Vector Αρχιτεκτονικές, SIMD Extensions & GPUs

Πηγές/Βιβλιογραφία

- "Computer Architecture: A Quantitative Approach", J. L. Hennessy, D. A. Patterson, Morgan Kaufmann Publishers, INC. 6th Edition, 2017
- Krste Asanovic, "Vectors & GPUs", CS 152 Computer Architecture and Engineering, EECS Berkeley, 2018
  - https://inst.eecs.berkeley.edu/~cs152/sp18/lectures/L15-Vectors.pdf
  - https://inst.eecs.berkeley.edu/~cs152/sp18/lectures/L17-GPU.pdf
- Onur Mutlu, "SIMD Processors & GPUs", Computer Architecture – ETH Zurich, 2017 (slides)
  - https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture8-afterlecture.pdf
  - https://www.youtube.com/watch?v=6DqM1UpTZDM
  - https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture9-afterlecture.pdf
  - https://www.youtube.com/watch?v=mgtlbEqn2dA

Προσεγγίσεις Αύξησης Απόδοσης

- **Παραλληλισμός σε επίπεδο εντολής (Instruction Level Parallelism – ILP)**
  - Εξαρτάται από τις πραγματικές εξαρτήσεις δεδομένων που υφίστανται ανάμεσα στις εντολές.
- **Παραλληλισμός σε επίπεδο νήματος (Thread-Level Parallelism – TLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή, χρησιμοποιώντας πολλαπλά νήματα εκτέλεσης τα οποία είναι εκ κατασκευής παράλληλα.
- **Παραλληλισμός σε επίπεδο δεδομένων (Data-Level Parallelism – DLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή ή δημιουργείται αυτόματα από τον μεταγλωττιστή.
  - Οι ίδιες εντολές επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα

Ταξινόμηση Παράλληλων Αρχιτεκτονικών

- *Single Instruction stream, Single Data stream (SISD)*
  - Uniprocessor.

- *Single Instruction stream, Multiple Data streams (SIMD)*
  - Πολλαπλοί επεξεργαστές, ίδιες εντολές, διαφορετικά δεδομένα (*data-level parallelism*).

- *Multiple Instruction streams, Single Data stream (MISD)*
  - Μέχρι σήμερα δεν έχει εμφανιστεί στην αγορά κάποιο τέτοιο σύστημα (είναι κυρίως για fault tolerance, π.χ. υπολογιστές που ελέγχουν πτήση αεροσκαφών).

- *Multiple Instruction streams, Multiple Data streams (MIMD)*
  - Ο κάθε επεξεργαστής εκτελεί τις δικές του εντολές και επεξεργάζεται τα δικά του δεδομένα. Πολλαπλά παράλληλα νήματα (thread-level parallelism).

[Mike Flynn, “Very high-speed computing systems”. Proc. of IEEE 54, 1966]

Παραλληλισμός σε επίπεδο Δεδομένων

- Προκύπτει από το γεγονός ότι οι ίδιες εντολές επεξεργάζονται πολλαπλά διαφορετικά δεδομένα ταυτόχρονα.
  - Πολλαπλές «επεξεργαστικές» μονάδες

- Παραδείγματα εφαρμογών με σημαντικό παραλληλισμό δεδομένων:
  - Επιστημονικές εφαρμογές (πράξεις με πίνακες)
  - Επεξεργασία εικόνας και ήχου
  - Αλγόριθμοι μηχανικής μάθησης

Παραλληλισμός σε επίπεδο Δεδομένων

- Γιατί Single Instruction Multiple Data (SIMD) αρχιτεκτονική?
- Μια εντολή → πολλαπλά δεδομένα → Καλύτερο energy-efficiency
  - Όταν υπάρχει DLP – εξαρτάται από την εφαρμογή, τον αλγόριθμο, προγραμματιστή, τον μεταγλωττιστή
  - Mobile devices
- Ο προγραμματιστής συνεχίζει να σκέφτεται (περίπου) ακολουθιακά

DLP Αρχιτεκτονικές

1. **Vector**

2. SIMD Extensions

3. GPUs

Vector Architecture

- Πρόκεται για αρχιτεκτονική που υποστηρίζει στο υλικό την εκτέλεση διανυσμάτων ή αλλιώς vectors
- Ένας vector (διάνυσμα) είναι ένας μονοδιάστατος πίνακας αριθμών
  - Εμφανίζονται σε πολλές εφαρμογές

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

- Ιστορικά
  - Εμφανίστηκαν στην δεκαετία του 1970
  - Κυριάρχησαν στον χώρο του supercomputing (1970-1990)
  - Σημαντικά πλεονεκτήματα για συγκεκριμένες εφαρμογές – επανέρχονται δυναμικά

Vector Architecture – Γενική Ιδέα

1. Φέρνει από την μνήμη ένα σύνολο δεδομένων και τα αποθηκεύει σε μεγάλα register files →Vector registers
2. Εκτελεί μία πράξη στο σύνολο όλων αυτών των δεδομένων
3. Γράφει τα αποτελέσματα πίσω στην μνήμη

- Οι καταχωρητές ελέγχονται από τον compiler και χρησιμοποιούντια για να:
  - Κρύψουν τον χρόνο πρόσβασης στην μνήμη
  - Εκμεταλλευτούν το memory bandwidth

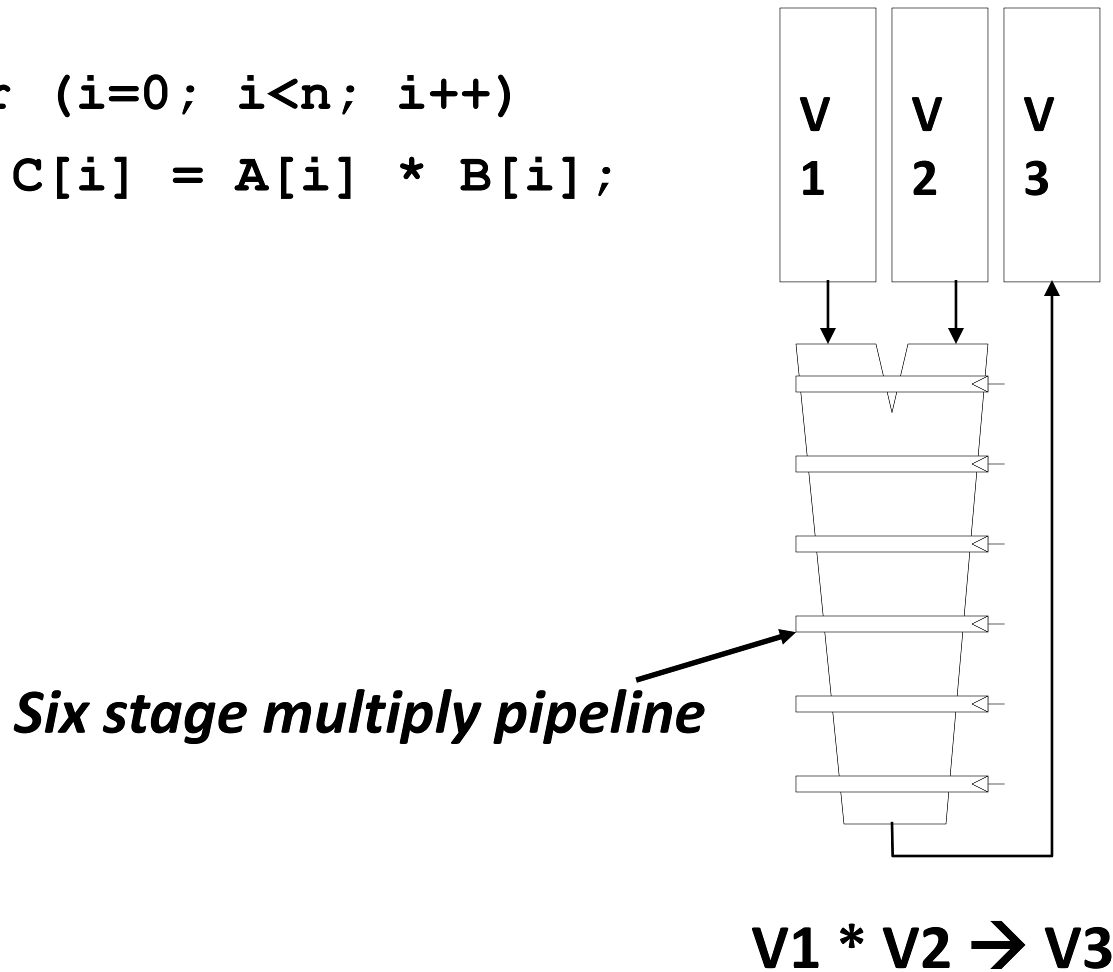
Vector Instructions & Architecture Support

Vector Architecture – Functionality

- Μια εντολή vector πραγματοποιεί μία πράξη σε κάθε στοιχείο του vector σε διαδοχικούς κύκλους
  - Pipelined functional units
  - Κάθε στάδιο του pipeline επεξεργάζεται ένα διαφορετικό στοιχείο

