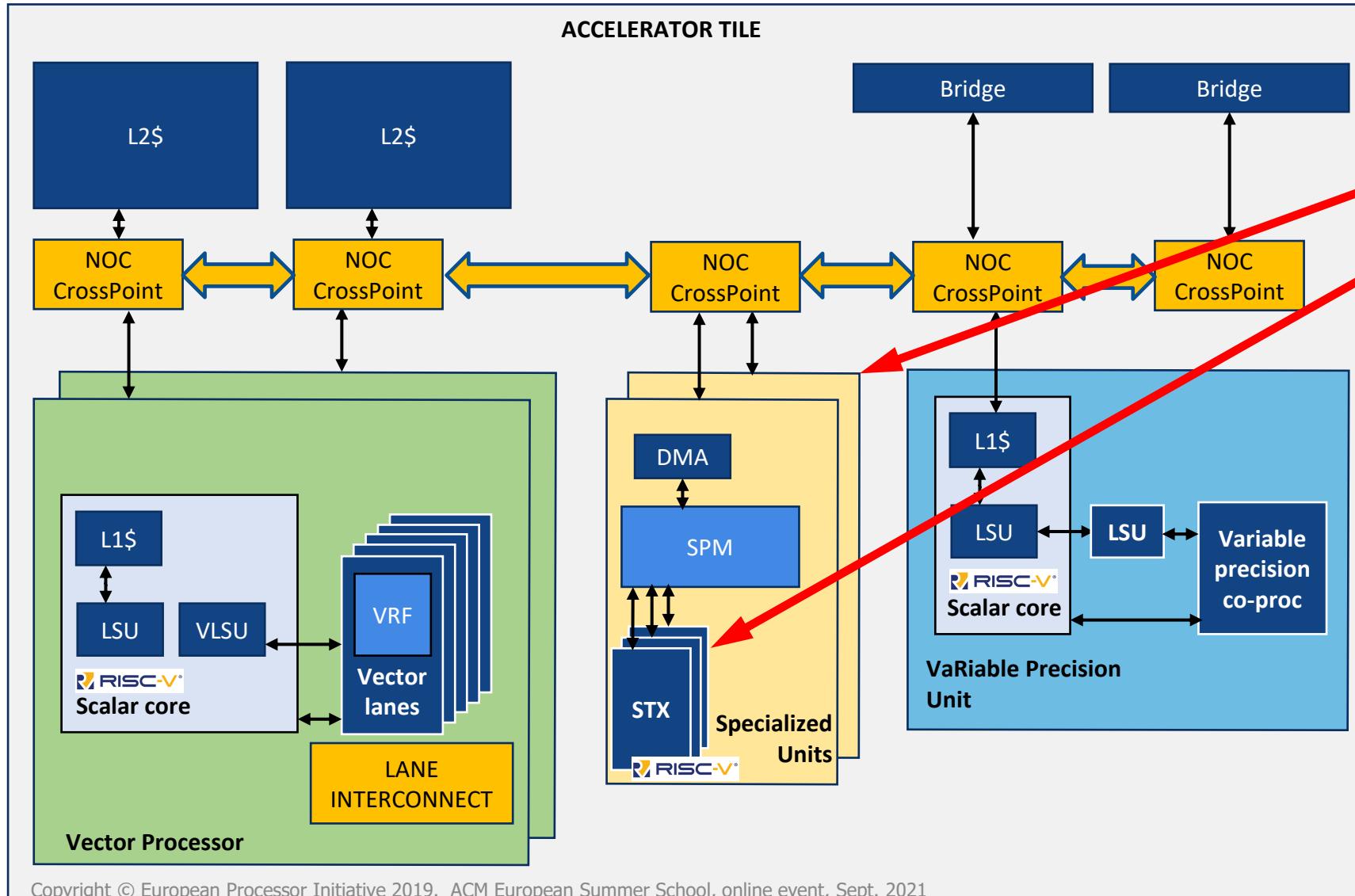
CONTRIBUTING PARTNERS:

- BSC
- CEA
- Chalmers
- E4
- ETH Zurich
- Extoll
- FORTH
- Fraunhofer
- Semidynamics
- Univ. of Zagreb

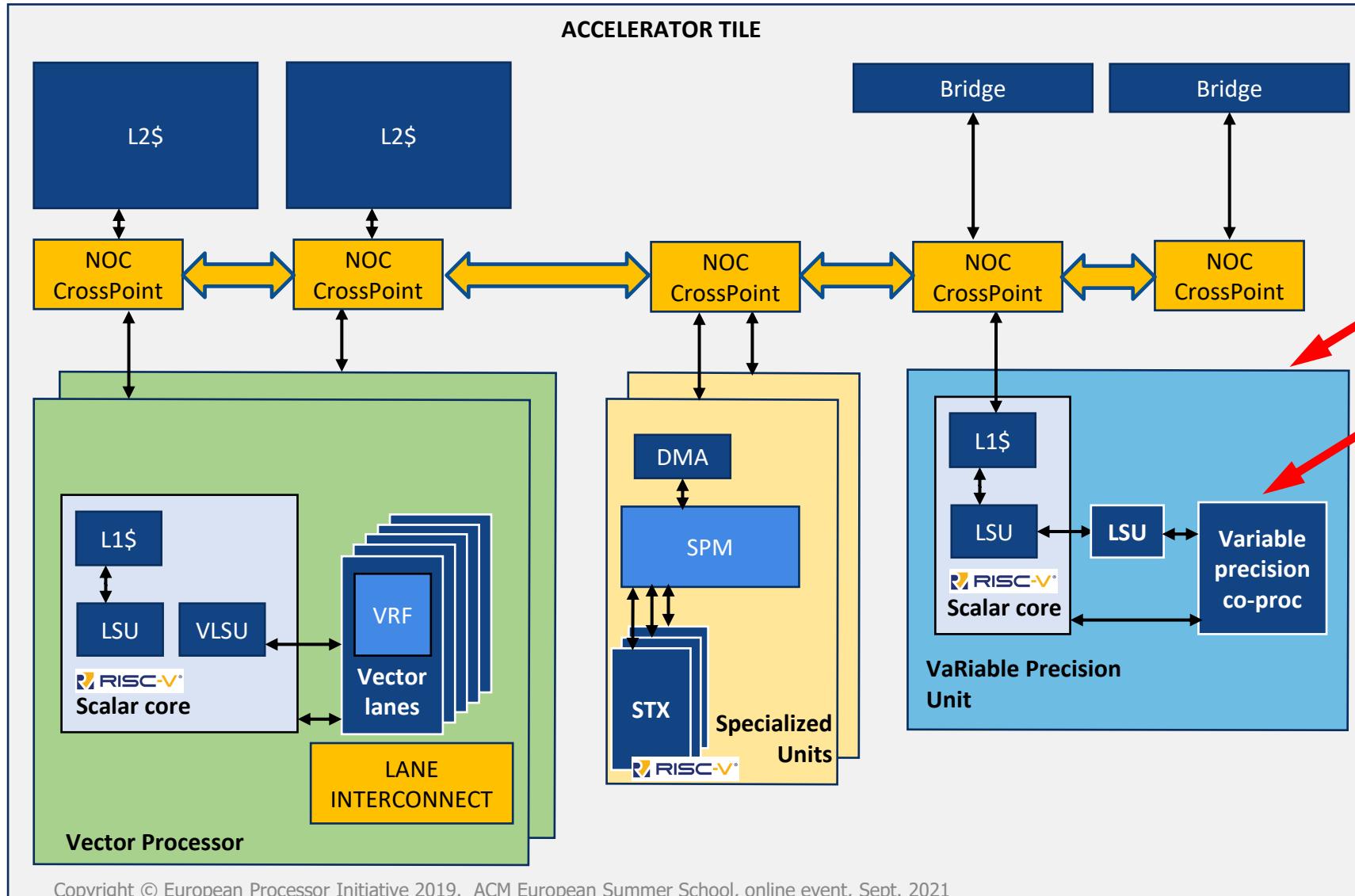
EPAC ARCHITECTURE VIEW



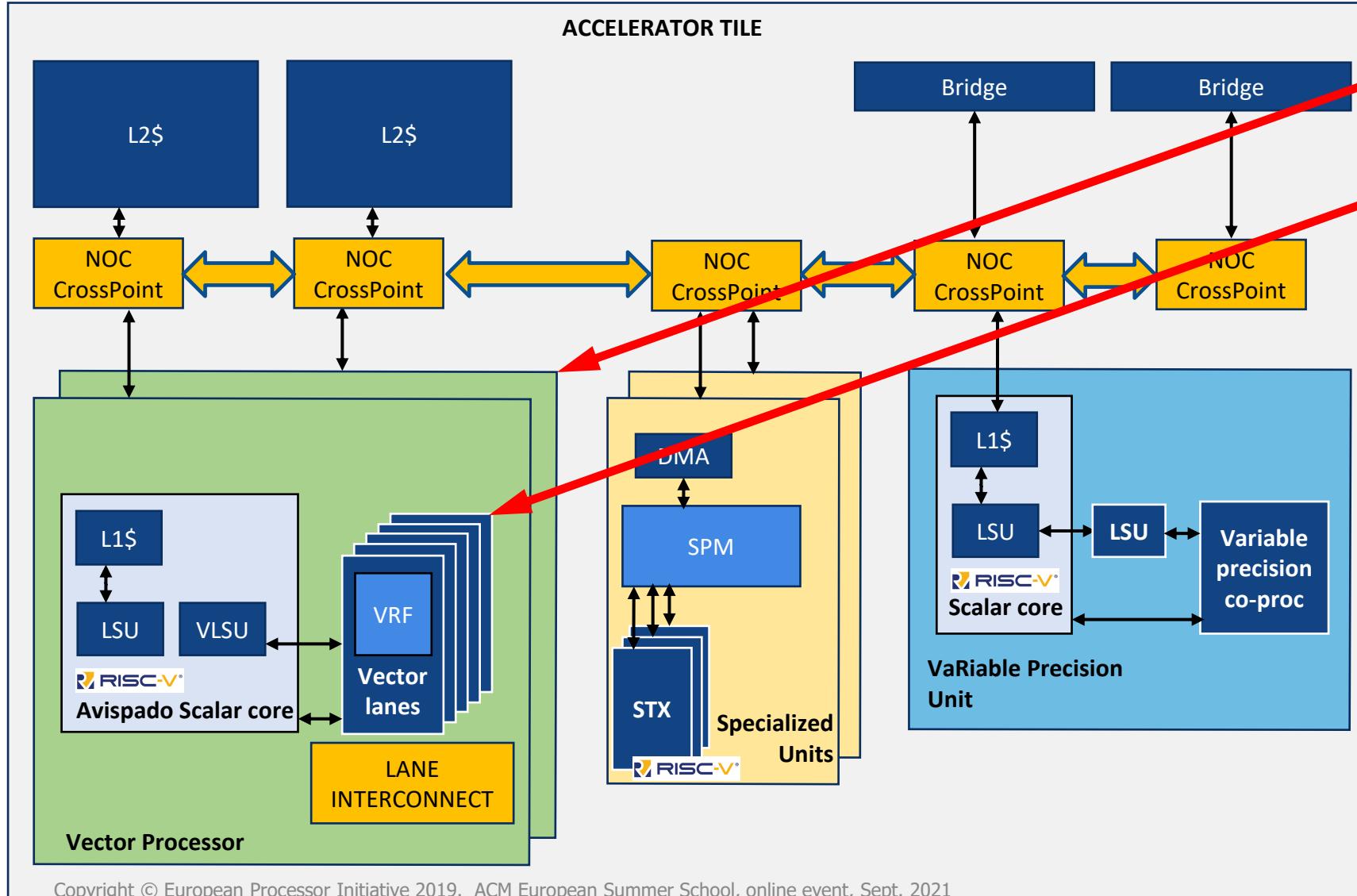
EPAC ARCHITECTURE VIEW



EPAC ARCHITECTURE VIEW



EPAC ARCHITECTURE VIEW



- Up to 8 **Vector Processors**
- Vector Lanes act as tightly coupled (ISA mapped) acceleration units to the scalar core in the vector processor
- Decoupled scalar core / vector processing unit hardware architecture
- Heavily pipelined
- RISC-V vector extension compliant

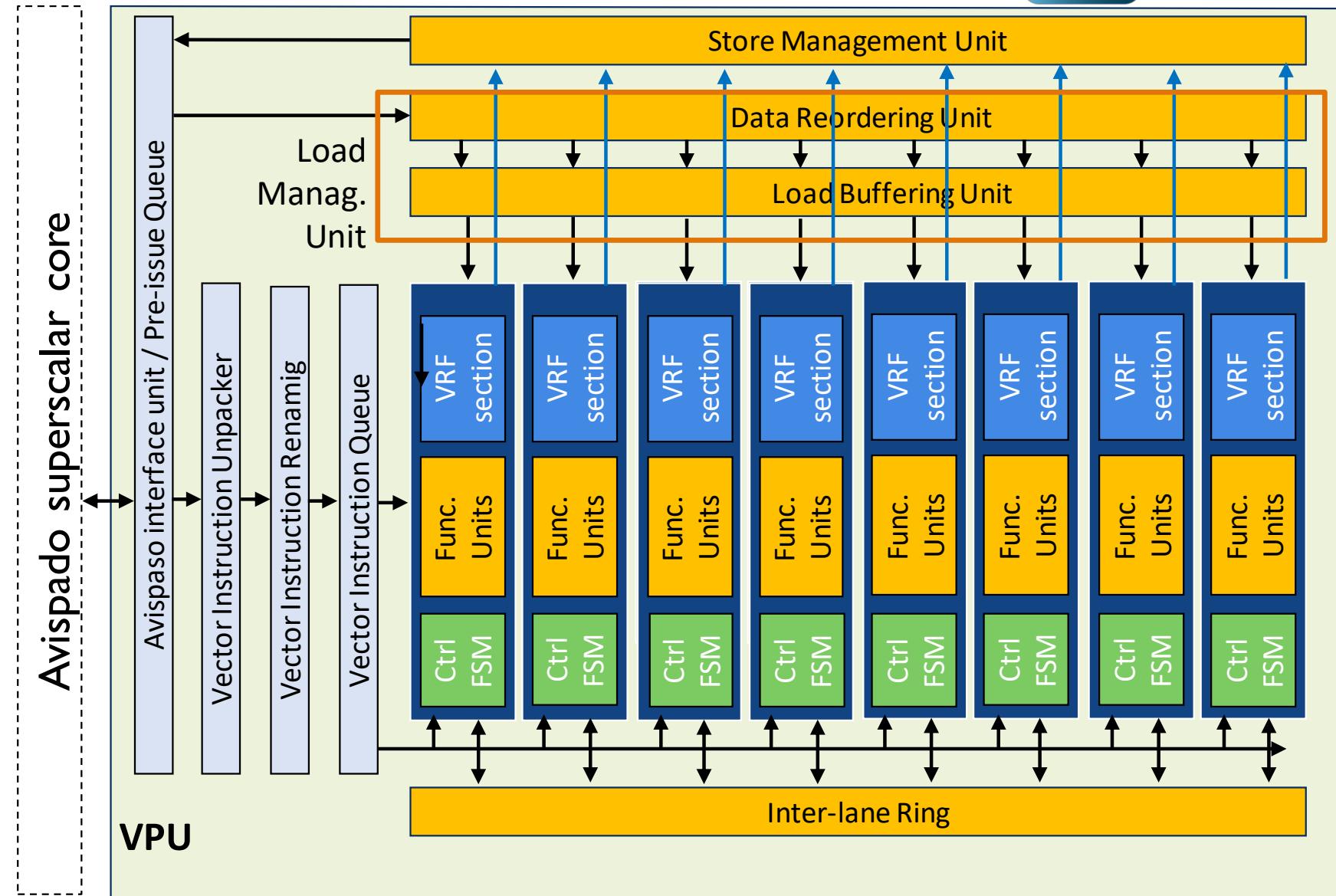
VECTOR PROCESSING UNIT (8 LANES, SIMPLIFIED VIEW)

