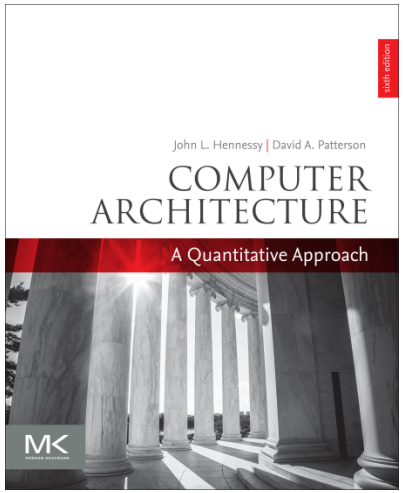
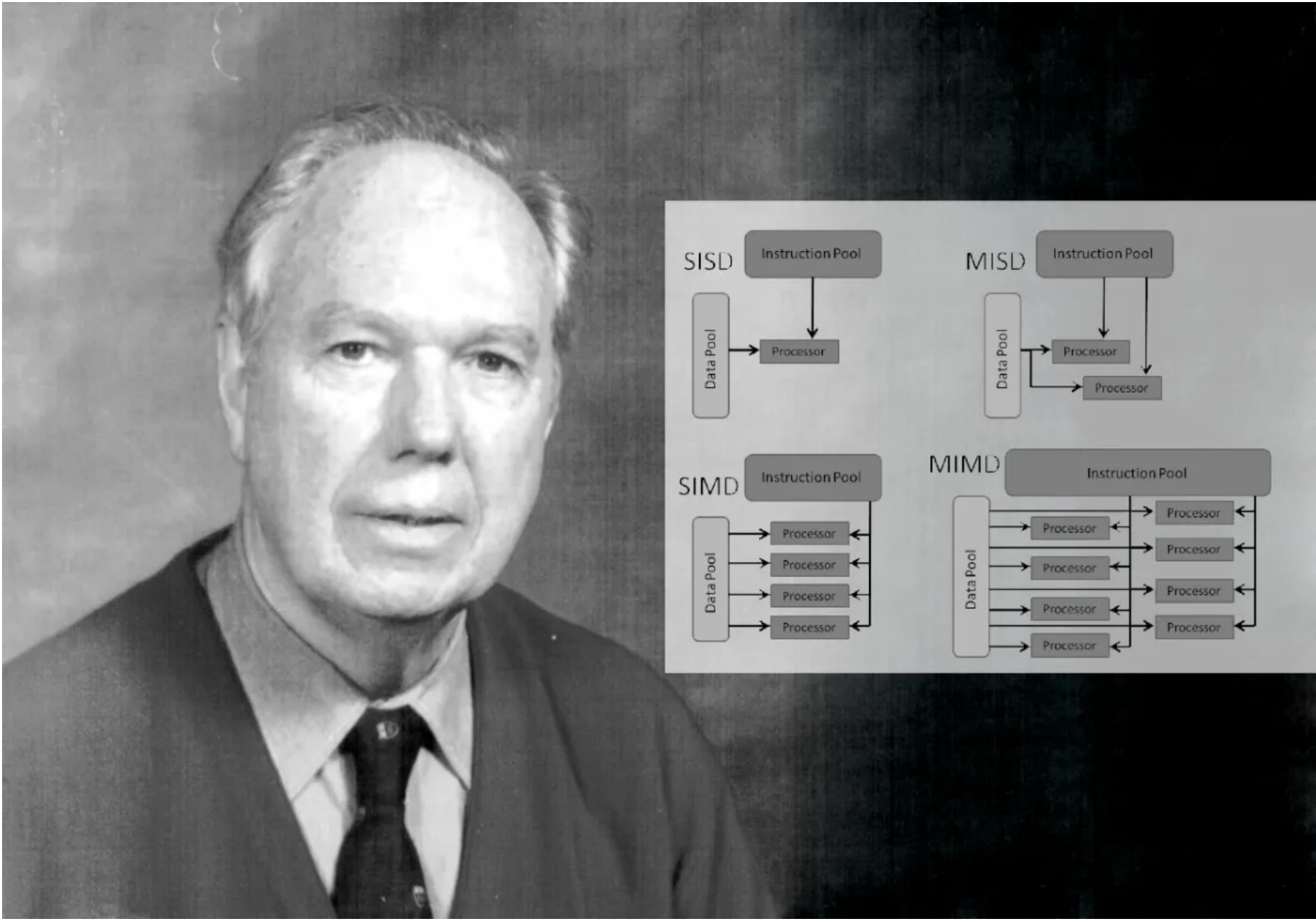


Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

4201 Lecture 13
2026B
עם עריכה, תוספות, ושינויים רבים
ע"י גיא תל-צור


SISD
Instruction Pool
Data Pool
Processor
MISD
Instruction Pool
Data Pool
Processor
Processor
SIMD
Instruction Pool
Data Pool
Processor
Processor
Processor
Processor
MIMD
Instruction Pool
Data Pool
Processor
Processor
Processor
Processor
Processor
Processor
Processor
Processor