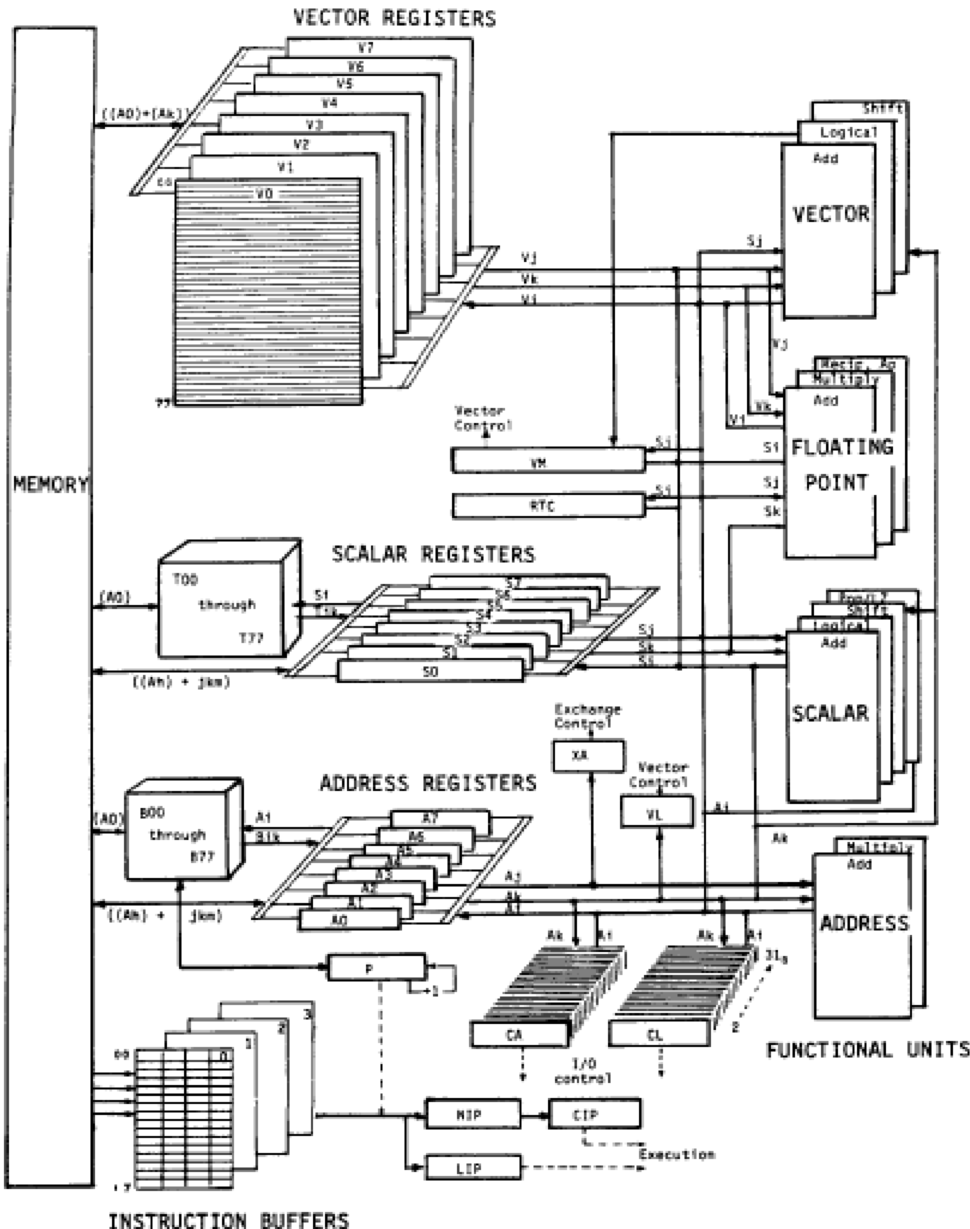
- Οι vector εντολές επιτρέπουν την υλοποίηση pipelines με περισσότερα στάδια (deep pipelines)
  - Δεν υπάρχουν εξαρτήσεις μέσα στους vectors
  - Δεν υπάρχει έλεγχος ροής μέσα στους vectors
  - Η γνώση των strides σχετικά με τις προσβάσεις στην μνήμη επιτρέπει αποτελεσματικό prefetching

Vector Architecture – Functionality



V1 * V2 → V3

Vector Architecture – Cray-1 (1978)



- Scalar Unit
  - Load/Store Architecture
- Vector Extensions
  - Vector Registers
  - Vector Instructions
- Highly pipelined functional units
- Interleaved memory system
  - Memory banks
- No data caches
- No virtual memory

Vector Architecture – Πλεονεκτήματα

- Δεν υπάρχουν εξαρτήσεις μέσα στους vectors
  - Η pipeline τεχνική δουλεύει πολύ καλά
  - Επιτρέπει την ύπαρξη deep pipelines
- Κάθε εντολή αντιπροσωπεύει πολλή δουλειά
  - Απλούστερη λογική για instruction fetch και ανάγκη για λιγότερο memory bandwidth λόγω instructions
- Οι προσβάσεις στην μνήμη γίνονται με regular patterns
- Λιγότερα branch instructions

Vector Architecture – Μειονεκτήματα

- Δουλεύει καλά μόνο όταν υπάρχει διαθέσιμος παραλληλισμός δεδομένων στην εφαρμογή

- Μπορεί να δημιουργηθεί πρόβλημα λόγω memory bandwidth όταν:
  - Ο λόγος των εντολών υπολογισμού προς τις εντολές μνήμης δεν καλύπτει το κόστος της ανάγνωσης/εγγραφής των vectors από την μνήμη
  - Τα δεδομένα δεν είναι κατάλληλα αποθηκευμένα στην μνήμη (memory banks)

Vector Architecture – VMIPS

- RV64V (RISC-V base instructions + vector extensions)
  - Αρχιτεκτονική βασισμένη στον Cray-1
- Vector data registers
  - 32 registers, 64-bits wide each element
  - 16 read ports & 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazard detection
- Vector load/store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general purpose registers + 32 floating point registers

Vector Architecture – Registers

- Vector data registers
- Vector control registers
  - VLEN, VMASK, VSTR
- VLEN (Vector Length Register)
  - Δηλώνει το μέγεθος του vector
  - Η μέγιστη τιμή του καθορίζεται από την αρχιτεκτονική
    - MVL – Maximum Vector Length
- VMASK (Vector Mask Register)
  - Δηλώνει τα στοιχεία του vector στα οποία εφαρμόζονται οι vector instructions
- VSTR (Vector Stride Register)
  - Δηλώνει την απόσταση των στοιχείων στην μνήμη για την δημιουργία ενός vector

DAXPY Παράδειγμα – Scalar Code

```
double a, X[N], Y[N];
for (i=0; i<32; i++) {
      Y[i] = a*X[i] + Y[i];
}
```

```
        fld      f0,a            # Load scalar a
        addi     x28,x5,#256     # Last address to load
Loop:
        fld      f1,0(x5)        # load X[i]
        fmul.d   f1,f1,f0        # a * X[i]
        fld      f2,0(x6)        # load Y[i]
        fadd.d   f2,f2,f1        # a * X[i] + Y[i]
        fsd      f2,0(x6)        # store into Y[i]
        addi     x5,x5,#8        # Increment index to X
        addi     x6,x6,#8        # Increment index to Y
        bne      x28,x5,Loop     # check if done
```

DAXPY Παράδειγμα – Vector Code

```
double a, X[N], Y[N];
for (i=0; i<32; i++) {
    Y[i] = a*X[i] + Y[i];
}
```

```
vsetdcfg 4*FP64       # Enable 4 DP FP vregs
fld      f0,a         # Load scalar a
vld      v0,x5        # Load vector X
vmul     v1,v0,f0     # Vector-scalar mult
vld      v2,x6        # Load vector Y
vadd     v3,v1,v2     # Vector-vector add
vst      v3,x6        # Store the sum
vdisable              # Disable vector regs
```

8 instructions for RV64V (vector code)

vs.

258 instructions for RV64G (scalar code)