Can execute

- **8 Double Precision Fused Mul/Add, and**
- **8 Double Precision Element Load/Store (stride one)**

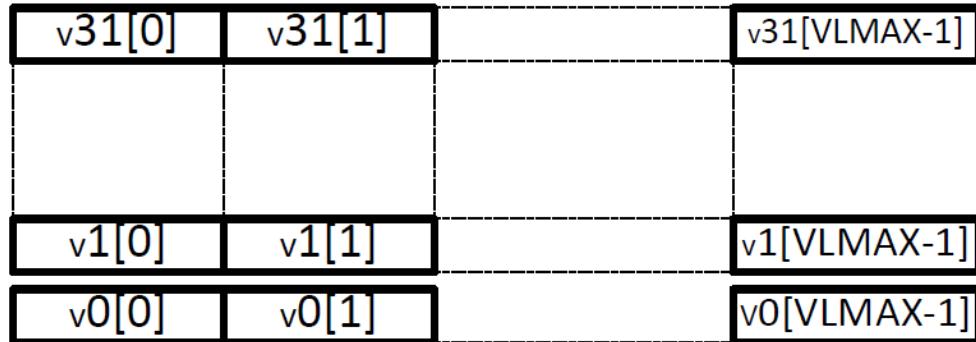
per clock cycle, with no stalls.

Physical Vector Length is up to 256 DP elements.



NOTE: RISC-V VECTOR EXTENSION

32 vector registers



Main features and operations:

- Orthogonal set of vector operations, parity with scalar ISA
- Rich set of integer, fixed-point, and floating-point instructions
- Vector-vector, vector-scalar, and vector-immediate instructions
- Masking on (almost) every vector instruction
- Non-strided and strided loads and stores, gathers, scatters
- Reduction instructions (sum, min/max, and/or, ...)

Vector Control-Status registers:

Vtype

Vtype sets width of element in each vector register (e.g., 16-bit, 32-bit, ...)

VI

Vector length CSR sets number of elements active in each instruction

Vstart

Resumption element after trap

fcsr

Fixed-point rounding mode and saturation flag fields

NOTE: RISC-V VECTOR EXTENSION

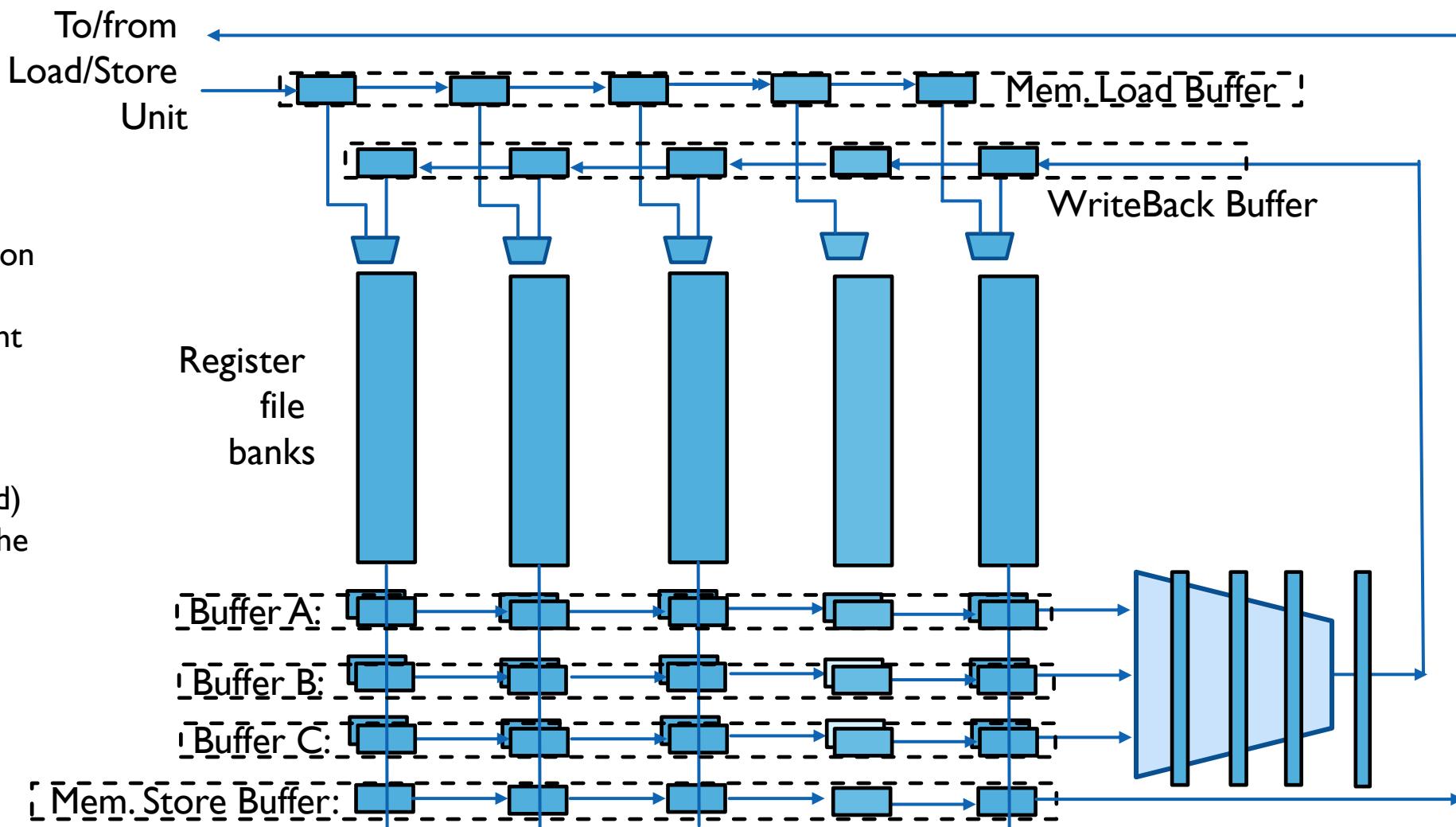
Example vector instruction

- **vfadd.vv vd, vs1, vs2 ,vm**
 - Adds two vector registers, element by elements, and puts result into destination vector register
- **vsetvli rd, rs1, vtypei**
 - Sets vector length VL and element width and type VTYPE
- **vle.v vd, (rs1), vm**
 - Loads a vector from memory into destination vector register, unit-strided
- **vse.v vs3, (rs1), vm**
 - Stores a vector from source vector register to memory, unit-strided

VECTOR LANE MICROARCHITECTURE (SIMPLIFIED VIEW)

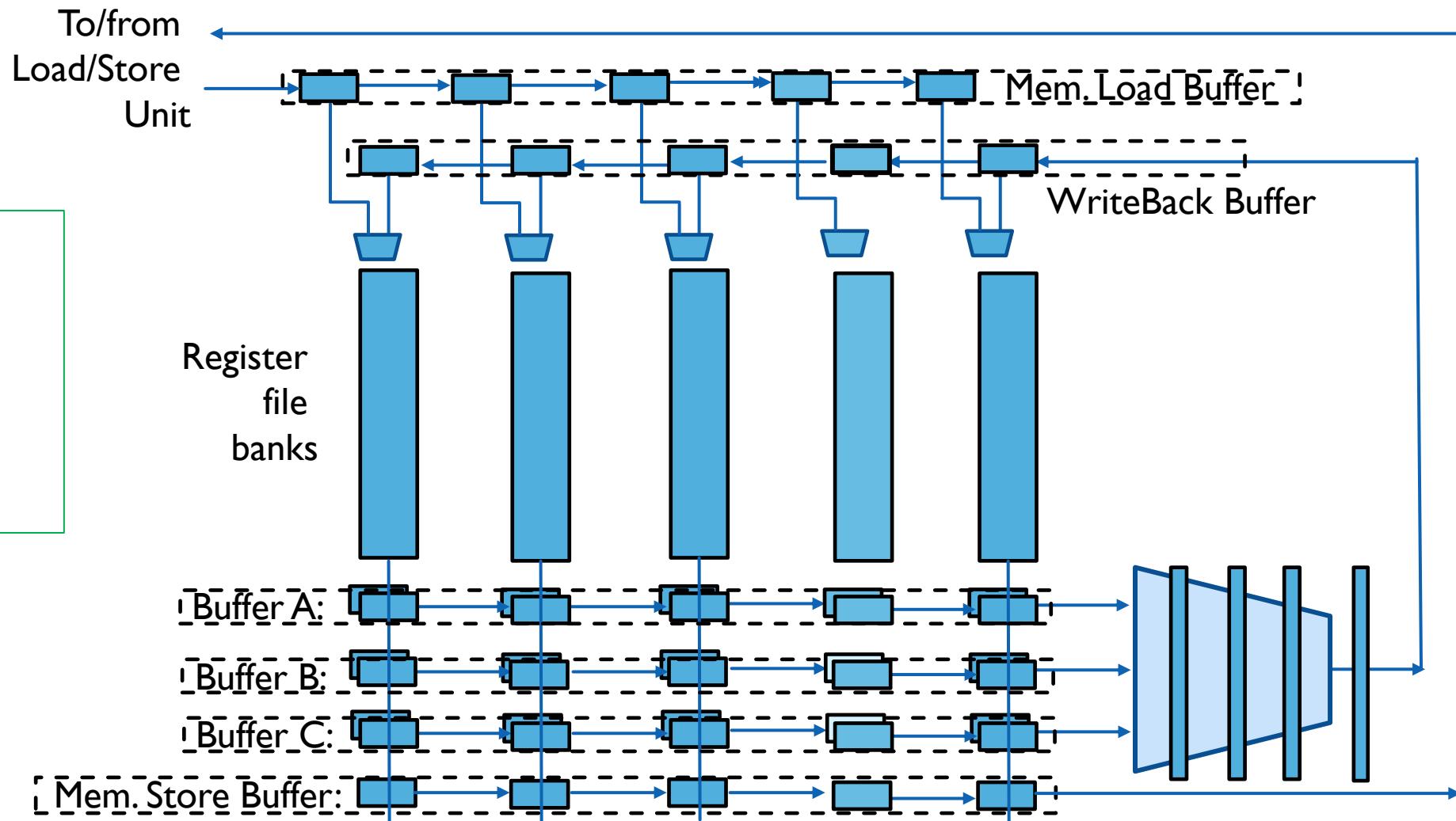
Buffer A, B, C: operand buffers,
WriteBack Buffer: holding operation results
Store Buffer: holding data to be sent to memory
Load buffer: holding data coming from memory

Buffers A,B,C are doubled (shadowed) to allow single-cycle full refill while the shadow buffer is being consumed.