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

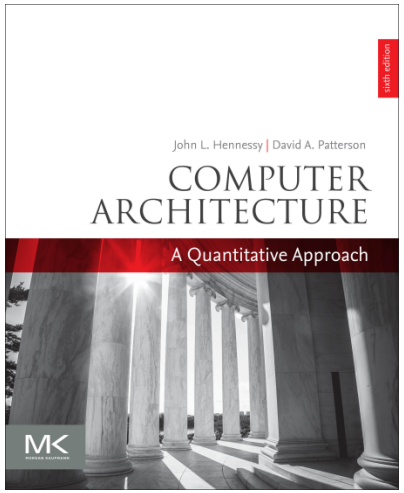
Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

INTRODUCTION

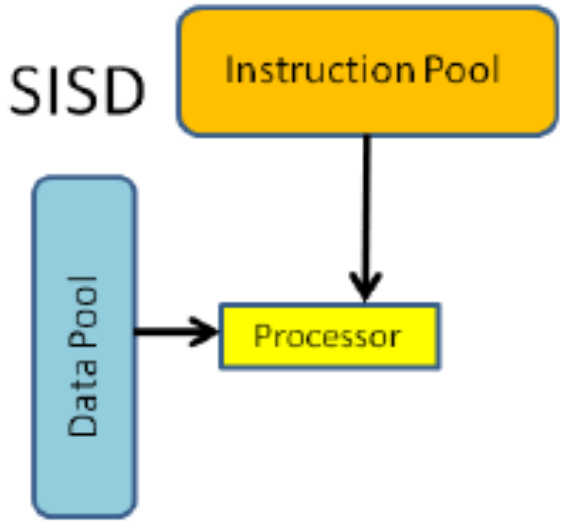
MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

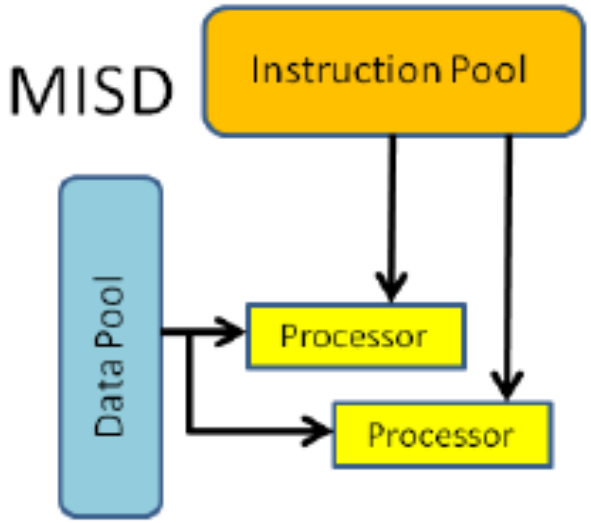
Flynn's Taxonomy

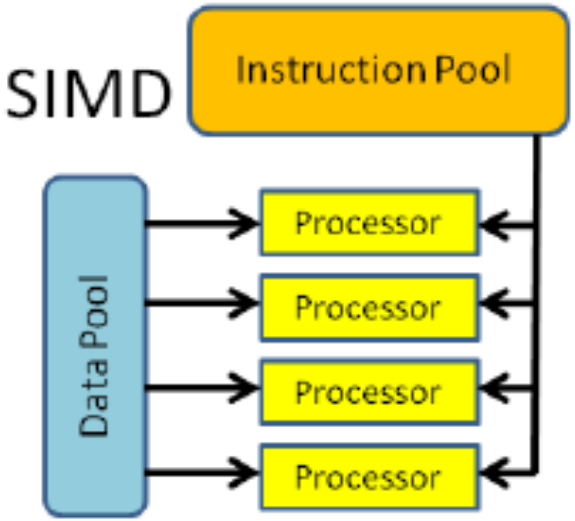


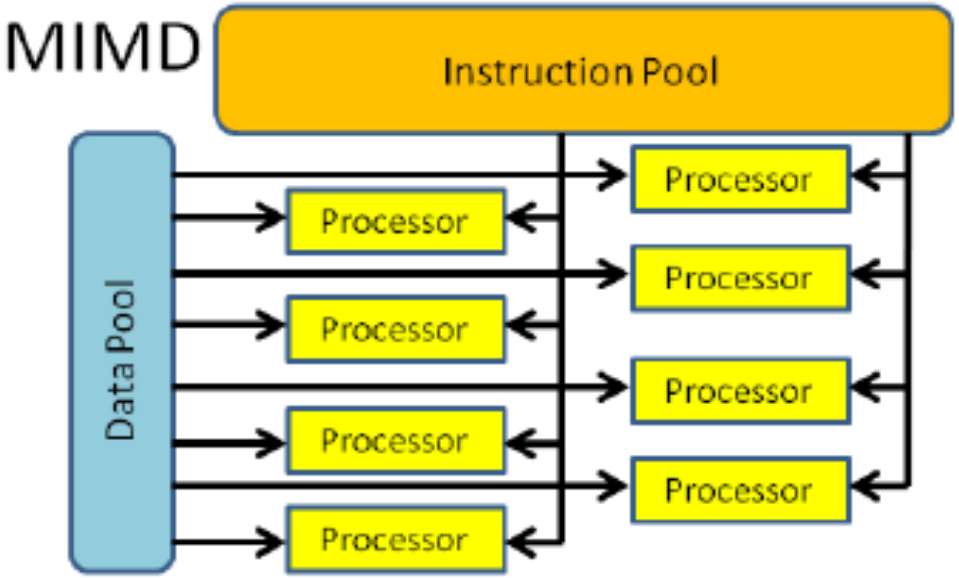
Data Streams \ Instruction Streams	one	many
one	**SISD** traditional von Neumann single CPU computer	**MISD** May be pipelined Computers
many	**SIMD** Vector processors fine grained data Parallel computers	**MIMD** Multi computers Multiprocessors