Vector Execution Time

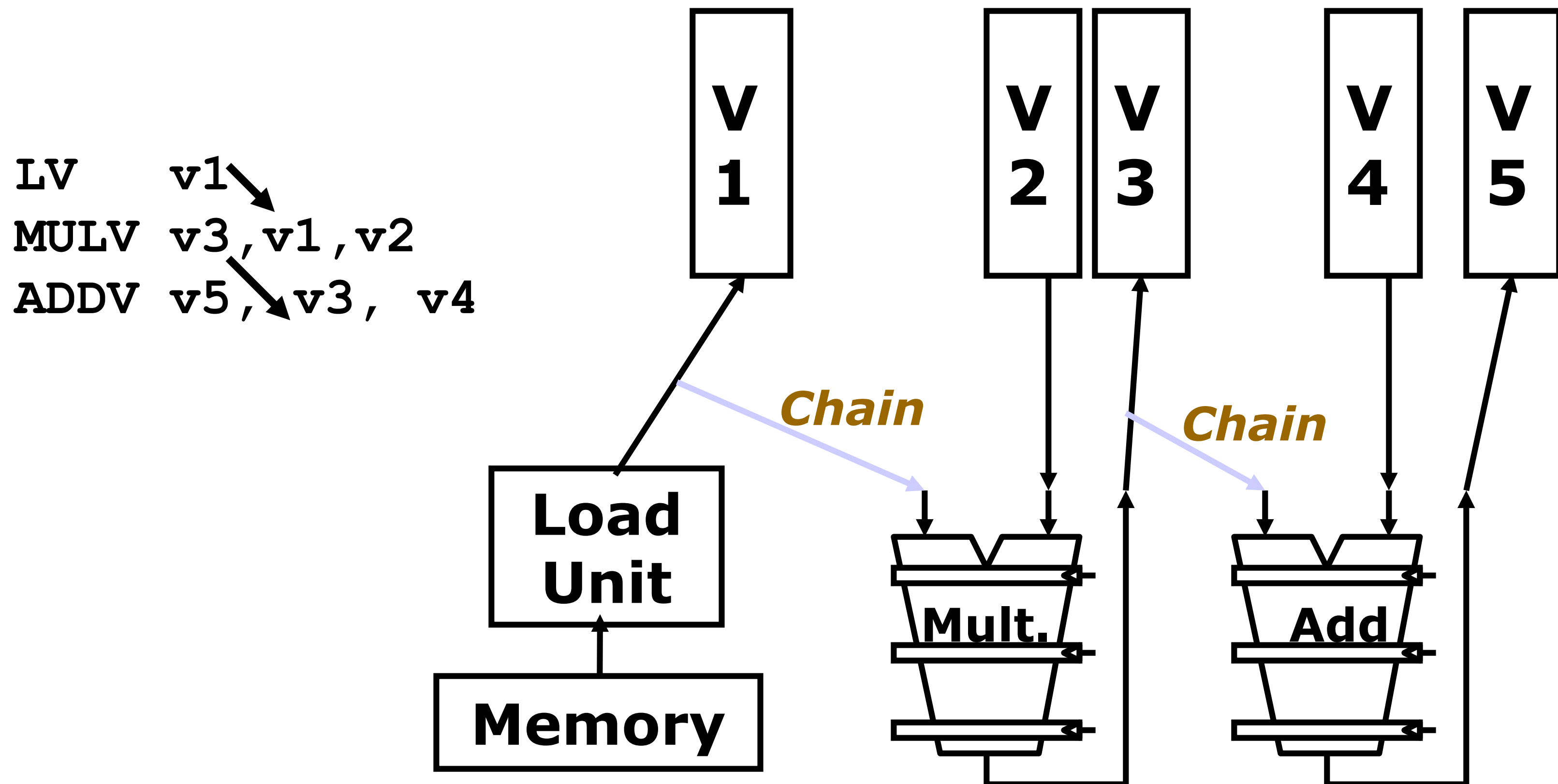
- Η επίδοση εξαρτάται από τρεις παράγοντες:
  - Μέγεθος των vectors
  - Δομικοί κίνδυνοι (structural hazards)
  - Εξαρτήσεις δεδομένων (data dependencies)

- Lanes
  - Πολλαπλά παράλληλα pipelines που παράγουν δύο ή περισσότερα αποτελέσματα σε κάθε κύκλο

- Convoy
  - Vector instructions που θα μπορούσαν να εκτελεστούν ταυτόχρονα
    - Χωρίς δομικούς κινδύνους

Vector Chaining & Chimes

- Ακολουθίες από read-after-write εξαρτήσεις δεδομένων τοποθετούνται στο ίδιο convoy και εκτελούνται μέσω της τεχνικής του ***chaining***



Vector Chaining & Chimes

- Chaining
  - Μια vector εντολή ξεκινάει να εκτελείται καθώς στοιχεία του vector source operand γίνονται διαθέσιμα

- Chime
  - Μονάδα χρόνου για την εκτέλεση ενός convoy
  - *m* convoys εκτελούνται σε *m chimes* για vector length *n*
  - Χρειάζεται *(m* x *n)* κύκλους για vector length *n*

Vector Chaining & Chimes – Παράδειγμα

```
vld     v0,x5      # Load vector X
vmul    v1,v0,f0   # Vector-scalar multiply
vld     v2,x6      # Load vector Y
vadd    v3,v1,v2   # Vector-vector add
vst     v3,x6      # Store the sum
```

Convoys:

1st chime:	vld	vmul
2nd chime:	vld	vadd
3rd chime:	vst	

- 3 chimes, 2 FP ops per result, cycles per FLOP = 1.5
- For 32 element vectors, requires 32 x 3 = 96 clock cycles

Vector Architecture – Challenges

- **Start up time →** Καθυστέρηση μέχρι να γεμίσει το pipeline
- **Execute more than vector elements per cycle?**
  - Multiple lanes execution
- **Vector length != Maximum Vector Length (MVL)?**
  - Vector Length Register + Strip mining
- **If statements + vector operations?**
  - Vector Mask Register + Predication
- **Memory system?**
  - Memory banks
- **Multiple dimensional matrices?**
  - Vector Stride Register
- **Sparse matrices?**
  - Scatter/gather operations
- **Programming a vector computer?**

Multiple Lanes Execution

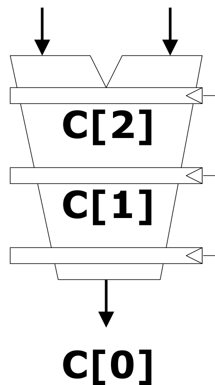
VADD A,B → C

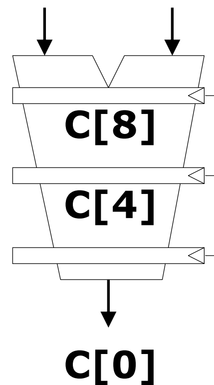
Execution using one pipelined functional unit

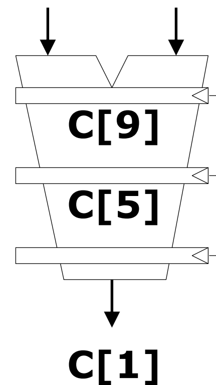
Execution using four pipelined functional units

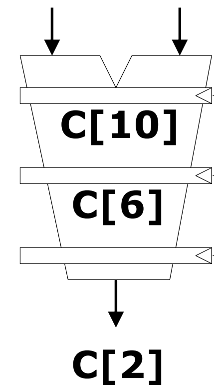
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]

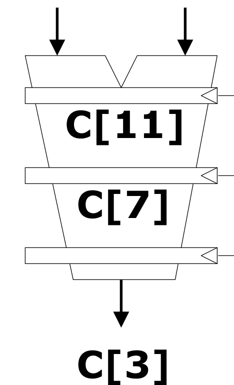
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]