VECTOR ARITHMETIC OPERATION EXECUTION

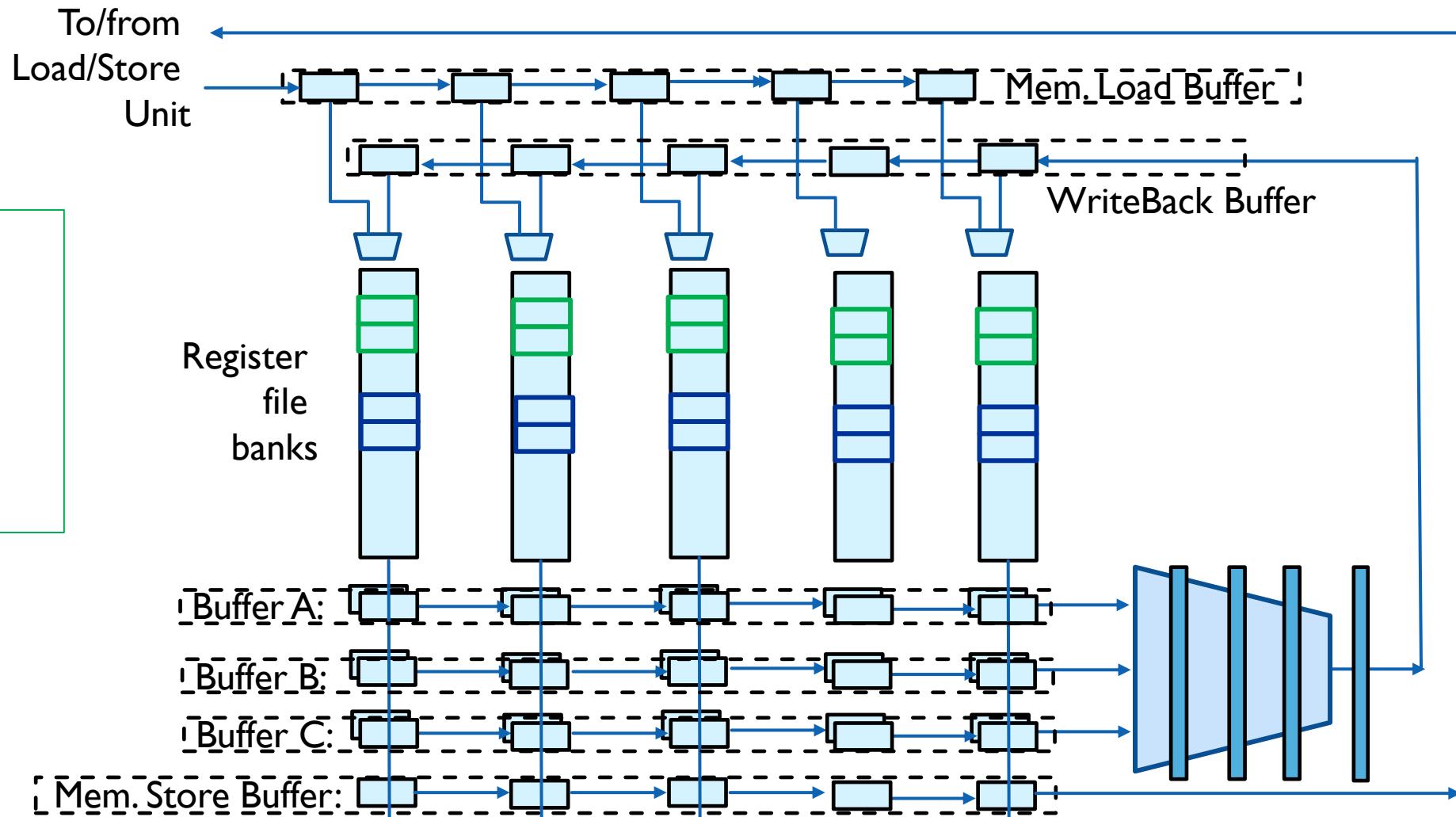
```
vfadd.vv v0, v1, v2
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

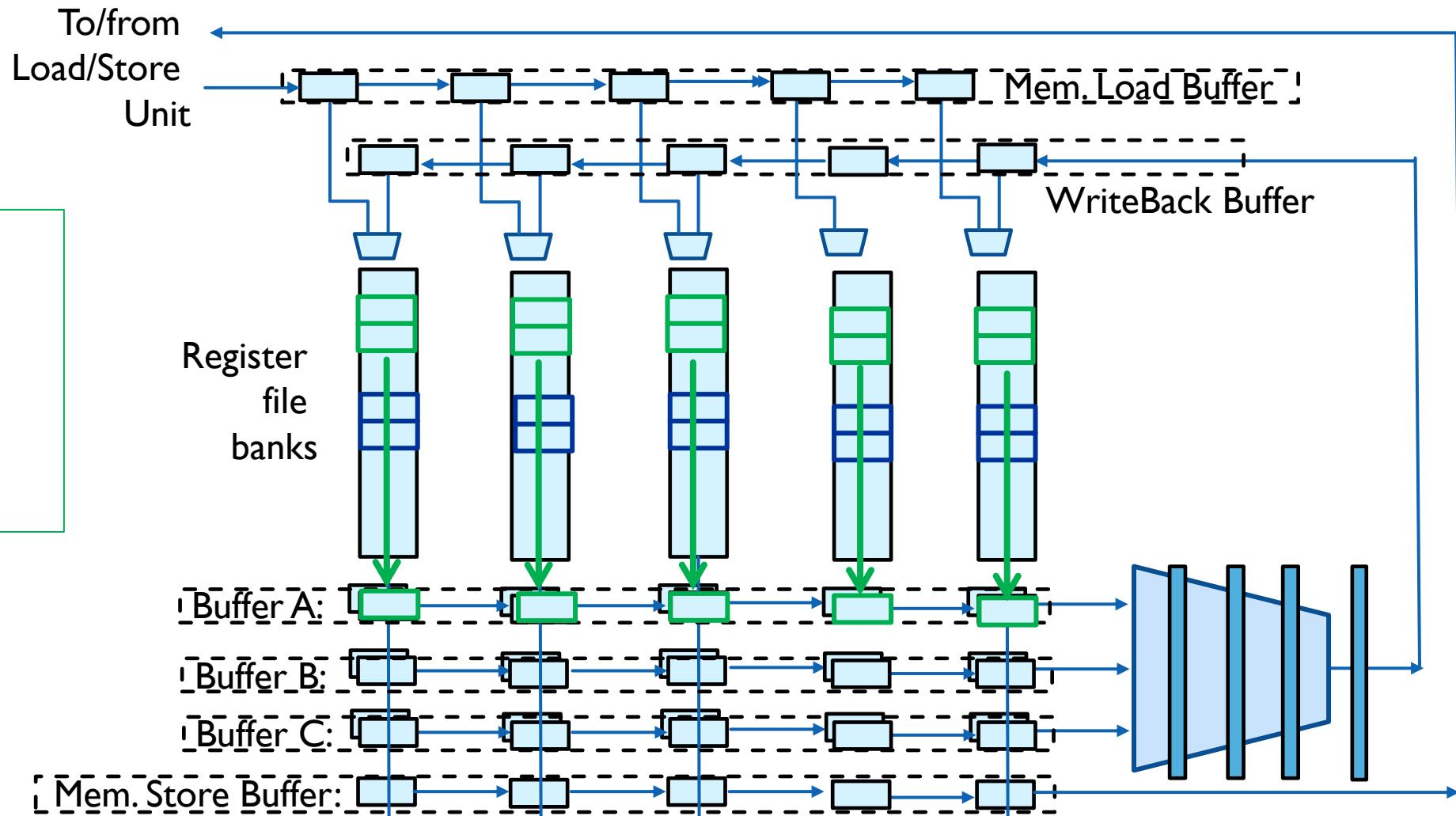
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

vfadd.vv **v0**, **v1**, **v2**

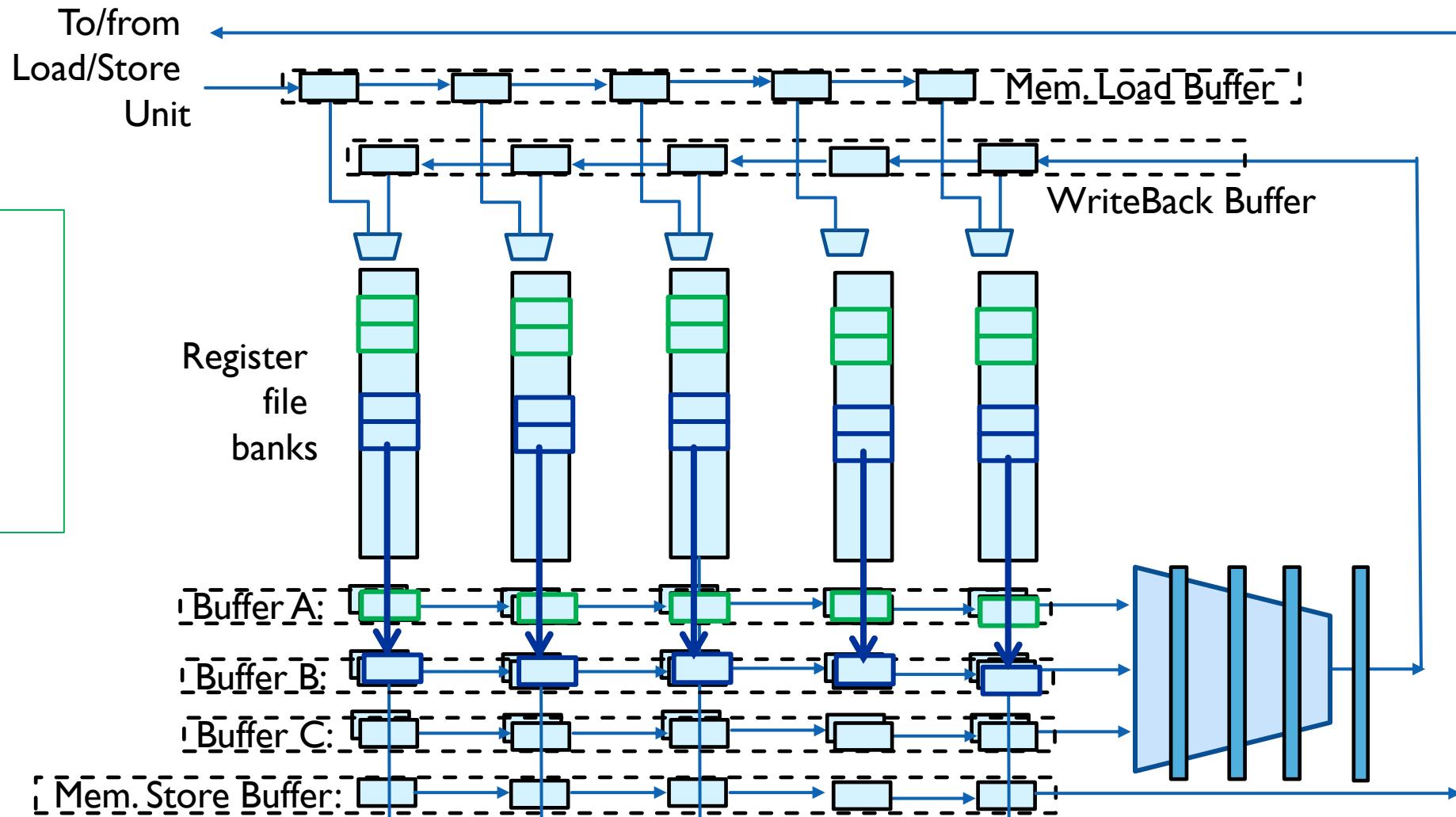
```
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

vfadd.vv **v0**, **v1**, **v2**

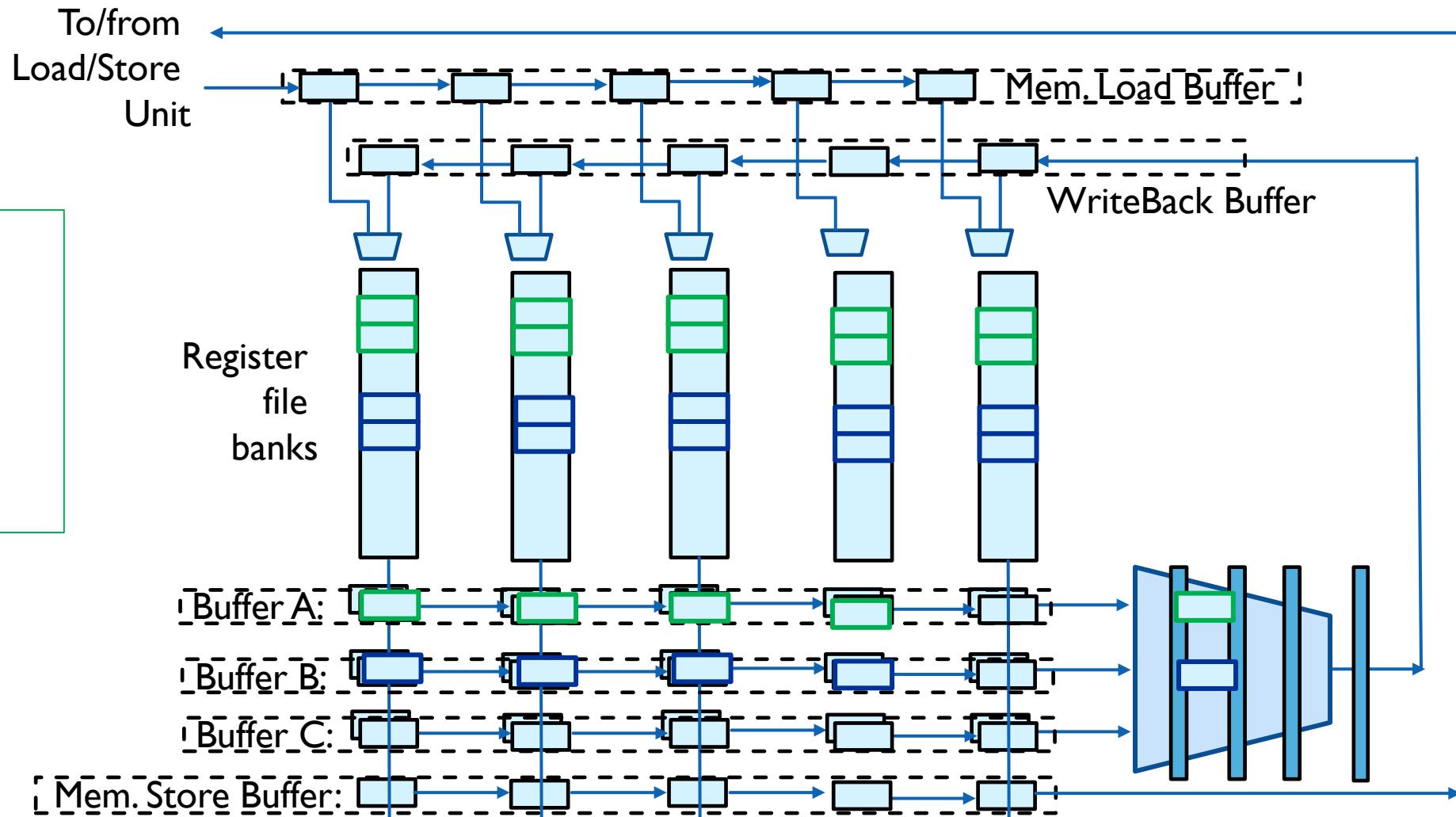
```
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

vfadd.vv **v0**, **v1**, **v2**

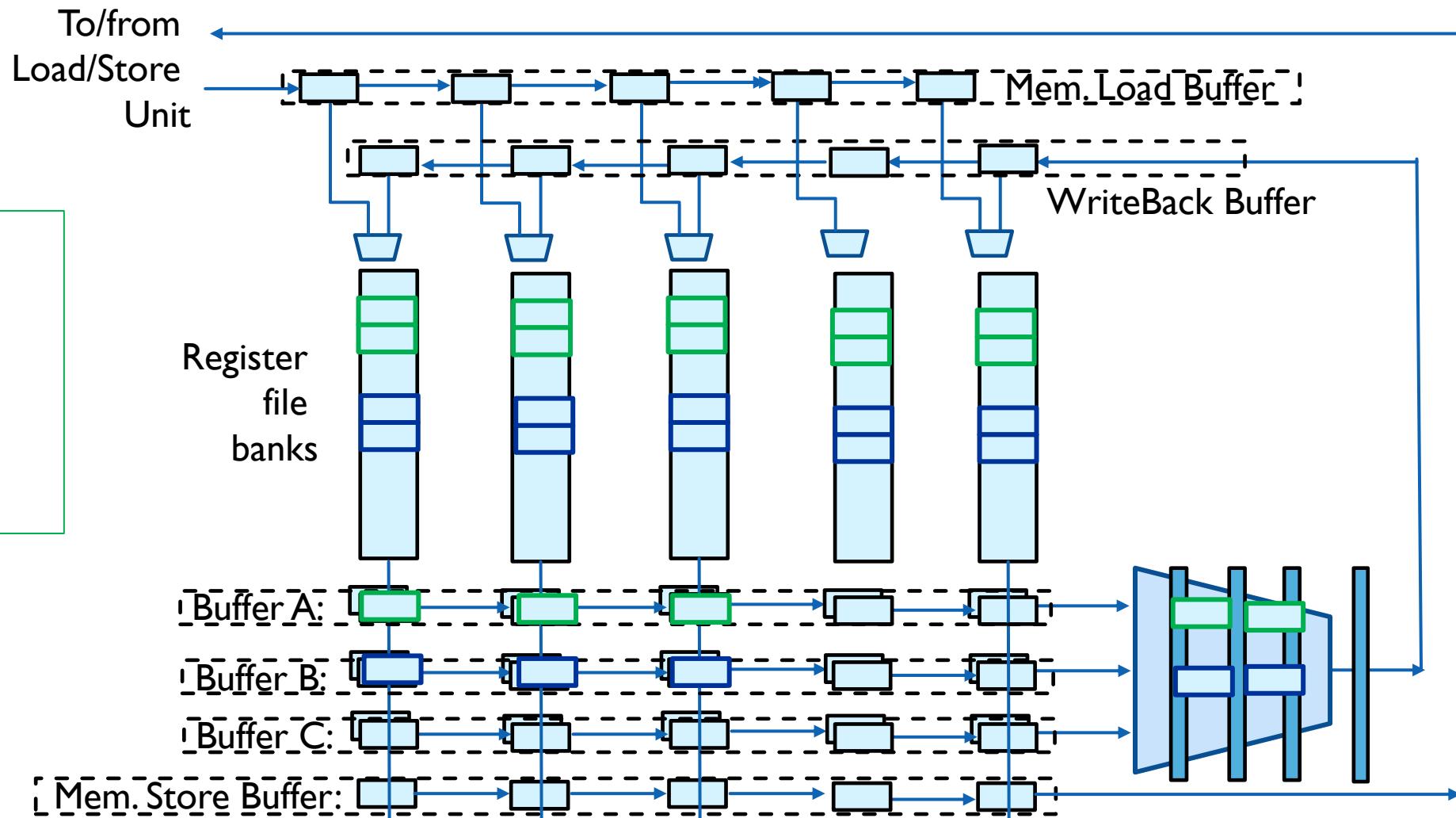
```
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