Flynn's Taxonomy









Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors

data parallel vs. task parallel

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory

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

יוסבר בהמשך

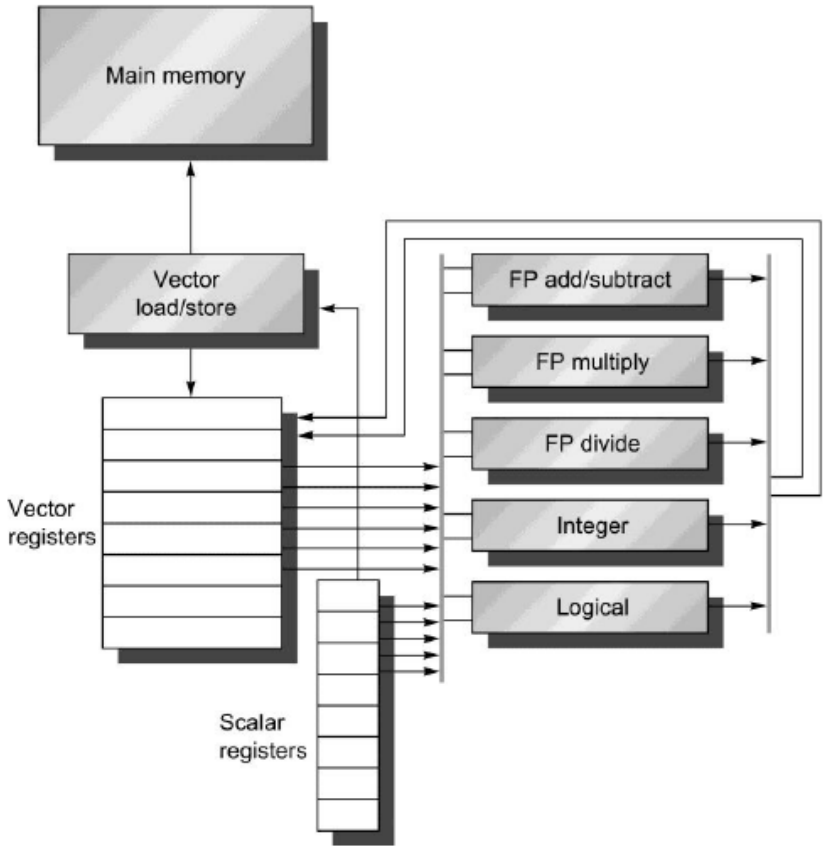


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. There are also 32 vector registers, and all the functional units are vector functional units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches *(thick gray lines)* connects these ports to the inputs and outputs of the vector functional units.

RV64V - בהשראת CRAY-1

https://rvv-isadoc.readthedocs.io/en/latest/index.html לפתוח את הקישור

- Example architecture: RV64V
  - Loosely based on Cray-1
  - 32 **VLEN**-bit vector registers

    (בשקפים המקוריים נכתב 62 ביט וזו טעות)
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 31 general-purpose registers
    - 32 floating-point registers

Cray-1

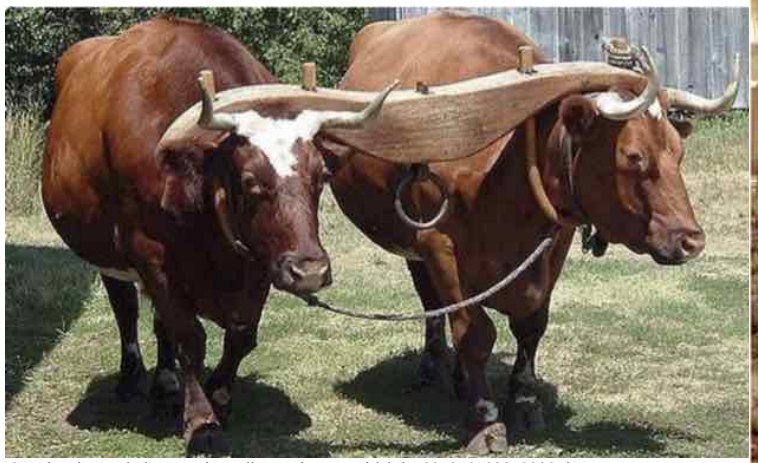
Design	
Manufacturer	Cray Research
Designer	Seymour Cray
Release date	1975
Units sold	Over 80
Price	US$7.9 million in 1977 (equivalent to $33.3 million in 2019)
Casing	
Dimensions	Height: 196 cm (77 in)[1] Dia. (base): 263 cm (104 in)[1] Dia. (columns): 145 cm (57 in)[1]
Weight	5.5 tons (Cray-1A)
Power	115 kW @ 208 V 400 Hz[1]
System	
Front-end	Data General Eclipse
Operating system	COS & UNICOS
CPU	64-bit processor @ 80 MHz[1]
Memory	8.39 Megabytes (up to 1 048 576 words)[1]
Storage	303 Megabytes (DD19 Unit)[1]
FLOPS	160 MFLOPS

10/11/2018
SC18. Dallas, TX

CESCA בברצלונה

Seymour Cray, Leader in Supercomputer Design

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"



© amityrebecca / Pinterest. https://www.pinterest.ch/pin/473018767088408061/

© Scott Sinklier / Corbis. http://america.aljazeera.com/articles/2015/2/20/the-short-brutal-life-of-male-chickens.html

https://en.wikipedia.org/wiki/Seymour_Cray

VMIPS Instructions – גרסה לימודית..