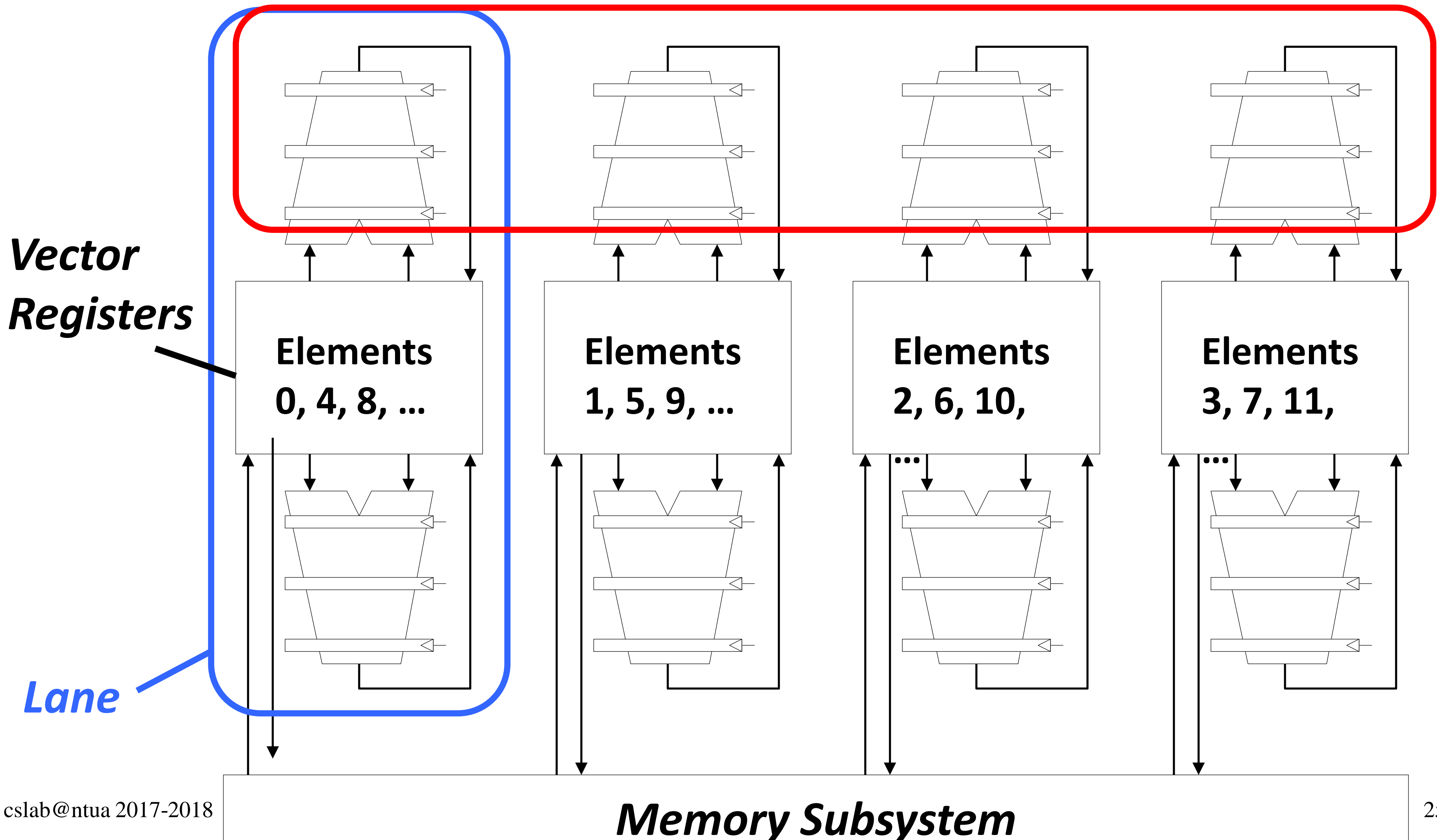






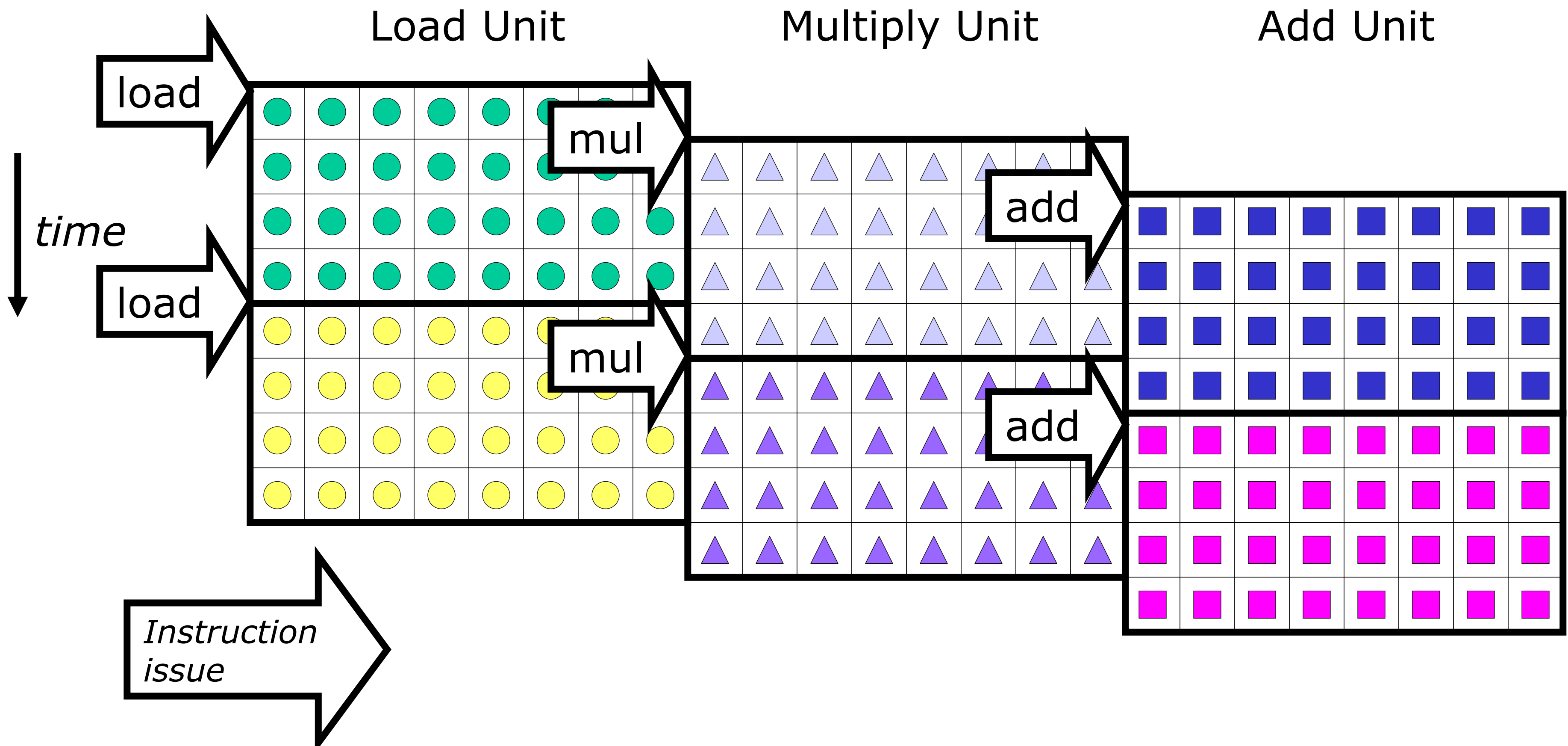
Multiple Lanes Execution

Το στοιχείο *n* του vector register A είναι "hardwired" στο στοιχείο *n* του vector register B → multiple HW lanes



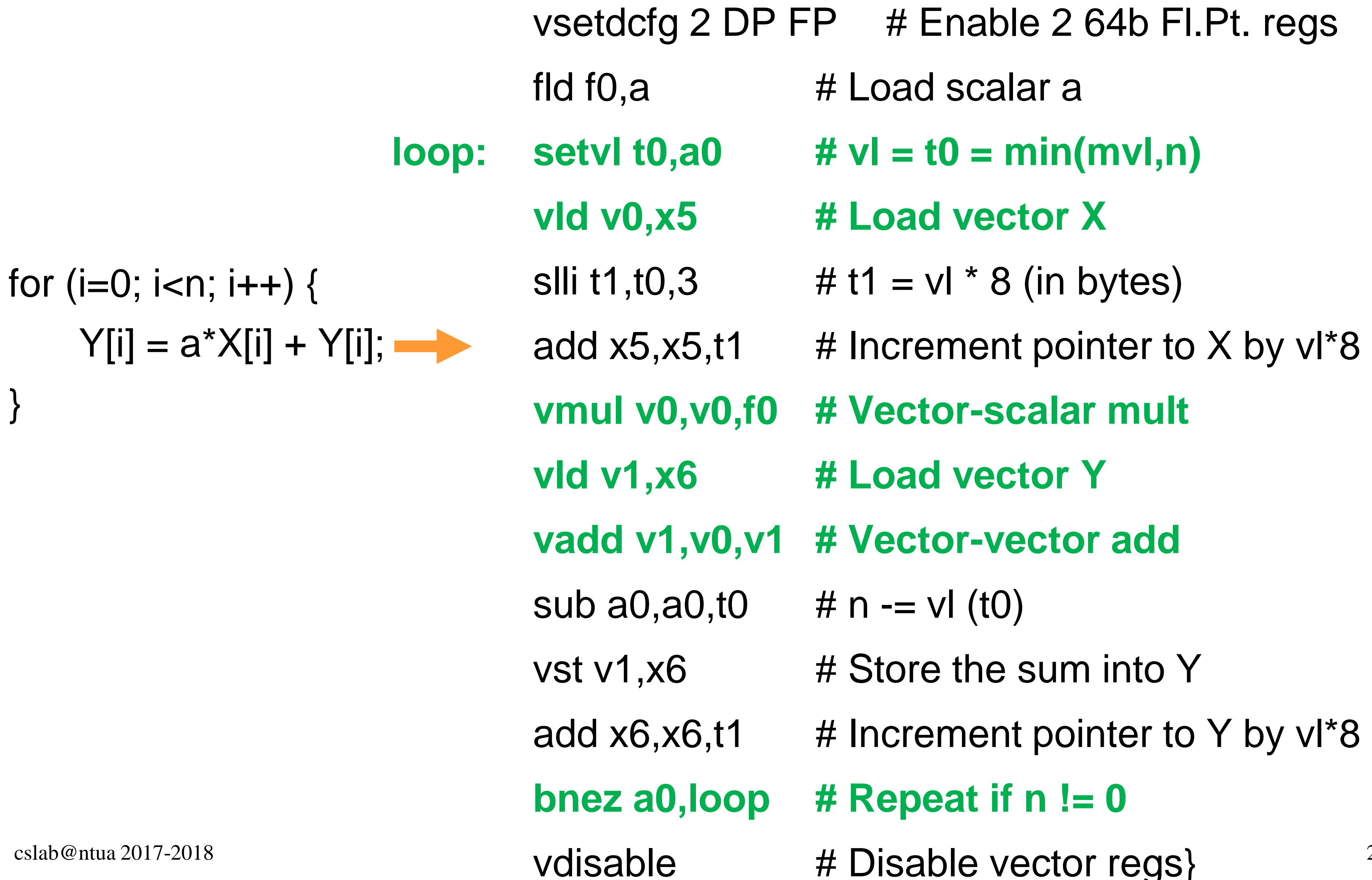
Vector Instruction Parallelism

- 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 Length Register – Strip-Mining Technique