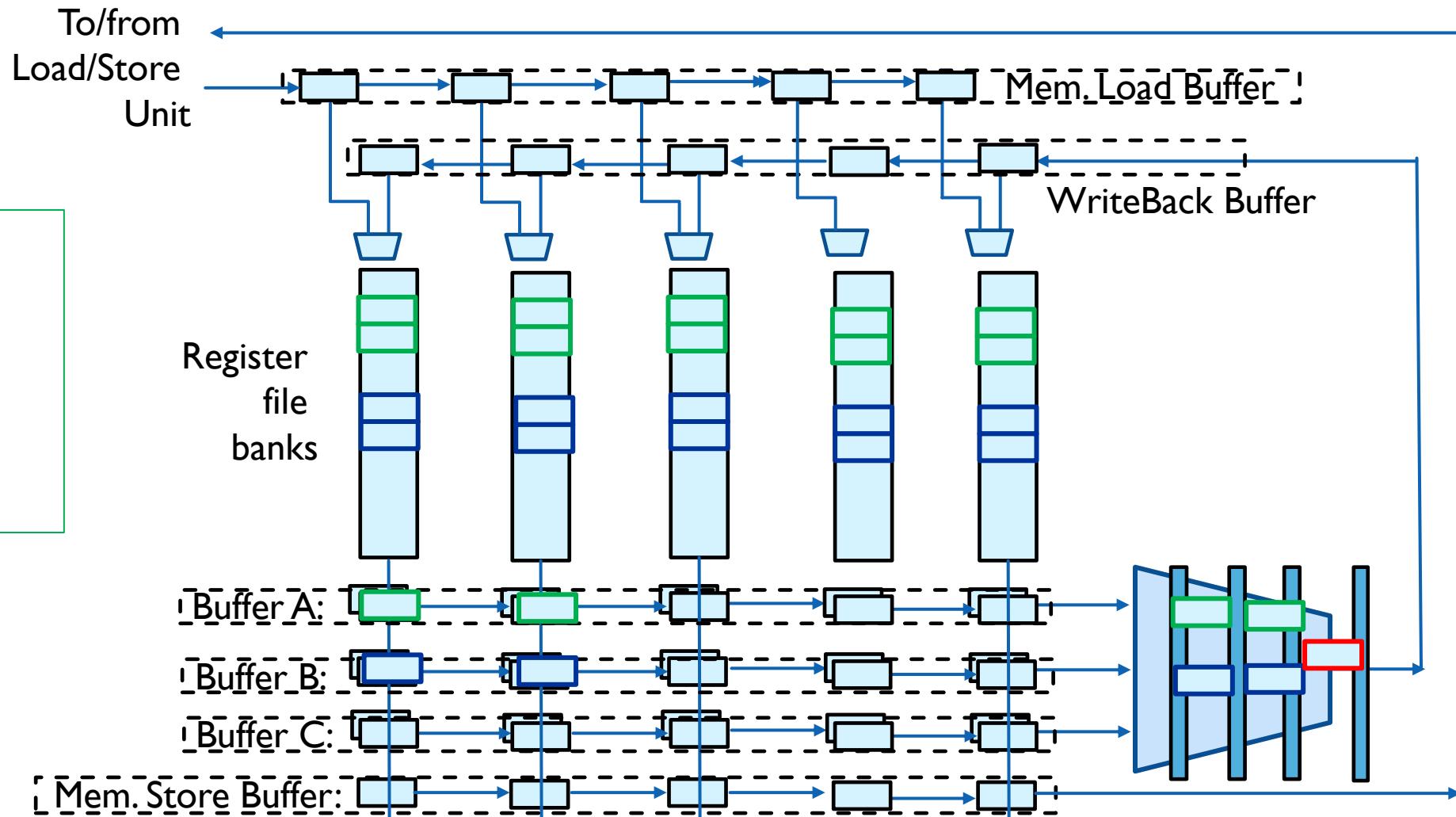
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

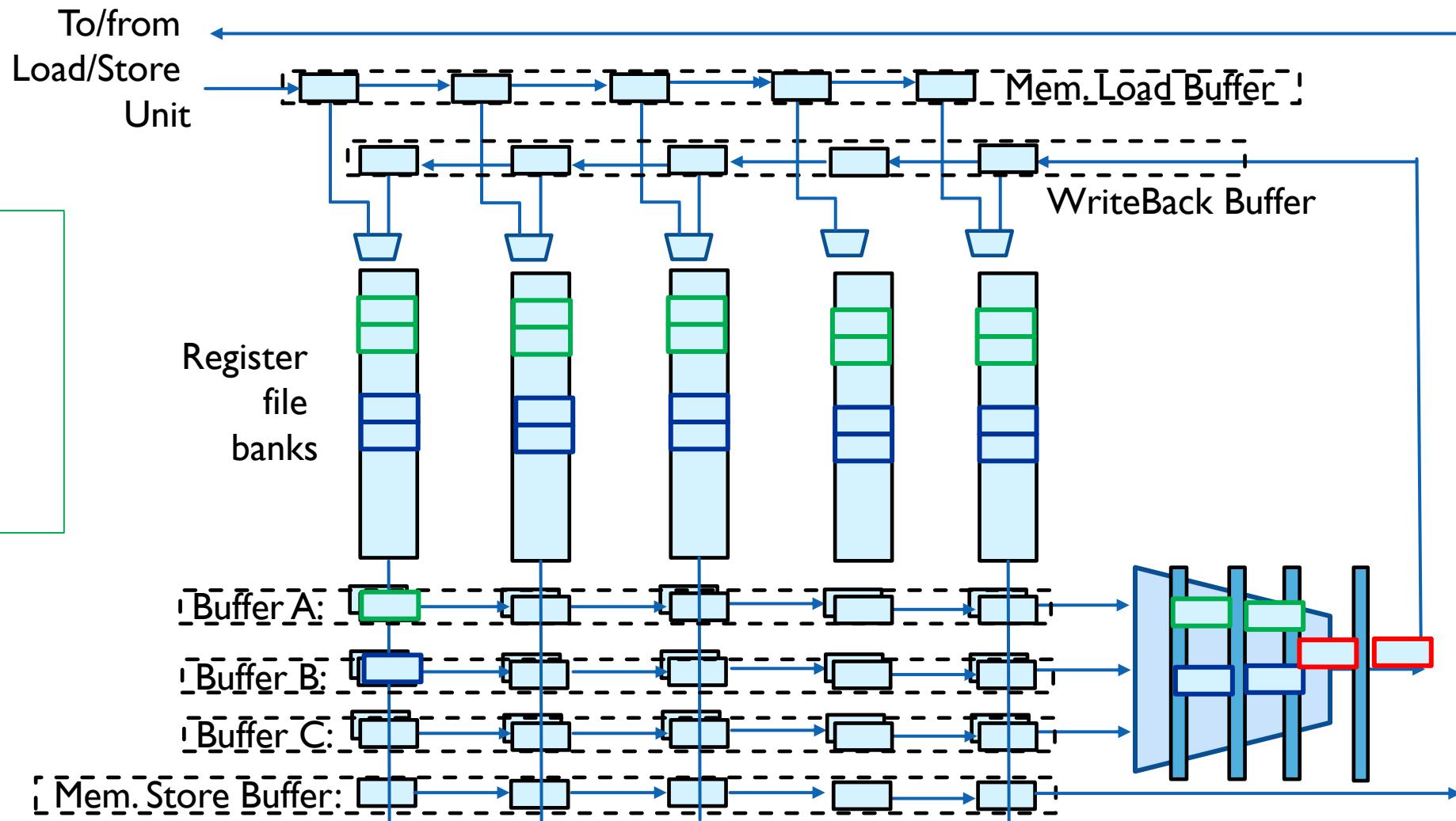
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

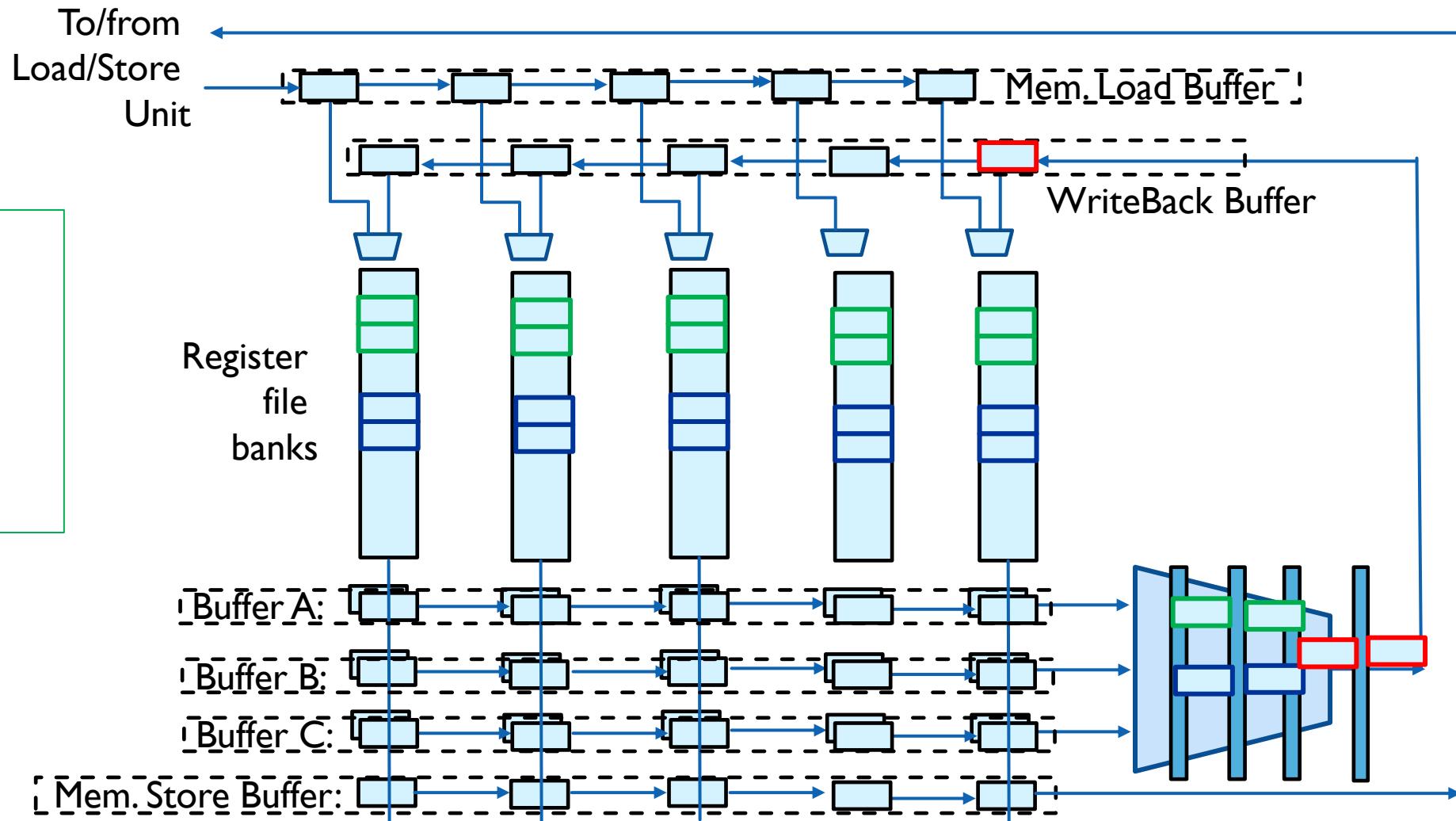
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