להתרשם מהסגנון; אין צורך לשנן...

- .vv: two vector operands
- .vs and .sv: vector and scalar operands
- LV/SV: vector load and vector store from address
- Example: **DAXPY: double Y = a*X+Y**

```
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**

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors, for example vlen=32 or 64
  - Structural hazards, for example two vector multiply instructions, but the processor only has one physical Vector Multiply functional unit,
  - Data dependencies, for example:

```
vfmul.vv v2, v0, v1   # Instruction 1 calculates v2
vfadd.vv v4, v2, v3   # Instruction 2 needs v2
```

- RV64V functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convey לשנע יחדאת האופרנדים של ההוראה הוקטורית*
  - Set of vector instructions that could potentially execute together

RV64 DAXPY

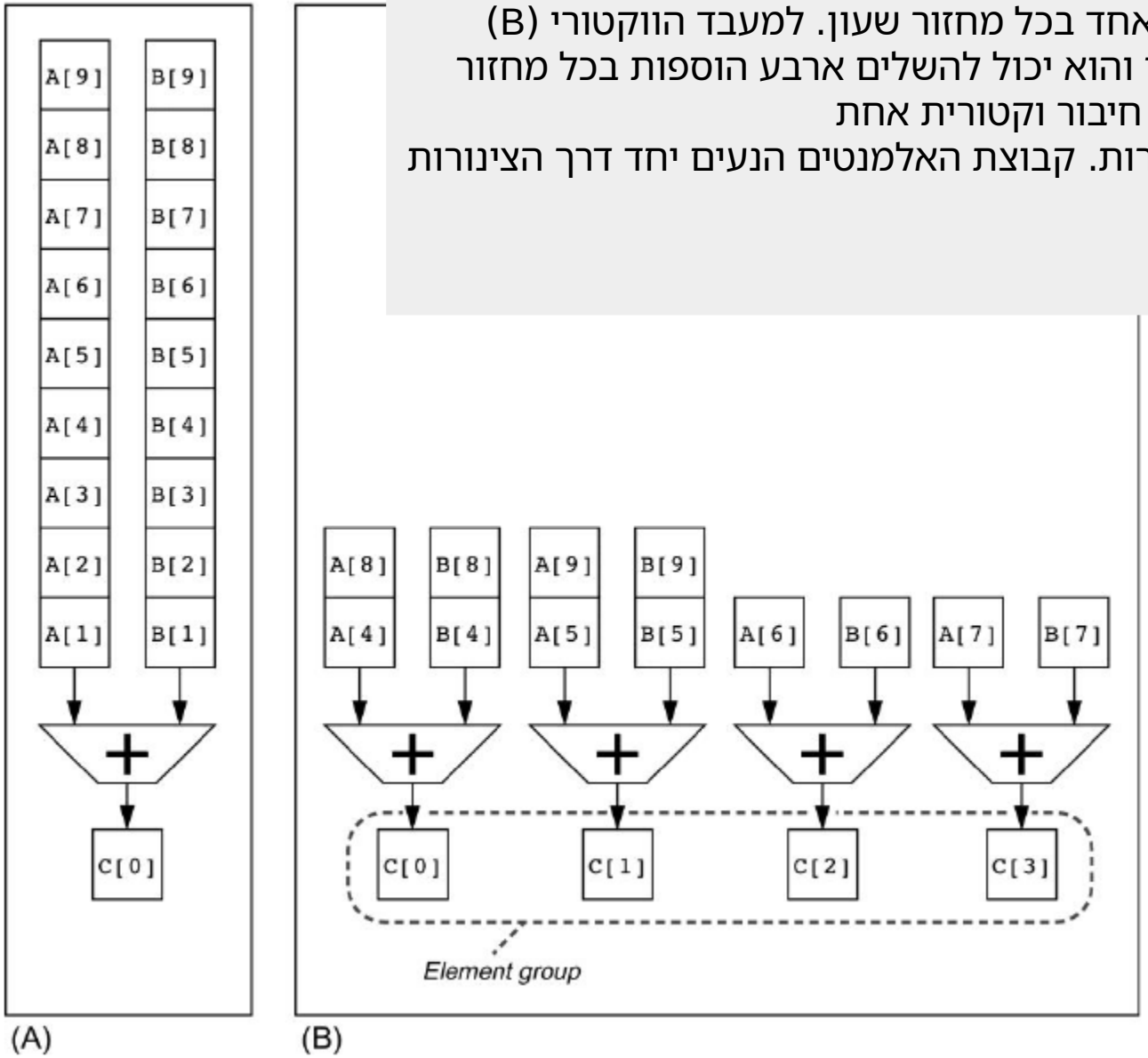
```
# DAXPY: y[i] = a * x[i] + y[i]
# Arguments:
#   fa0 = scalar 'a' (F register)
#   x5  = base address of vector x
#   x6  = base address of vector y
#   x7  = number of elements (vector length)