Vector Mask Registers
Disable elements through predication

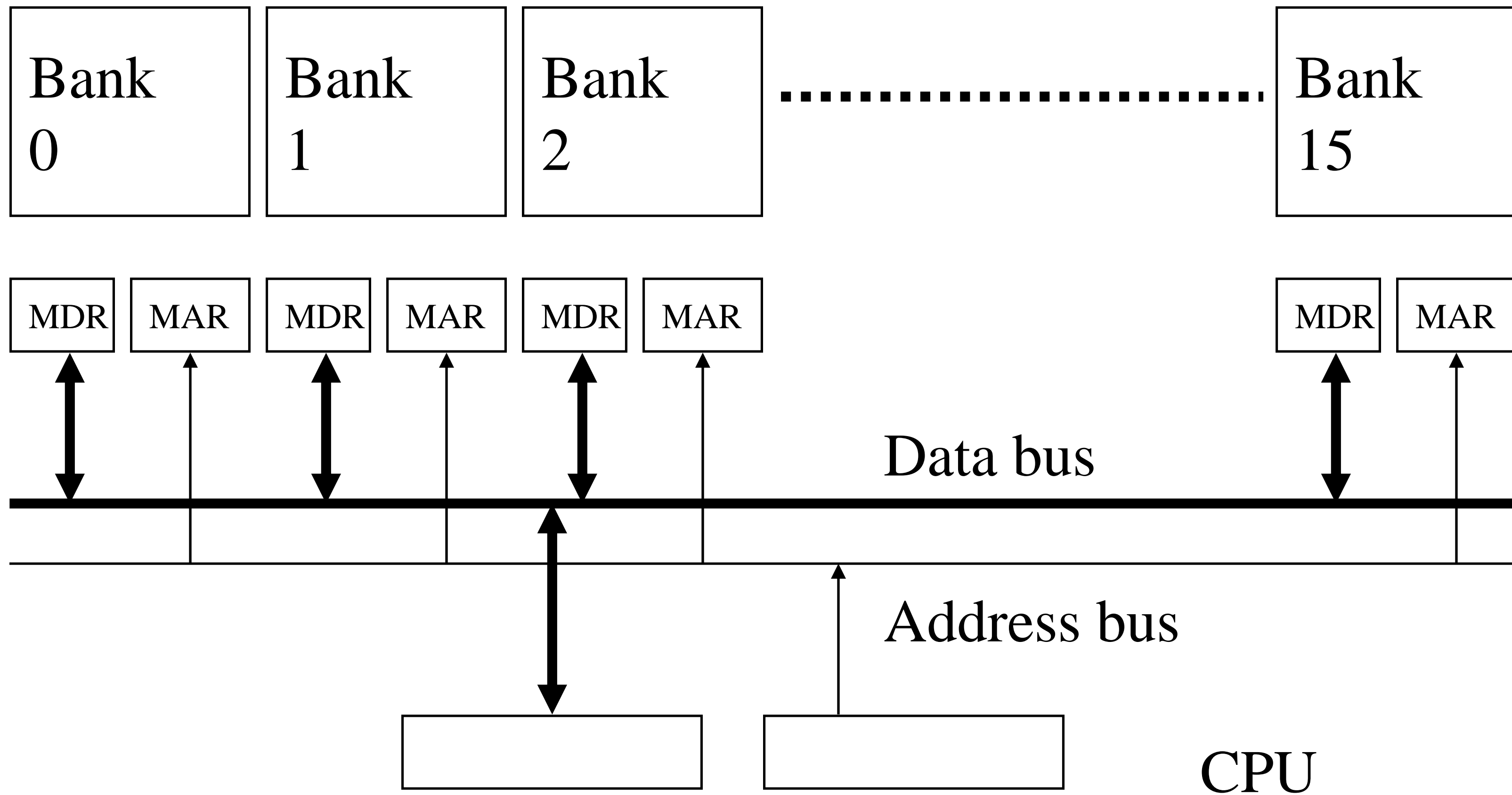
```
for (i = 0; i < 64; i=i+1) {
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
}
```

```
vsetdcfg   2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi  1           # Enable 1 predicate register
vld        v0,x5       # Load vector X into v0
vld        v1,x6       # Load vector Y into v1
fmv.d.x    f0,x0       # Put (FP) zero into f0
vpne       p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub       v0,v0,v1    # Subtract under vector mask
vst        v0,x5       # Store the result in X
vdisable               # Disable vector registers
vpdisable              # Disable predicate registers
```

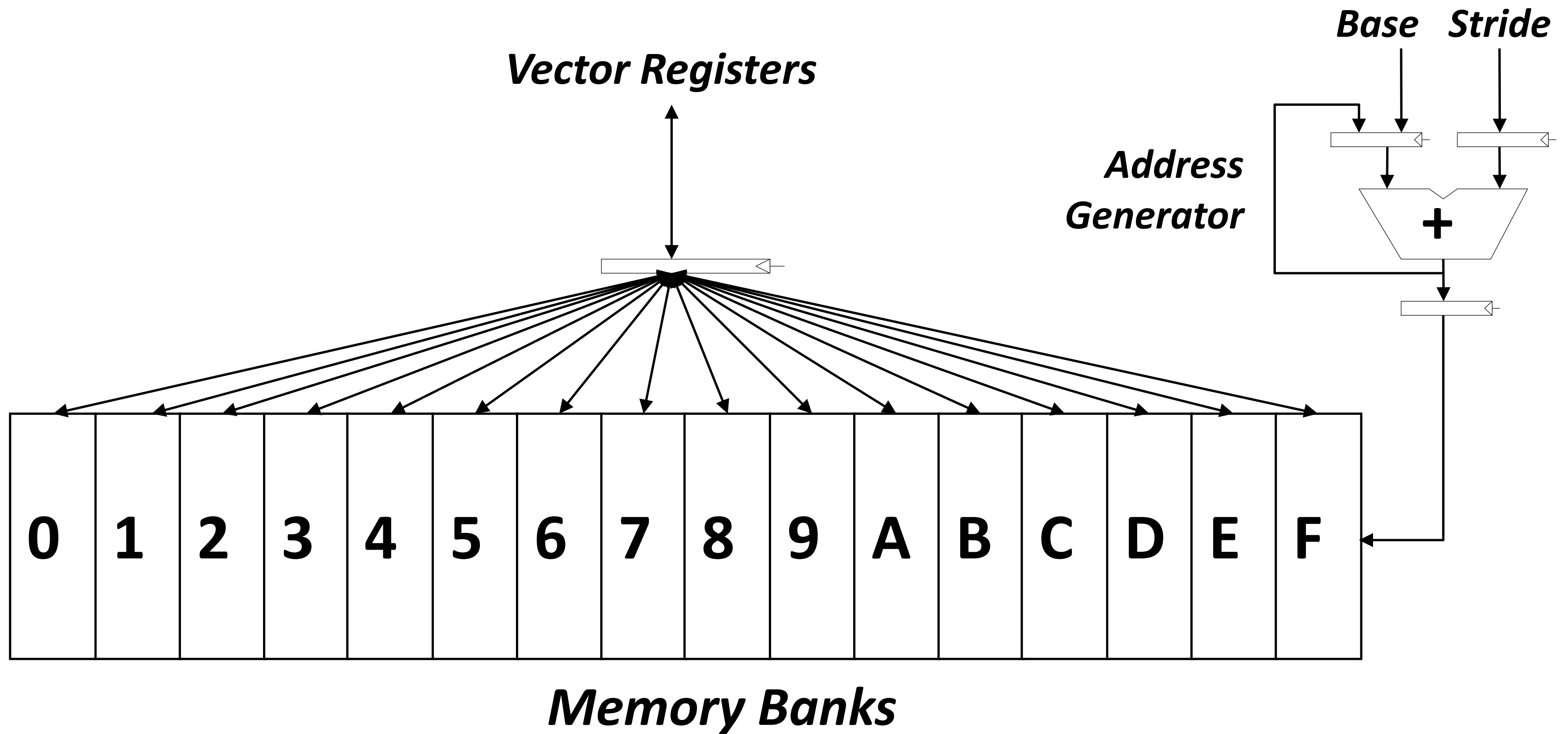
Memory Banks

- Το σύστημα μνήμης πρέπει να υποστηρίζει υψηλό bandwidth για vector loads & stores
- Προσέγγιση: Διαμοιρασμός των προσβάσεων μνήμης σε πολλαπλά banks
- Η μνήμη χωρίζεται σε banks τα οποία προσπελαύνονται ανεξάρτητα
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Μπορεί να εξυπηρετήσει N παράλληλες προσπελάσεις αν όλες πηγαίνουν σε διαφορετικά banks

Memory Banks



Memory Banks



Stride Accesses

- Παράδειγμα:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Vector Stride Register
  - Stride: απόσταση μεταξύ δύο στοιχείων για τον σχηματισμό vector
  - Ίσως προκύψουν bank conflicts

Scatter-Gather

- Παράδειγμα:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Index vector instructions

```
vsetdcfg   4*FP64        # 4 64b FP vector registers
vld        v0, x7        # Load K[]
vldx       v1, x5, v0    # Load A[K[]]
vld        v2, x28       # Load M[]
vldx       v3, x6, v2    # Load C[M[]]
vadd       v1, v1, v3    # Add them
vstx       v1, x5, v0    # Store A[K[]]
vdisable                 # Disable vector registers
```

Programming a Vector computer

- Ο compiler μπορεί να δώσει feedback στον προγραμματιστή
  - Για να εφαρμόσει vector optimizations

- Ο προγραμματιστής μπορεί να δώσει hints στον compiler
  - Ποια κομμάτια κώδικα να γίνουν vectorized

DLP Αρχιτεκτονικές

1. Vector
2. **SIMD Extensions**
3. GPUs

SIMD Extensions

- Πολλές εφαρμογές πολυμέσων επεξεργάζονται δεδομένα που έχουν μέγεθος μικρότερο από το μέγεθος λέξης (π.χ. 32 bits) για το οποίο ο επεξεργαστής είναι βελτιστοποιημένος
- Παραδείγματα:
  - Τα pixels γραφικών αναπαριστώνται συνήθως με 8 bits για κάθε κύριο χρώμα + 8 bits για transparency
  - Τα δείγματα ήχου αναπαριστώνται συνήθως με 8 ή 16 bits
- SIMD Extensions
  - Διαμοιρασμός των functional units για την ταυτόχρονη επεξεργασία στοιχείων μικρών vectors
  - Παράδειγμα: “partitioned” adder