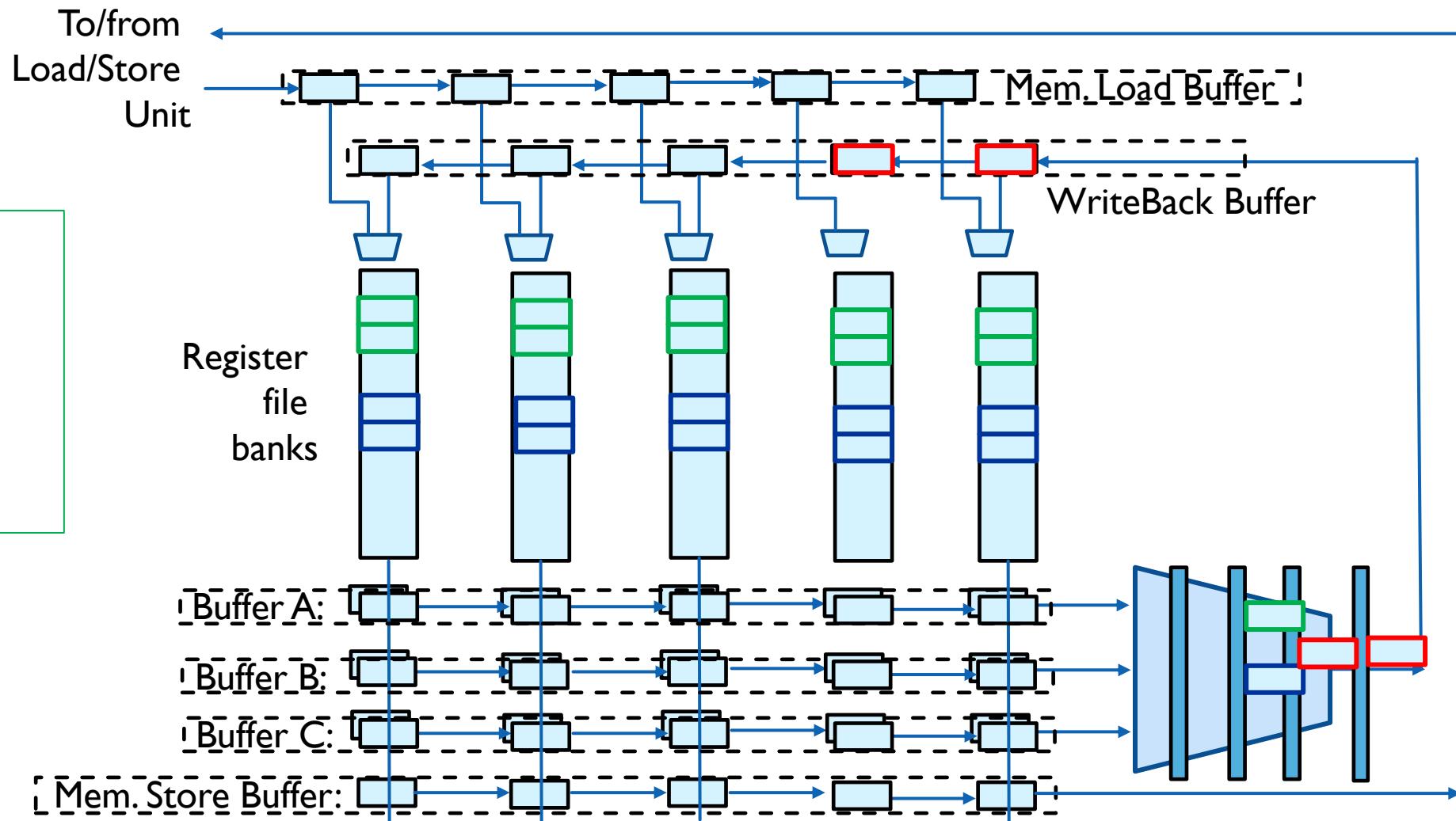
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

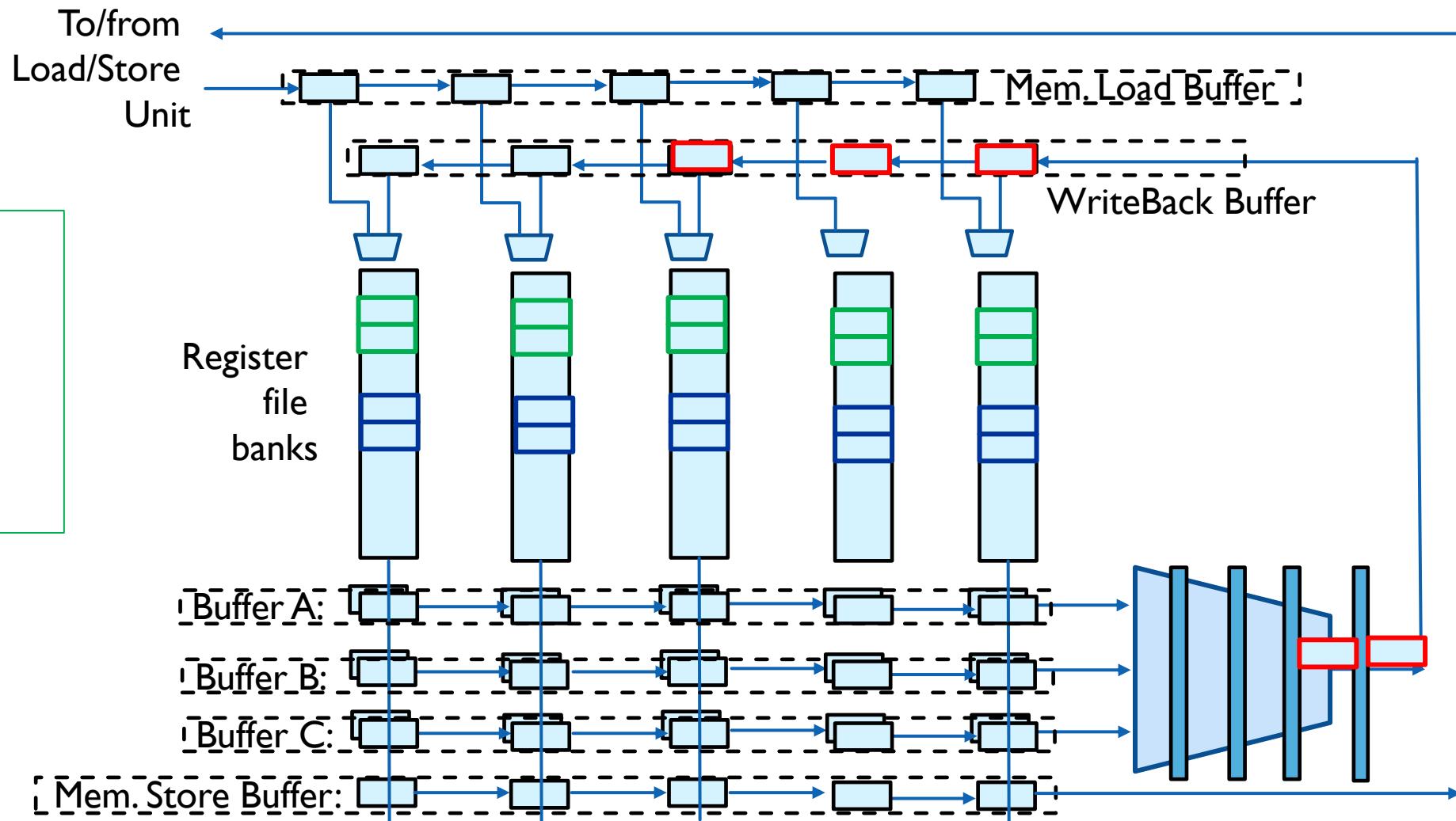
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

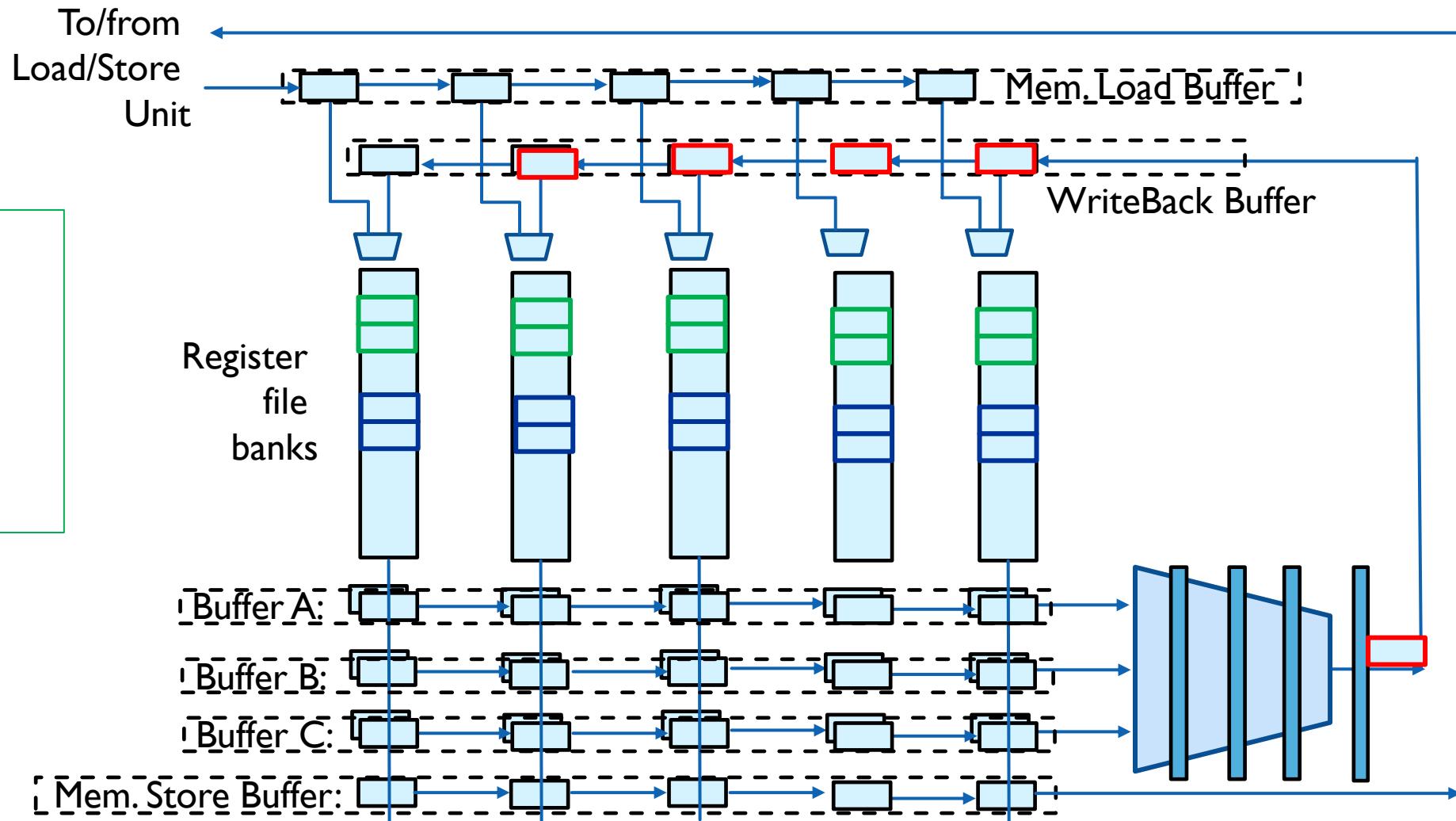
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

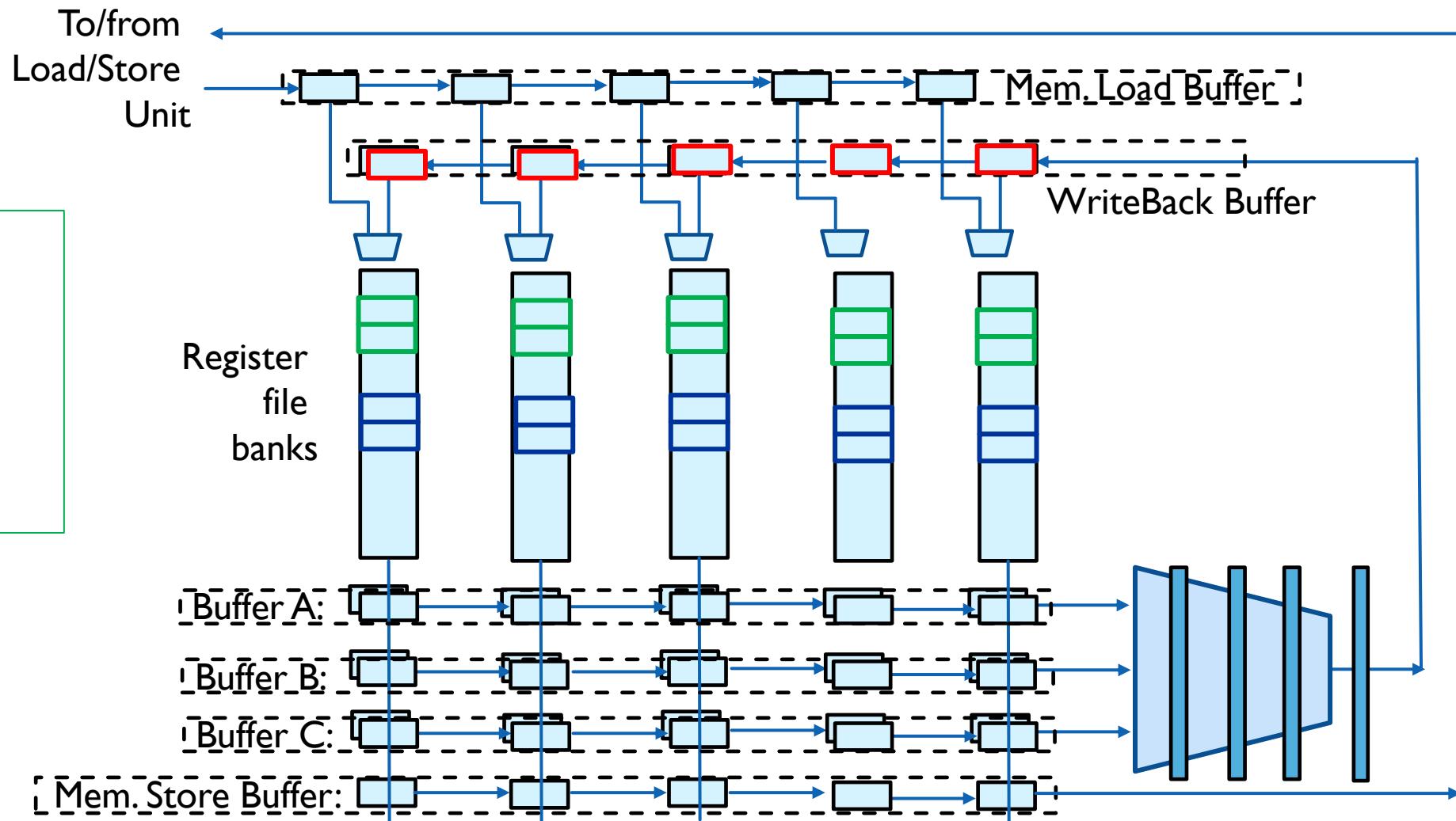
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