  # Set vector length based on element size and available data
  li      t0, 64           # Element size: 64 bits (8 bytes)
  vsetvli t1, x7, e64, m1 # Configure for 64-bit elements, LMUL=1 - השקף הבא
  # Load vector x
  vle64.v v0, (x5)         # Load x into vector register v0
  # Multiply: v1 = a * x
  vfmul.vf v1, v0, fa0     # Vector-scalar multiply (fa0 = a)
  # Load vector y
  vle64.v v2, (x6)         # Load y into vector register v2
  # Add: v3 = (a * x) + y
  vfadd.vv v3, v1, v2      # Vector-vector add
  # Store result back to v
  vse64.v v3, (x6)
```

Instruction	Description
vsetvli	Set vector length (configures VL based on element size)
vle64.v	Load vector of 64-bit elements
vse64.v	Store vector of 64-bit elements
vfmul.vf	Vector × scalar FP multiply
vfadd.vv	Vector + vector FP add

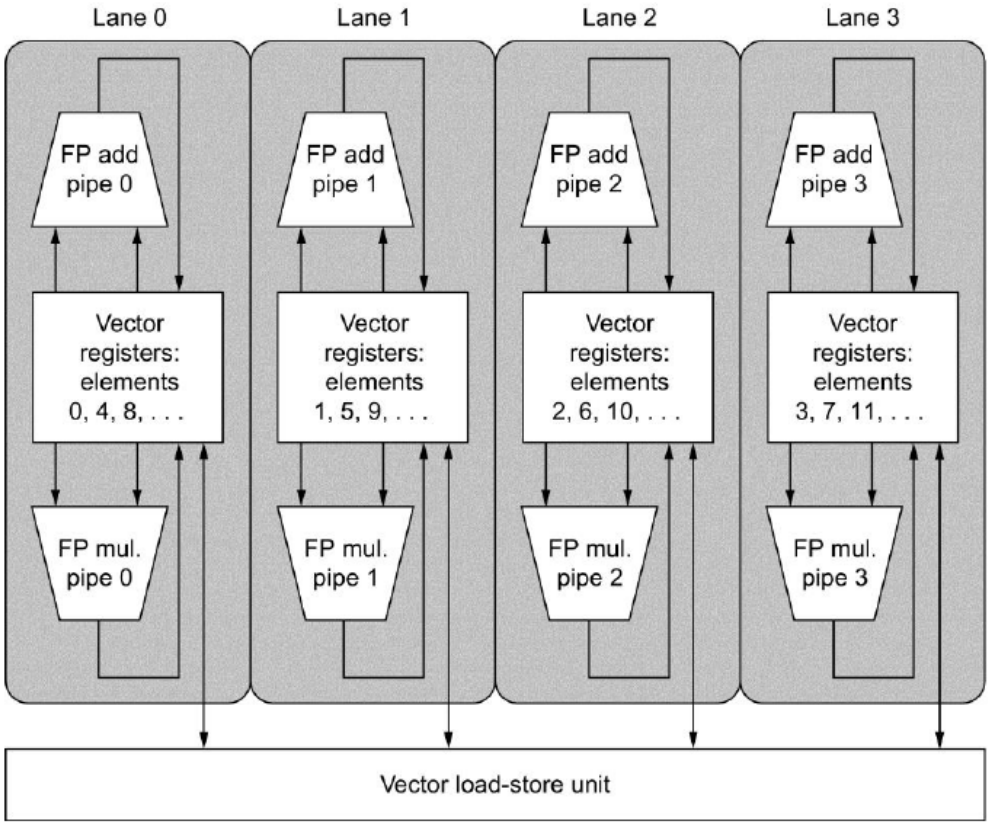
הסבר ההוראה מהשקף הקודם
vsetvli t1, x7, e64, m1

```
vsetvli  t1,  x7,  e64,  m1
   |      |    |    |     |
   |      |    |    |     +--- [LMUL] Vector Register Grouping (Multiplier = 1)
   |      |    |    +--------- [SEW] Selected Element Width (64-bit data)
   |      |    +-------------- [AVL] Application Vector Length (Requested elements in x7)
   |      +------------------- [Destination] General Purpose Register to store actual 'vl'
   +-------------------------- Instruction: "Vector Set Vector Length Item"
```

שימוש ביחידות פונקציונליות מרובות כדי לשפר את הביצועים של הוראת חיבור וקטורית אחת, C = A + B. למעבד הווקטורי (A) משמאל יש צינור חיבור יחיד והוא יכול להשלים חיבור אחד בכל מחזור שעון. למעבד הווקטורי (B) מימין יש ארבעה צינורות חיבור והוא יכול להשלים ארבע הוספות בכל מחזור שעון. האלמנטים בתוך הוראת חיבור וקטורית אחת
משולבים על פני ארבעת הצינורות. קבוצת האלמנטים הנעים יחד דרך הצינורות נקראת קבוצת אלמנטים.



מבנה של יחידת וקטור המכילה ארבעה נתיבים. זיכרון אוגר הווקטורים מחולק על פני הנתיבים, כאשר כל נתיב מכיל כל אלמנט רביעי של כל אוגר וקטור. האיור מציג שלוש יחידות פונקציונליות וקטוריות: יחידת חיבור FP, יחידת כפל FP, ויחידת טעינה-אחסון. כל אחת מיחידות האריתמטיקה הווקטוריות מכילה ארבעה צינורות ביצוע, אחד לכל נתיב, הפועלים יחד כדי להשלים הוראת וקטור אחת. איור זה אינו מציג את הנתיב לספק את האופרנד הסקלרי עבור הוראות וקטור-סקלריות, אך המעבד הסקלרי משדר ערך סקלרי לכל הנתיבים.



Vector Length Register