SIMD Extensions vs. Vectors

- SIMD extensions operate on *small vectors*
  - Πολύ μικρότερα register files
  - Λιγότερη ανάγκη για υψηλό memory bandwidth

- Περιορισμοί των SIMD Extensions:
  - Ο αριθμός των data operands αντικατοπτρίζεται στο op-code
    - Το instruction set γίνεται περισσότερο περίπλοκο
  - Δεν υπάρχει Vector Length Register
  - Δεν υποστηρίζονται περίπλοκοι τρόποι διευθυνσιοδότησης (strided, scatter/gather)
  - Δεν υπάρχει Vector Mask Register

SIMD Extensions – Υλοποιήσεις

- Intel MMX (1996)
  - Eight 8-bit integer ops or four 16-bit integer ops
  - Peleg and Weiser, “MMX Technology Extension to the Intel Architecture”, IEEE Micro, 1996
- 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

SIMD Extensions – Παράδειγμα

```
for (i=0; i<n; i++) {
    Y[i] = a*X[i] + Y[i];
}
```

```
       fld        f0,a            # Load scalar a
       splat.4D   f0,f0           # Make 4 copies of a
       addi       x28,x5,#256     # Last addr. 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]
       addi       x5,x5,#32       # Increment index to X
       addi       x6,x6,#32       # Increment index to Y
       bne        x28,x5,Loop     # Check if done
```

DLP Αρχιτεκτονικές

1. Vector
2. SIMD Extensions
3. **GPUs**

Graphical Processing Units

- Επιταχυντές για την επεξεργασία γραφικών
  - Υψηλός παραλληλισμός σε εφαρμογές γραφικών

- GP-GPUs
  - **G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits
  - Χρησιμοποιούνται ευρέως για την επιτάχυνση data και compute intensive εφαρμογών

- Εφαρμογές ιδανικές για GPGPUs
  - Υψηλός παραλληλισμός
  - Υψηλό arithmetic intensity
  - Μεγάλα data sets

Graphical Processing Units

- Ετερογενές μοντέλο εκτέλεσης
  - Η CPU είναι ο *Host*
  - Η GPU είναι το *device*
  - Επικοινωνία μέσω PCIe bus
- Μοντέλο προγραμματισμού
  - C-like γλώσσα προγραμματισμού για GPU
  - Βασίζεται στην έννοια των “threads”
  - **Single Instruction, Multiple threads (SIMT)**
    - Multi-threaded προγραμματιστικό μοντέλο
    - Διαφορετικό από SIMD που χρησιμοποιεί data parallel προγραμματιστικό μοντέλο
    - Συνδυάζει SIMD και Multithreading

GPUs & Flynn’s Taxonomy

- Οι GPUs αποτελούνται από πολλαπλούς επεξεργαστές
  - Κάθε ένας επεξεργαστής αποτελείται από ένα multithreaded SIMD pipeline
- Το pipeline εκτέλεσης των εντολών λειτουργεί σαν ένα SIMD pipeline

- Όμως, ο προγραμματισμός γίνεται με την έννοια των threads
  - Όχι με την έννοια των SIMD εντολών

- Κάθε thread εκτελεί την ίδια εντολή αλλά επεξεργάζεται διαφορετικά δεδομένα
- Κάθε thread έχει το δική του κατάσταση και μπορεί να εκτελεστεί ανεξάρτητα

GPUs & Flynn’s Taxonomy (2)

- SIMD Extensions
  - Το SIMD υλικό αξιοποιείται μέσα από:
    - *data parallel* προγραμματιστικό μοντέλο
      - Ο προγραμματιστής (ή ο μεταγλωττιστής) χρησιμοποιεί τις SIMD εντολές για να εκτελέσει την ίδια εντολή σε πολλά δεδομένα
    - Το οποίο εκτελείται μέσα από ένα SIMD μοντέλο εκτέλεσης
- GPUs
  - Το SIMD υλικό αξιοποιείται μέσα από:
    - Multithreaded προγραμματιστικό μοντέλο
      - Ο προγραμματιστής (ή ο μεταγλωττιστής) παράγει σειριακό κώδικα και δημιουργεί νήματα για να εκτελέσει τον ίδιο κώδικα σε πολλά δεδομένα
    - Το οποίο εκτελείται μέσα από ένα SIMD μοντέλο εκτέλεσης

SIMD vs. SIMT Μοντέλο Εκτέλεσης Υλικού

- Μοντέλο Εκτέλεσης Υλικού
  - Καθορίζει πως εκτελείται ένας κώδικας στο υλικό

- SIMD – Single Instruction Multiple Data
  - Μία ροή από διαδοχικές SIMD εντολές
  - Κάθε SIMD εντολή καθορίζει πολλά δεδομένα προς επεξεργασία

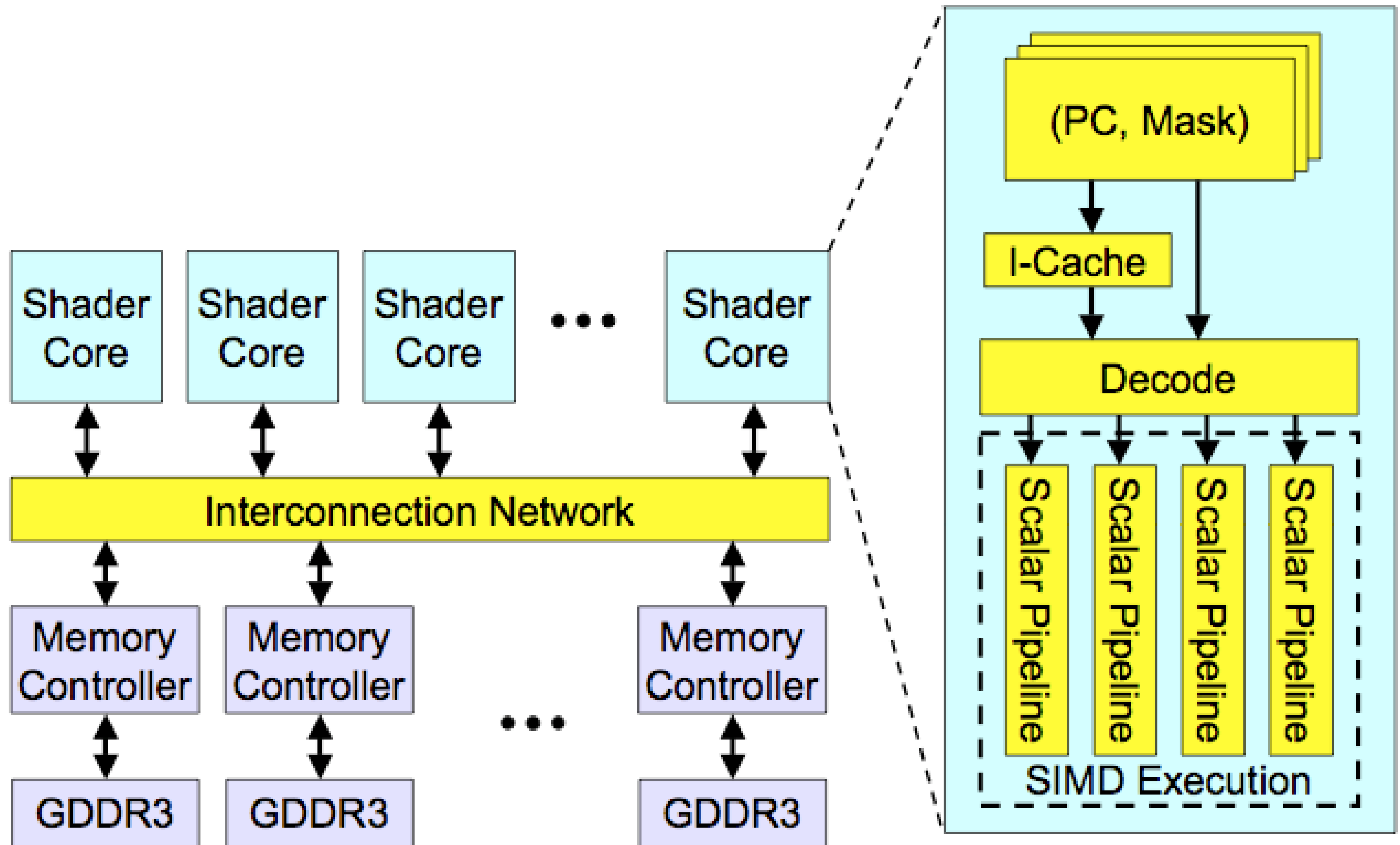
- SIMT – Single Instruction Multiple Threads
  - Πολλές ροές από scalar εντολές (μία για κάθε νήμα εκτέλεσης)
  - Πολλά νήματα ομαδοποιούνται *δυναμικά από το hardware* για να εκτελέσουν την ίδια ροή σε διαφορετικά δεδομένα