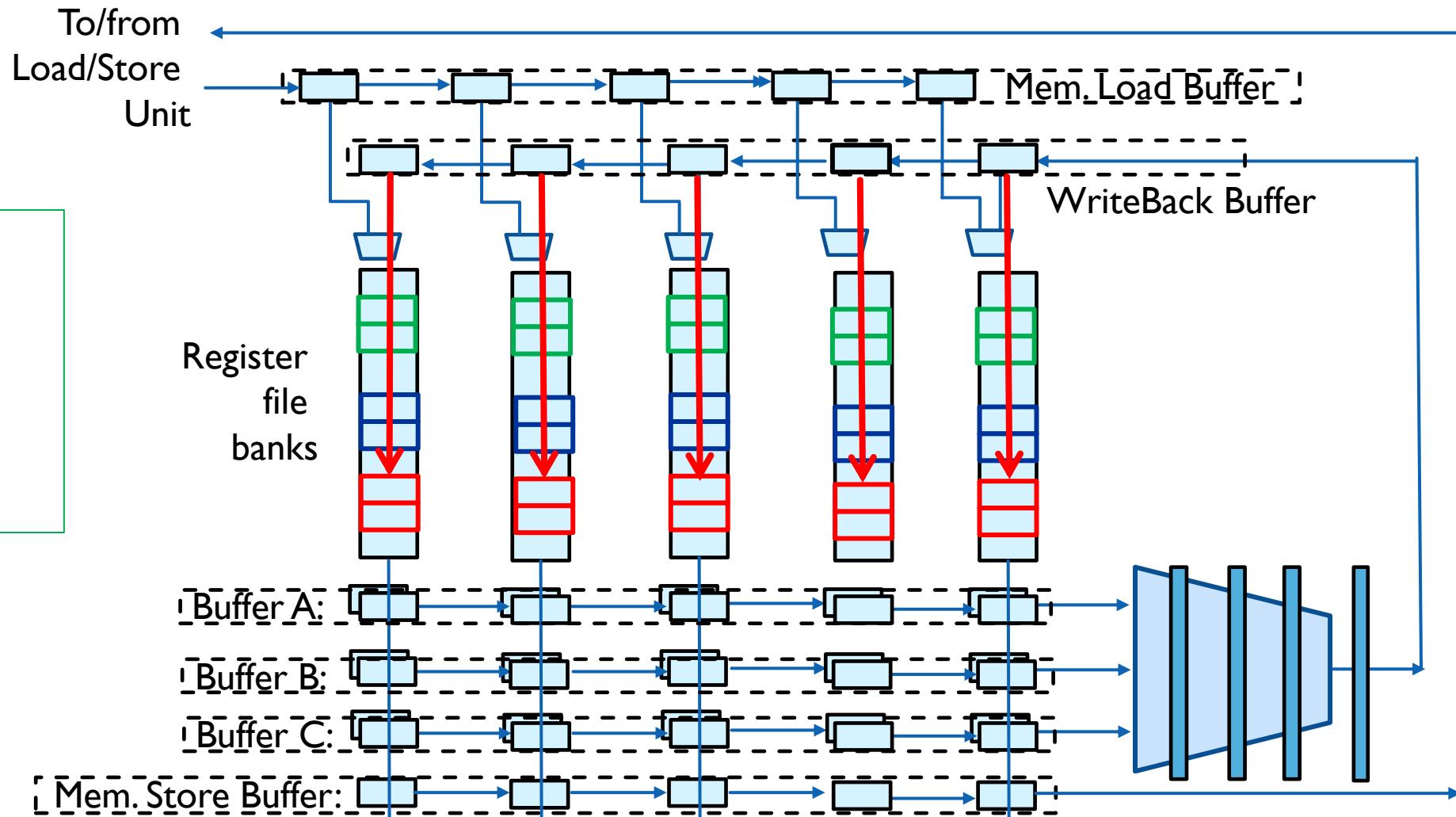
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



VECTOR ARITHMETIC OPERATION EXECUTION

vfadd.vv **v0**, **v1**, **v2**

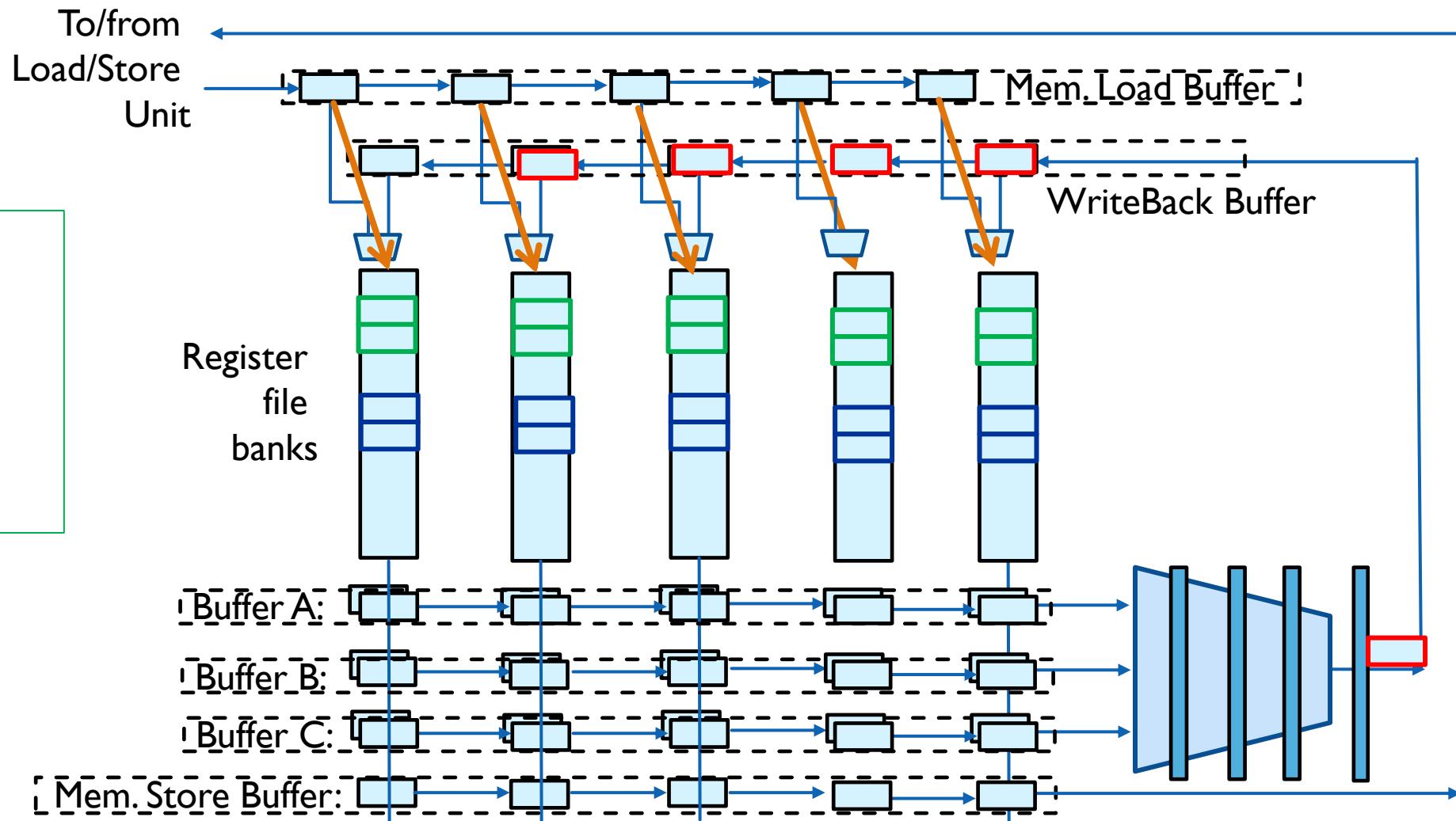
```
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



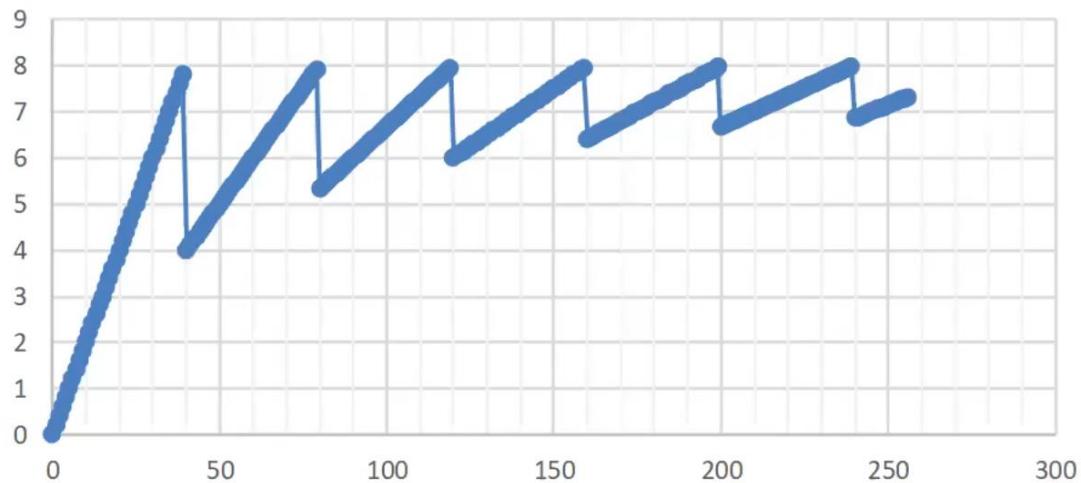
INSERTION OF LOAD OPERATION EXECUTION

```
vfadd.vv v0, v1, v2

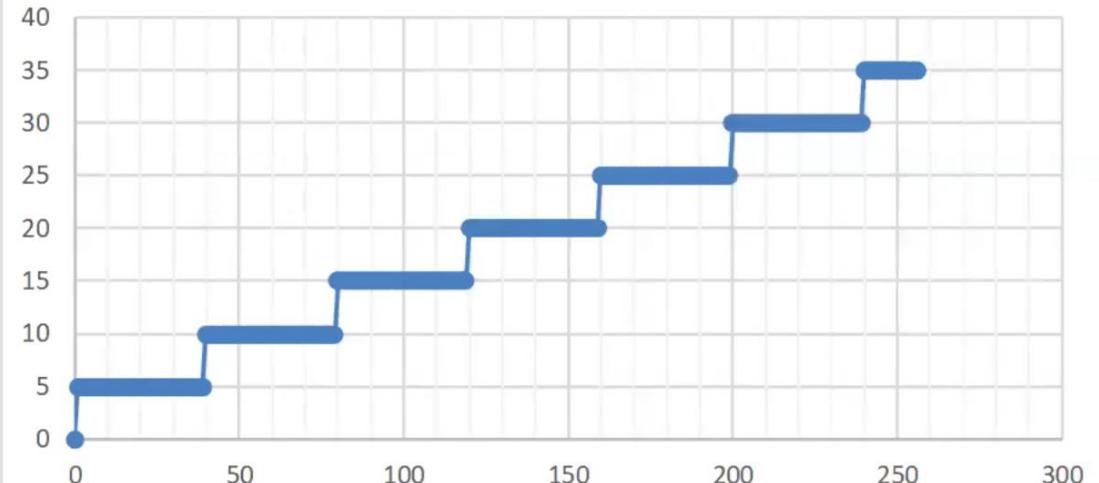
for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL ... VLMAX] = 0;
```



element-wise operations per cycle



instruction execution latency



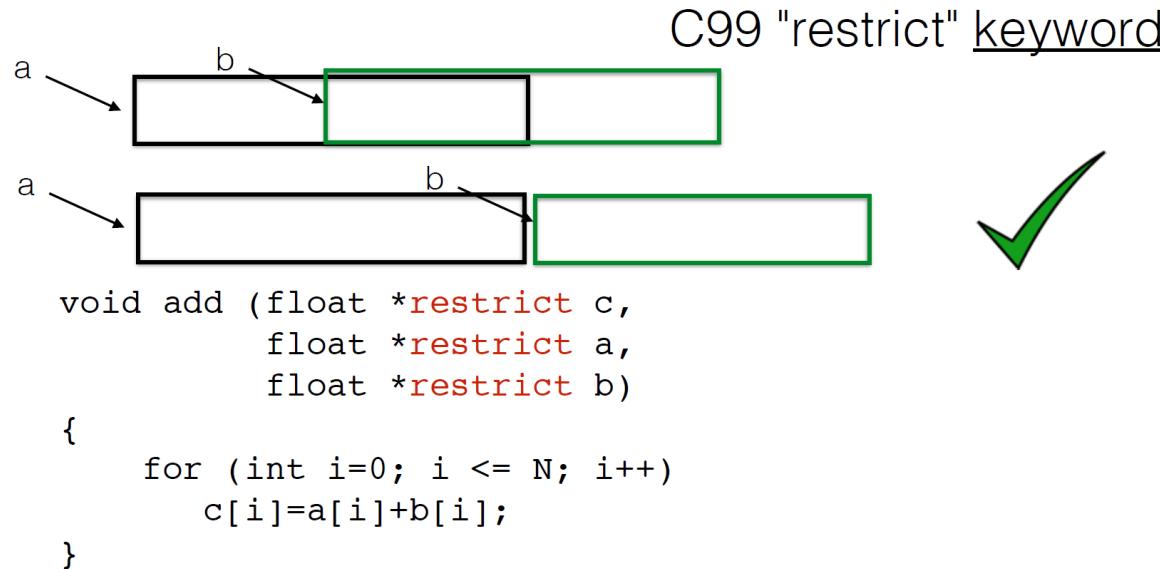
Vectorization:

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



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