```
for (i=0; i <n; i=i+1) Y[i] = a * X[i] + Y[i];
                vsetdcfg 2 DP FP # Enable 2 64b Fl.Pt. registers
                fld f0,a         # Load scalar a
        loop:   setvl t0,a0      # vl = t0 = min(mv
                vld v0,x5        # Load ve
                slli t1,t0,3     # t1
                add x5,x5,t1     #  X by vl*8
                vmul             #  mult
                                 #  vector Y
                                 # Vector-vector add
                                 # n -= vl (t0)
                vst v1,x6        # Store the sum into Y
                add x6,x6,t1     # Increment pointer to Y by vl*8
                bnez a0,loop     # Repeat if n != 0
                vdisable         # Disable vector regs}
```

out of date

הקוד כאן מדגים
גרסה ישנה של
RVV 0.7 ולא
RVV 1.0

DAXPY (long vectors), RVV 1.0

```
# RVV 1.0 DAXPY
# Inputs: a0 = ptr to 'a', a1 = n (element count), x5 = base X, x6 = base Y
    fld       fa0, 0(a0)      # Load scalar a
    mv        a2, x5          # Save base X to working register
    mv        a3, x6          # Save base Y to working register
loop:
    vsetvli   t1, a1, e64, m1 # t1 = vl (elements processed this iteration)

    vle64.v   v0, (a2)        # Load X elements from (a2)
    vle64.v   v2, (a3)        # Load Y elements from (a3)

    vfmul.vf v1, v0, fa0      # v1 = a * X
    vfadd.vv v3, v1, v2       # v3 = a * X + Y
    vse64.v   v3, (a3)        # Store result to (a3)

    # --- Address Calculation ---
    slli      t2, t1, 3  # t2 = t1 * 8 (convert element count to byte offset)
    add       a2, a2, t2      # Properly advance X pointer by byte offset
    add       a3, a3, t2      # Properly advance Y pointer by byte offset

    sub       a1, a1, t1      # Decrement remaining element count: n -= vl
    bnez      a1, loop        # Repeat if n != 0
```

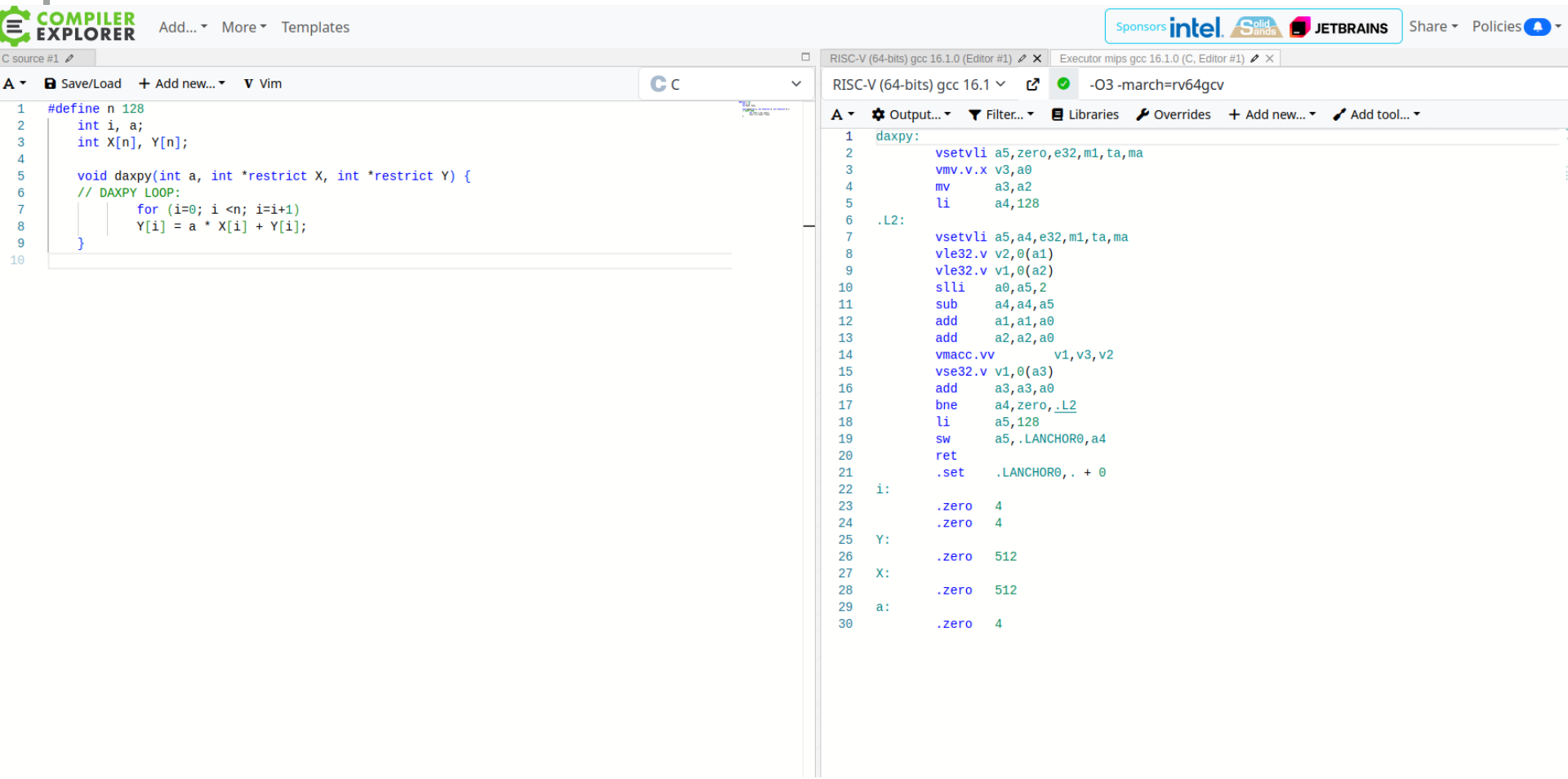
במקום שתי הוראות אלה (הן נכונות) ניתן לרשום הוראה מקוצרת אחת FMA – ראו השקף הבא

Fused Multiply Add