Πλεονεκτήματα SIMT Μοντέλου Εκτέλεσης

- Το υλικό μεταχειρίζεται κάθε thread σαν ανεξάρτητη οντότητα
  - Μπορεί να εκτελέσει κάθε thread ανεξάρτητα
  - Κερδίζοντας τα οφέλη του MIMD μοντέλου εκτέλεσης

- Το υλικό μπορεί να σχηματίσει δυναμικά groups από threads
  - Που εκτελούν την ίδια εντολή
  - Κερδίζοντας τα οφέλη του SIMD μοντέλου εκτέλεσης

GPU Αρχιτεκτονική



Threads and Blocks

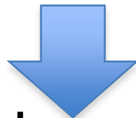
- Ένα *thread* συνδέεται με την επεξεργασία ενός υποσυνόλου του *data set (data elements)*
- Τα *threads* οργανώνονται σε *blocks*
- Τα *blocks* οργανώνονται σε ένα *grid*

- Το υλικό (hardware) της GPU διαχειρίζεται τα νήματα
  - Όχι οι εφαρμογές ή το λειτουργικό σύστημα
  - Multithreading SIMD execution

Παράδειγμα

CPU code

```
for (i = 0; i < 8192; ++i) {
        A[i] = B[i] * C[i];
}
```



CUDA code

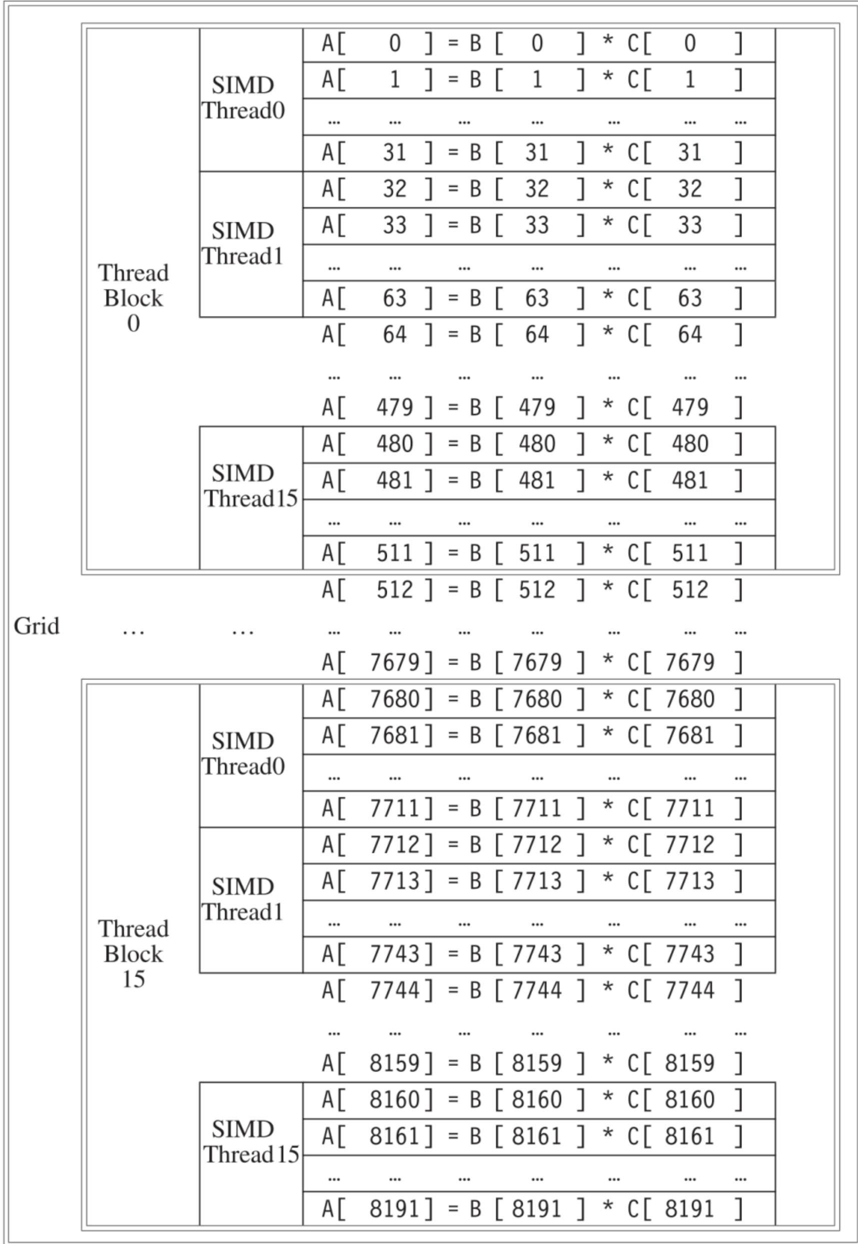
```
// there are 8192 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varB = B[tid];
  int varC = C[tid];
  A[tid] = varB + varC;
}
```

Παράδειγμα – Threads and Blocks

- Code that works over all elements is the **grid**
- **Thread blocks** break this down into manageable sizes
  - 512 threads per block (defined by the programmer)
- **Groups of 32 threads** combined into a **SIMD thread** or **“warp”**
- Grid size = 16 thread blocks
  - 8192 elements / 512 threads per block
- Block is analogous to a strip-mined vector loop with vector length of 32

- A thread block is assigned to a **multithreaded SIMD processor** by the thread block scheduler
- **Current-generation GPUs have tens of multithreaded SIMD processors**