```
void add (float *c, float *a, float *b)
{
    Option 1: #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)

```
#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

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

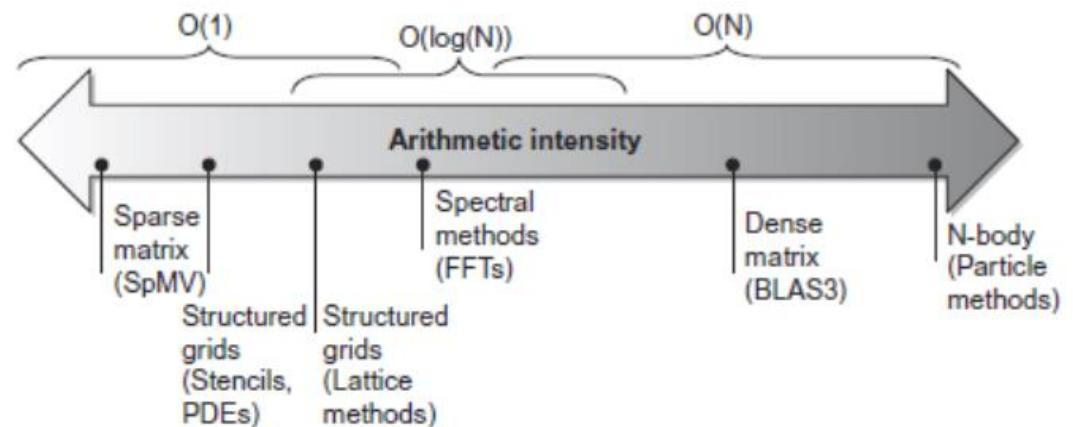
Example SIMD Code

- 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

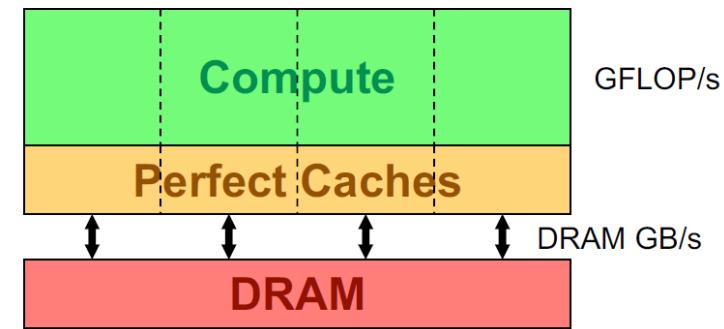
- Basic idea:
 - Plot peak floating-point throughput as a function of arithmetic intensity
 - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
 - Floating-point operations per byte read



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

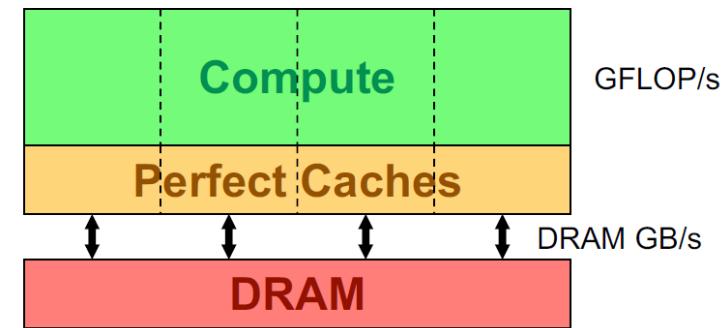
$$\text{Time} = \max \begin{cases} \# \text{FLOPs} / \text{Peak GFLOP/s} \\ \# \text{Bytes} / \text{Peak GB/s} \end{cases}$$



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

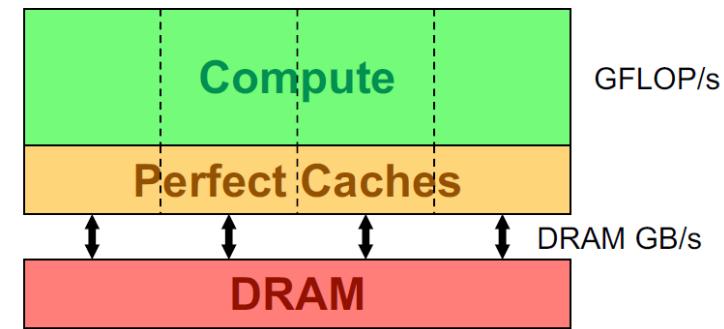
$$\frac{\text{Time}}{\#\text{FLOPs}} = \max \left\{ \begin{array}{l} 1 / \text{Peak GFLOP/s} \\ \#\text{Bytes} / \#\text{FLOPs} / \text{Peak GB/s} \end{array} \right.$$



Roofline (DRAM)

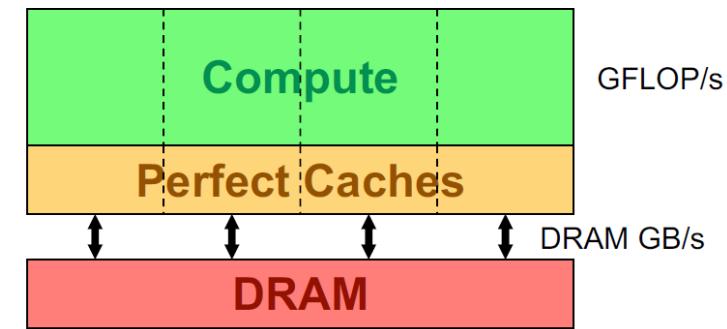
- Any given loop nest will perform:
 - Computation (e.g. FLOPs)
 - Communication (e.g. moving data to/from DRAM)
- With perfect overlap of communication and computation...
 - Run time is determined by whichever is greater

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



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



$$\text{GFLOP/s} = \min \left\{ \begin{array}{l} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{array} \right\}$$

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

Aritmetic Intensity

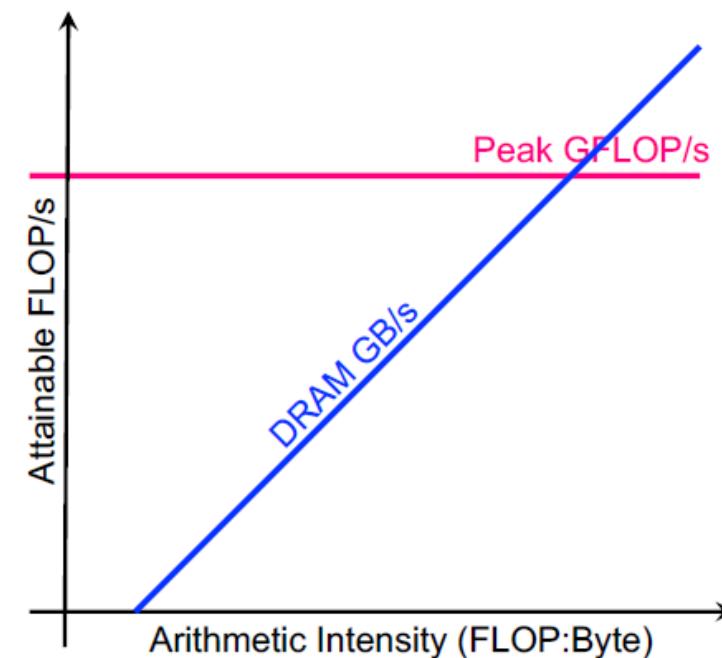
- Measure of data locality (data reuse)
- Ratio of **Total Flops** performed to **Total Bytes** moved
- For the DRAM Roofline...
 - Total Bytes to/from DRAM
 - Includes all cache and prefetcher effects
 - Can be very different from total loads/stores (bytes requested)
 - Equal to ratio of sustained GFLOP/s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

Roofline Model

$$\text{GFLOP/s} = \min \left\{ \begin{array}{l} \text{Peak GFLOP/s} \\ \text{AI * Peak GB/s} \end{array} \right.$$

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...

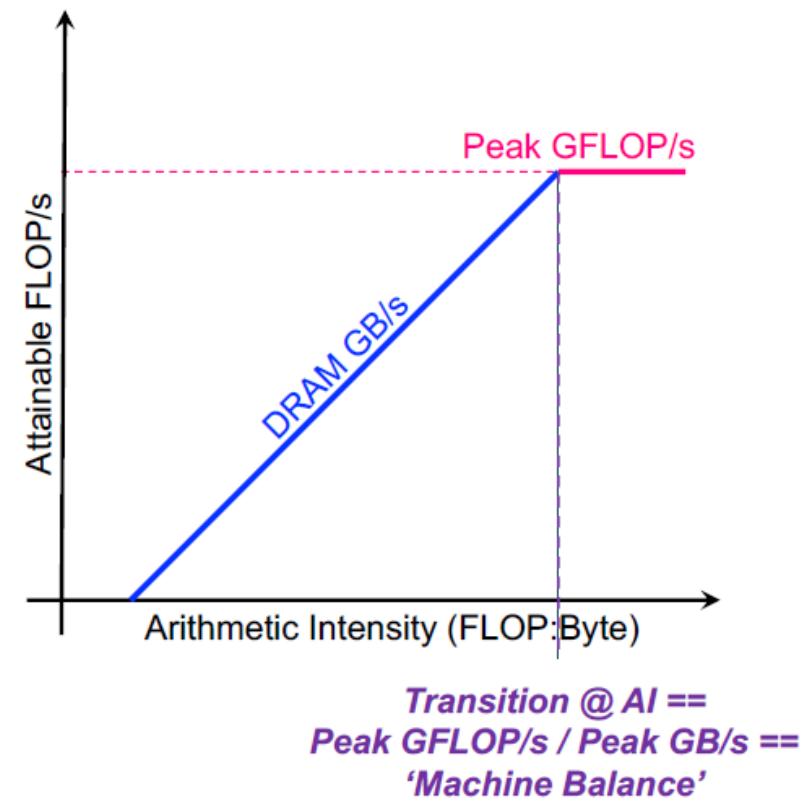


Roofline Model

$$\text{GFLOP/s} = \min \left\{ \begin{array}{l} \text{Peak GFLOP/s} \\ \text{AI * Peak GB/s} \end{array} \right\}$$

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 Examples #1

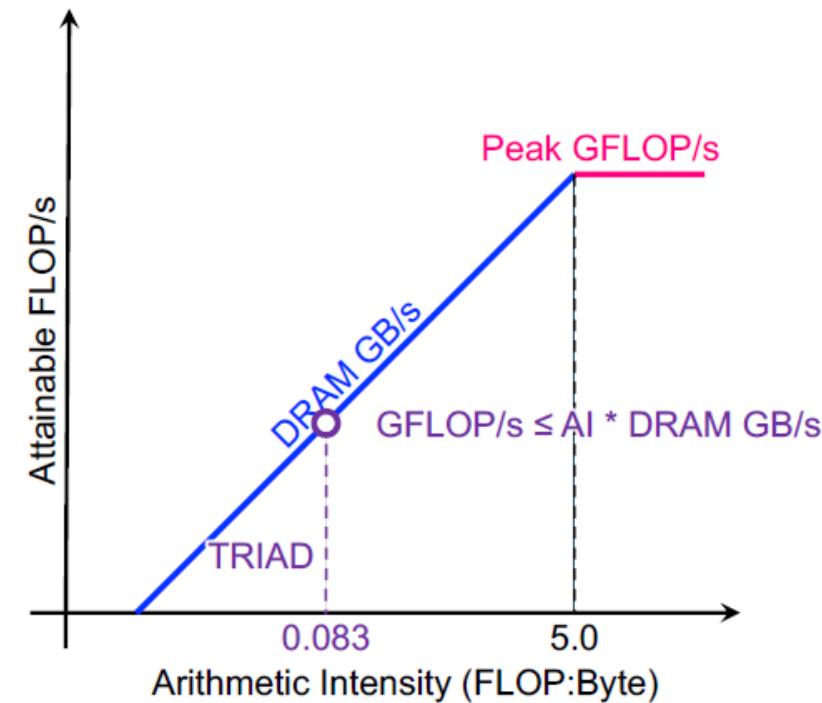
- Typical machine balance is 5-10 FLOPs per byte...

- 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];
}
```

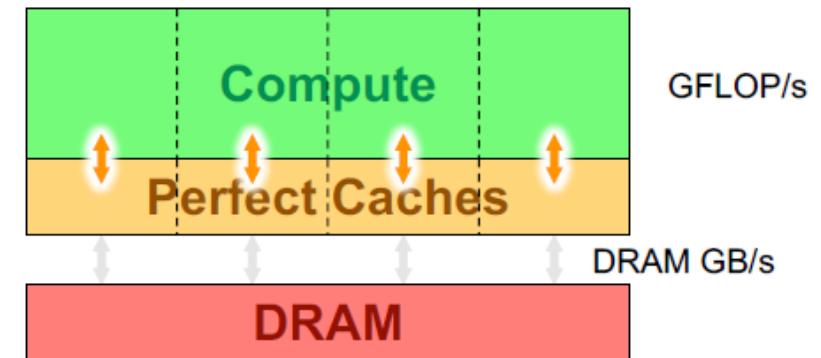
- 2 FLOPs per iteration
 - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
 - **AI = 0.083 FLOPs per byte == Memory bound**



Roofline Examples #2

- Conversely, 7-point constant coefficient stencil...
 - 7 FLOPs
 - 8 memory references (7 reads, 1 store) per point
 - $AI = 7 / (8 \cdot 8) = 0.11 \text{ FLOPs per byte}$
(measured at the L1)

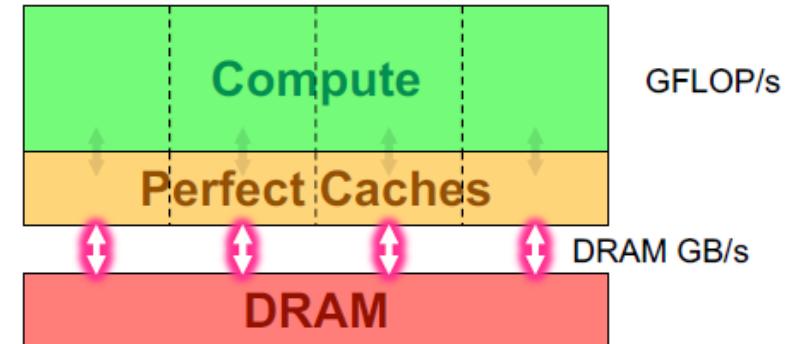
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
    for(j=1;j<dim+1;j++){
        for(i=1;i<dim+1;i++){
            new[k][j][i] = -6.0*old[k][j][i]
                           + old[k][j][i-1]
                           + old[k][j][i][i+1]
                           + old[k][j-1][i]
             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