```
vle64.v  v0, (a2)
vle64.v  v2, (a3)
vfmacc.vf v2, fa0, v0    # v2 = (fa0 * v0) + v2
vse64.v  v2, (a3)
```

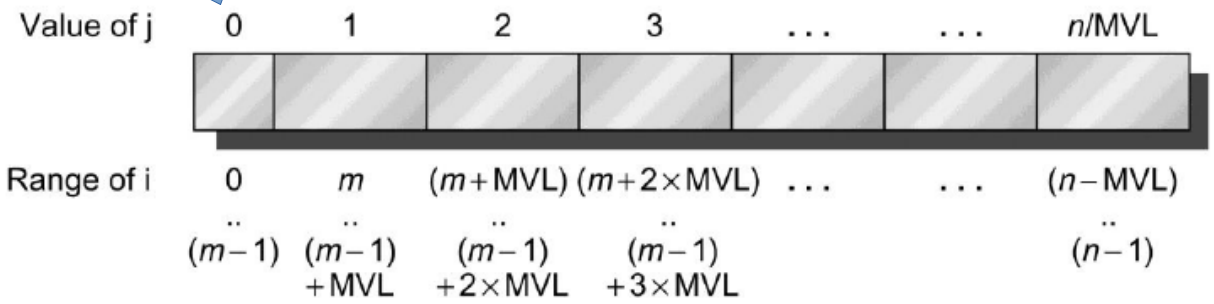
Instruction	Operation
vfmacc.vv	vd += vs1 * vs2
vfmacc.vf	vd += scalar * vs2
vfnmacc.vv	vd -= vs1 * vs2 (negated add)
vfnmacc.vf	vd -= scalar * vs2
vfmsbc.vv	vd -= vs1 * vs2 with borrow
vfmul.vf	vd = scalar * vs2 (multiply only)
vfadd.vv	vd = vs1 + vs2 (add only)

Compiler Explorer, https://godbolt.org/



A vector of arbitrary length processed with strip mining

השארית תחילה



ב- X86 המתכנת נדרש לתת הנחיה לקומפיילר כגון:

`pragma omp simd` or `#pragma vector always #`

לעומת זאת בארכיטוקטורת RISC-V RVV מספיק לקמפל עם דגל O3 והוקטוריזציה תתבצע מעצמה

Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, we use the variable *m* for the expression (n % MVL). (The C operator % is modulo.)

Vector Mask Registers

- Consider:

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

- Use **predicate register** to “disable” elem

```
vsetdcfg      2*FP64    # Enable 2 64b FP vector regs
vsetpcfgi     1         # Enable 1 predicate re
vld                     v0,x5       # L
vld                     v1,x6
fmv.d.x       f0,x0
vpne                                # v0(i)!=f0
vsub                                ct under vector mask
vst                                 # Store the result in X
vdisable                            able vector registers
vpdisable               # Disable predicate registers
```

out of date

V 0.7
in :(

קיים רק בגרסאות
וקטוריות של RISC-V
ומיוחס ל-v0
- תיקון בשקף הבא

RVV 1.0 corrected

```
# Input: a0 = 64 (count), x5 = &X, x6 = &Y

li       t0, 64
vsetvli  t1, t0, e64, m1, ta, ma   # Configure for 64-bit FP elements, m1
LMUL

vle64.v  v1, (x5)                  # Load vector X into v1
vle64.v  v2, (x6)                  # Load vector Y into v2