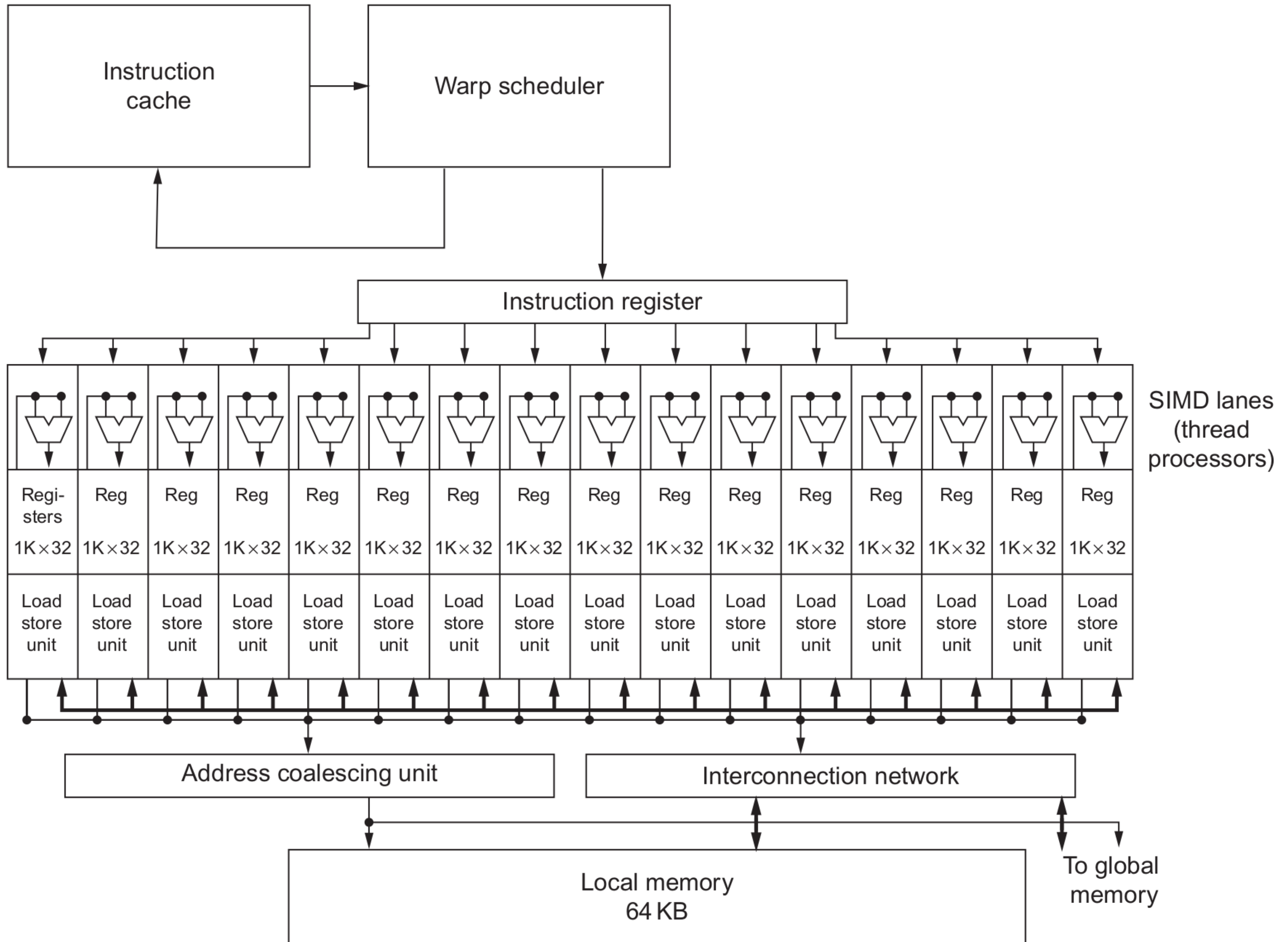
Παράδειγμα – Threads and Blocks



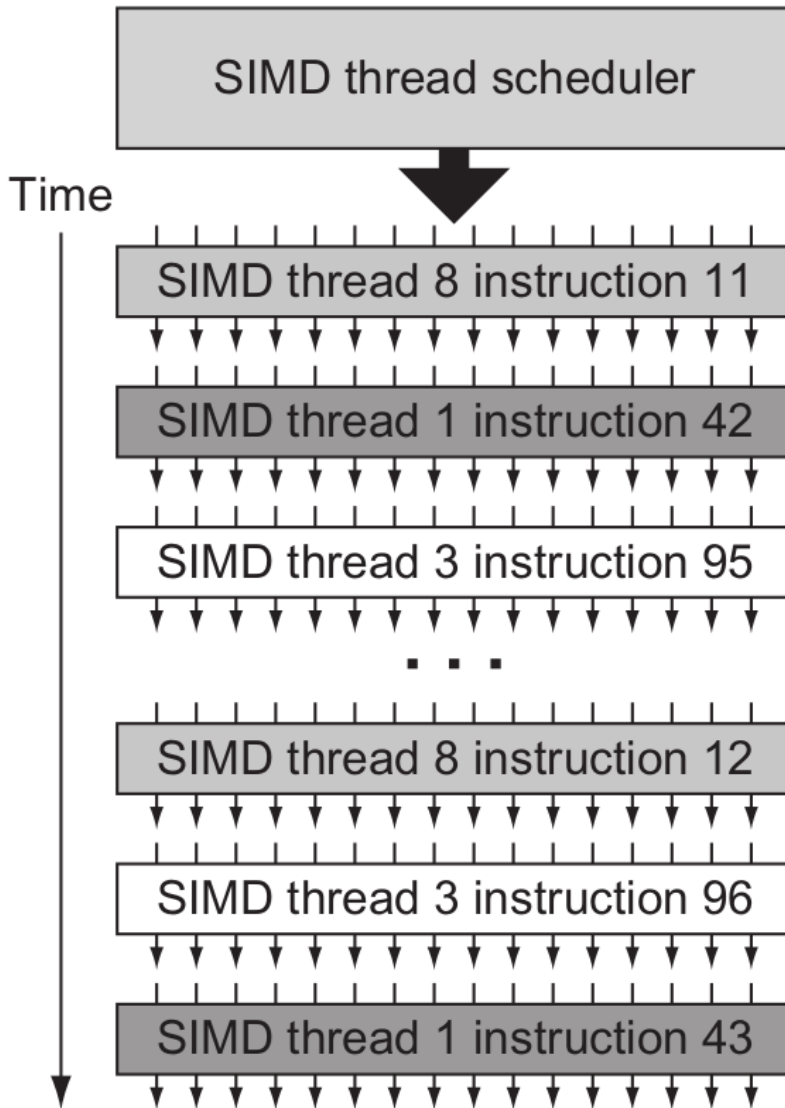
Παράδειγμα – GPU αρχιτεκτονική

- **Groups of 32 threads** combined into a **SIMD thread** or **“warp”**
  - Mapped to 16 (or 32) physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - **Each warp has its own PC**
  - Each thread in a warp has its own register set (depends on the architecture & limits the number of warps per SIMD processor)
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - **Wide and shallow** pipelined functional units compared to vector processors

Παράδειγμα – GPU αρχιτεκτονική



Warps are multithreaded on the SIMD core



- Warp == SIMD thread
- One warp is a single thread in the hardware
- Multiple warps are interleaved in execution on a single SIMD processor to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps, all mapped to single SIMD processor
- Can have multiple thread blocks executing on one SIMD processor

NVIDIA Instruction Set Architecture

- ISA is an abstraction of the hardware instruction set
  - **"Parallel Thread Execution (PTX)"**
    - opcode.type d,a,b,c;
  - Χρησιμοποιεί virtual registers
  - Η μετάφραση σε κώδικα μηχανής πραγματοποιείται από το software
  - Παράδειγμα:

```
shl.s32     R8, blockIdx, 9        ; Thread Block ID * Block size (512 or 29)
add.s32     R8, R8, threadIdx      ; R8 = i = my CUDA thread ID
ld.global.f64     RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64     RD2, [Y+R8]      ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4              ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2              ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0          ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

GPU Memory Structures

- **Shallow memory hierarchy**
  - **Multithreading → hides memory latency**
- Each SIMD Lane has private section of off-chip DRAM
  - **"Private memory"** ("local memory" in NVIDIA's terminology)
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - **"local memory"** ("shared memory" in NVIDIA's terminology)
  - Scratchpad memory → managed explicitly by the programmer
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - **"GPU memory"** ("global memory" in NVIDIA's terminology)
  - Host can read and write GPU memory

NVIDIA’s Pascal Architecture Innovations

- Each SIMD processor has:
  - Two SIMD thread (warp) schedulers, two instruction dispatch units
  - Two sets of:
    - 16 SIMD lanes (SIMD width=32, chime=2 cycles)
    - 16 load-store units,
    - 8 special function units
  - Two threads of SIMD instructions (warps) are scheduled every two clock cycles simultaneously
- Fast single-, double-, and half-precision
- High Bandwidth Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

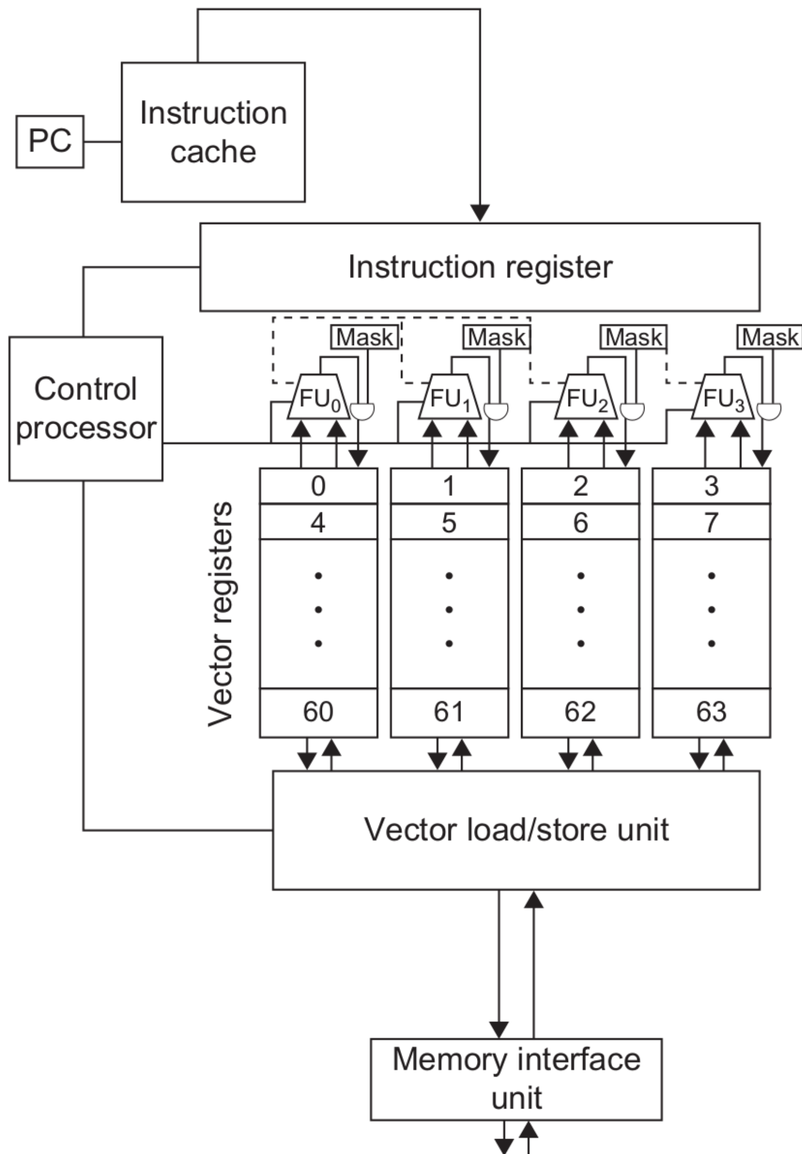
NVIDIA's Pascal SIMD Processor

GPUs vs. Vector Architectures

- Ομοιότητες με vector αρχιτεκτονικές
  - Δουλεύει επίσης καλά για data-level parallel προβλήματα
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Διαφορές με vector αρχιτεκτονικές
  - Δεν υπάρχει scalar επεξεργαστής
  - Χρησιμοποιεί multithreading για να κρύψει την καθυστέρηση στην πρόσβαση της μνήμης
  - Έχει πολλαπλά functional units, σε αντίθεση με τα λίγα deeply pipelined functional units που έχουν οι vector αρχιτεκτονικές

GPUs vs. Vector Architectures



Vector Architecture

PC
PC
PC
PC
Instruction cache
SIMD thread scheduler
Dispatch unit
Instruction register
Mask
FU0
FU1
FU2
FU3
Registers
0
1
1023
SIMD load/store unit
Address coalescing unit
Memory interface unit

GPU Architecture

SIMD Extensions vs. GPUs

- Οι GPUs έχουν περισσότερα SIMD lanes
- Οι GPUs υποστηρίζουν στο υλικό περισσότερα threads
- Και οι δύο έχουν 2:1 αναλογία μεταξύ double- and single-precision performance
- Και οι δύο έχουν 64-bit διευθύνσεις, αλλά οι GPUs έχουν μικρότερη μνήμη
- Τα SIMD extensions δεν υποστηρίζουν scatter-gather εντολές