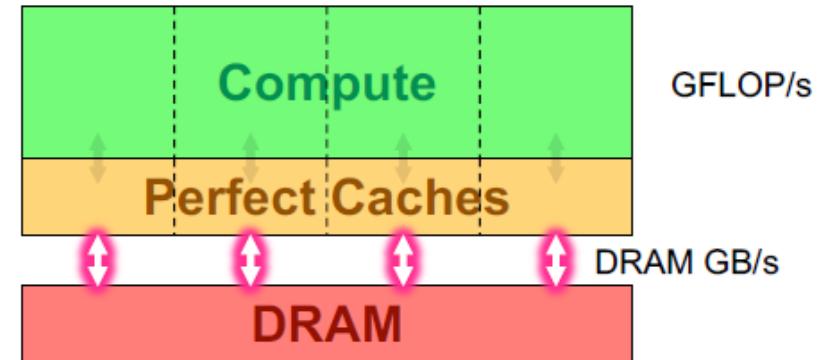
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
    for(j=1;j<dim+1;j++){
        for(i=1;i<dim+1;i++){
            new[k][j][i] = -6.0*old[k][j][i]
                + old[k][j][i-1]
                + old[k][j][i+1]
                + old[k][j-1][i]
                + old[k][j+1][i]
                +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

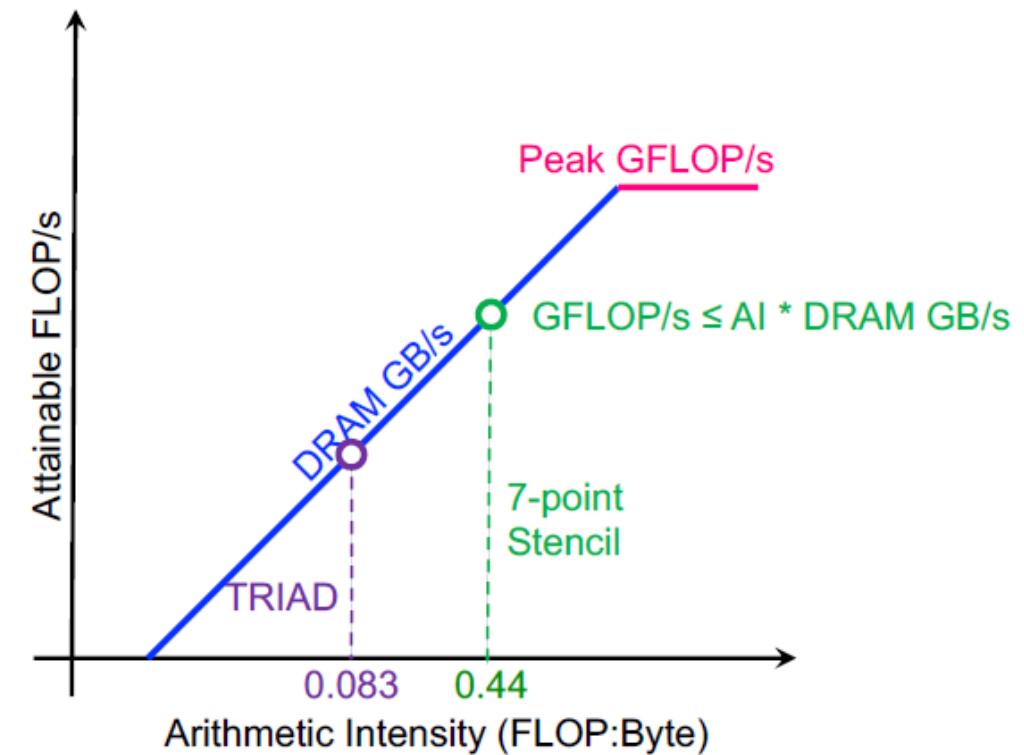


Roofline Examples #2

- Conversely, 7-point constant coefficient stencil...

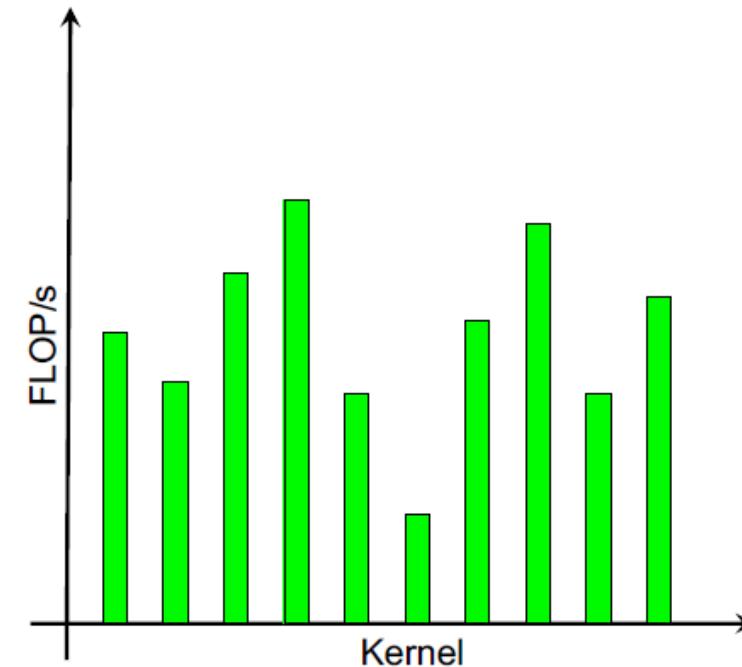
- 7 FLOPs
- 8 memory references (7 reads, 1 store) per point
- Ideally, cache will filter all but 1 read and 1 write per point
- $7 / (8+1) = 0.44 \text{ FLOPs per byte (DRAM)}$
- == memory bound, but 5x the FLOP rate as TRIAD**

```
#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];
        }
    }
}
```



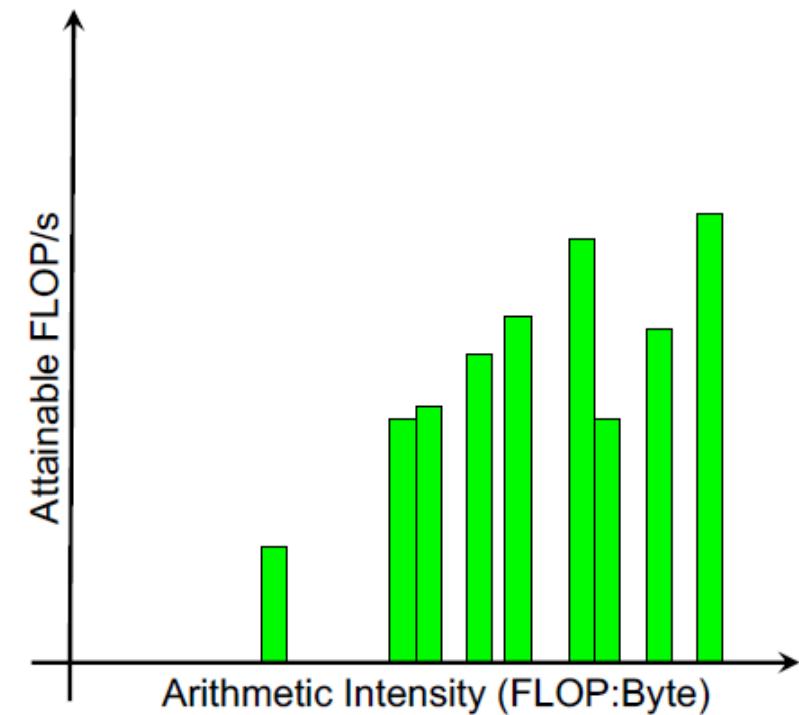
What is “Good” Performance?

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



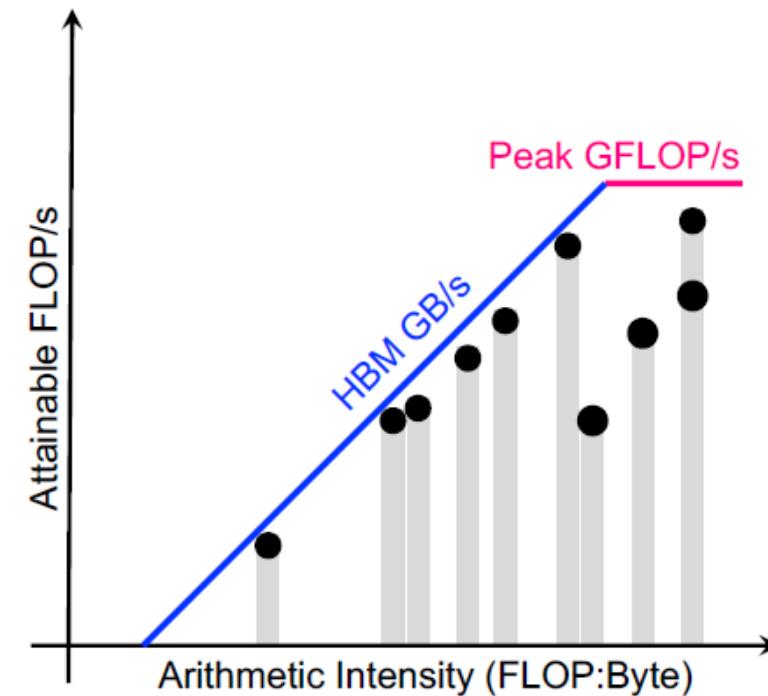
What is “Good” Performance?

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



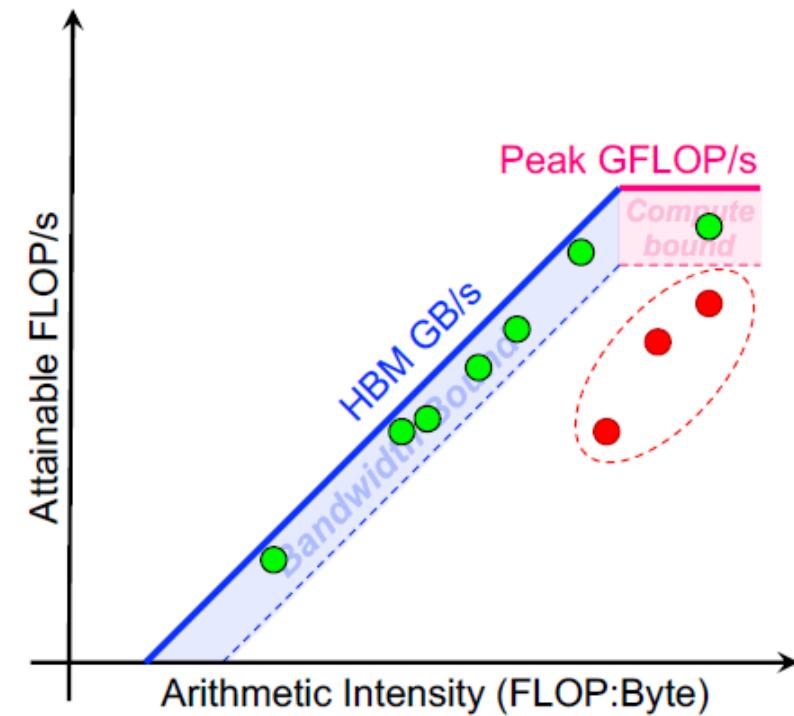
What is “Good” Performance?

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



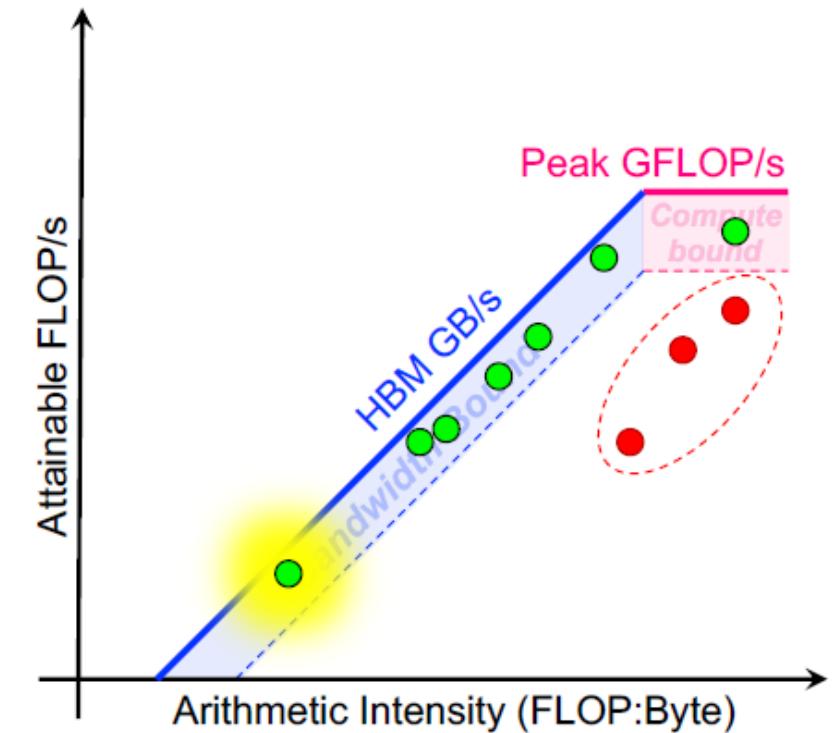
What is “Good” Performance?

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



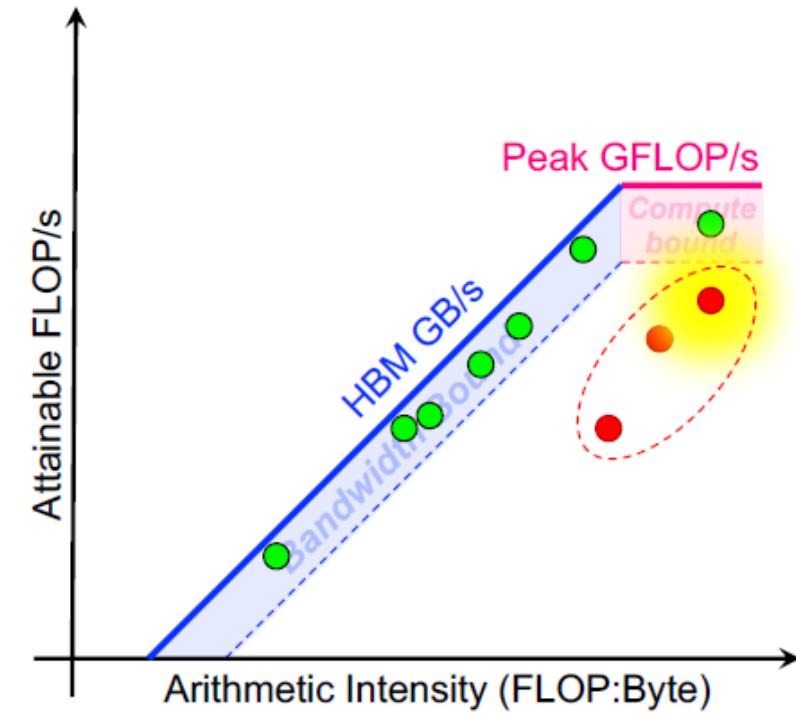
What is “Good” Performance?

- 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.)