# 1. Generate the mask: Compare X != 0.0
# f0 is the standard RISC-V scalar FP register for 0.0 (if cleared)
# or we can clear a regular scalar register.
fmv.d.x  f0, zero                  # f0 = 0.0
vmfne.vf v0, v1, f0                # v0 is the mask. v0[i] = (v1[i] != 0.0)

# 2. Masked Subtract: X = X - Y under mask v0.t
# We overwrite v1 (X) in-place so unmasked elements stay untouched!
vfsub.vv v1, v1, v2, v0.t          # v1 = v1 - v2 where v0.t is true

# 3. Store the result back to X
vse64.v  v1, (x5)                  # Store updated X back to memory
```

v0 is the designated Mask Register

Vector Masks :גיא

Register(s)	Purpose
v0–v7	Vector mask (predicate) registers for conditional execution
vtype	CSR controlling element width, LMUL, SEW, etc.
vstart	CSR showing/sets the index of the first active element (for vector tail handling)
vl	CSR holding the current vector length
vcsr	Vector control and status register

The mask registers are 1 bit per element, so their length equals the MVL (Maximum Vector Length) supported by the implementation. Per the spec:

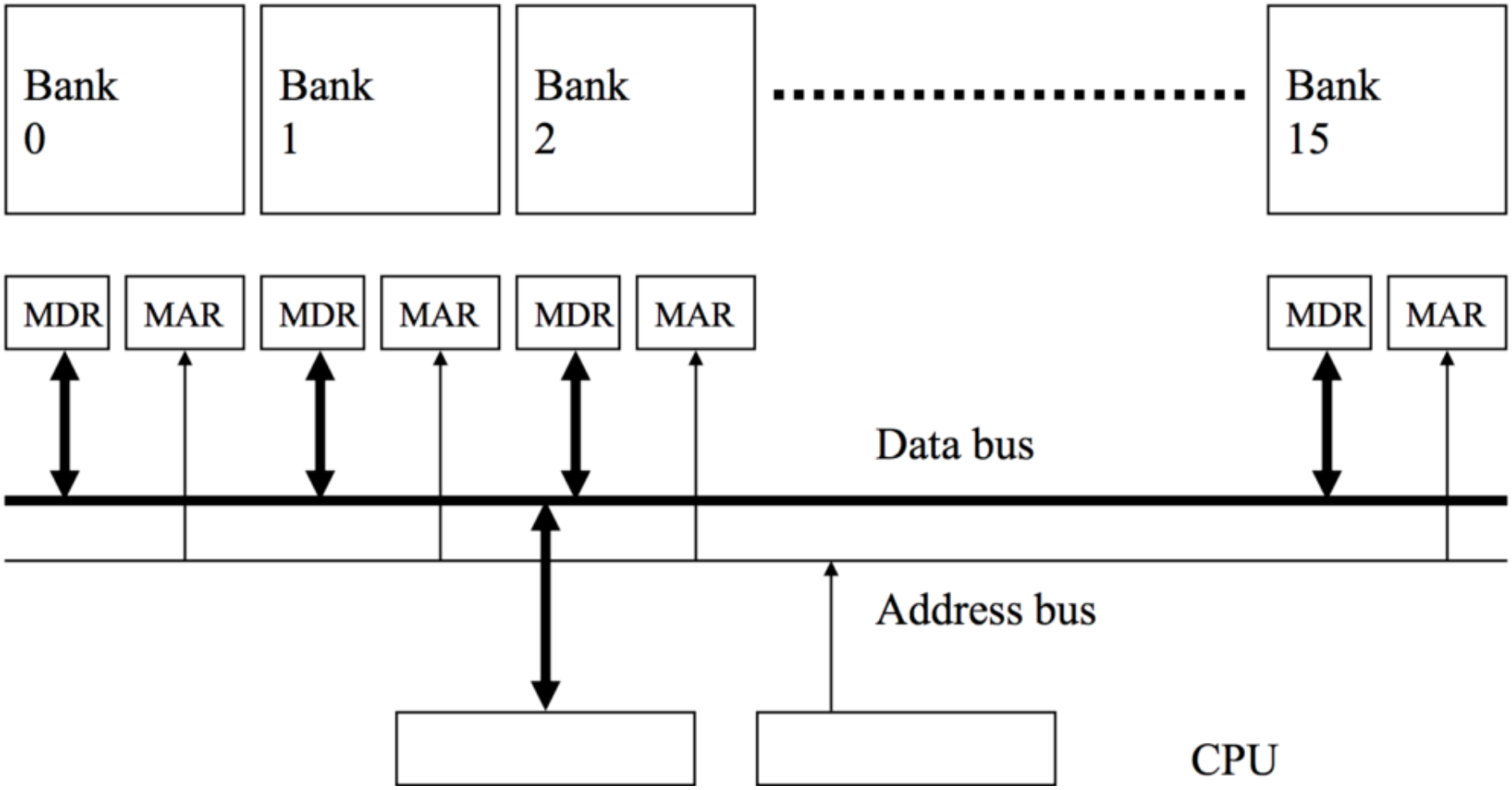
- MVL ≥ 64 (minimum required)
- Most implementations support MVL of 64, 128, 256, or 512 elements

if the hardware has MVL=256, each mask register (v0–v7) is effectively 256 bits (one bit per vector element position).

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words (need independent bank addressing)
  - Support multiple vector processors sharing the same memory

- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?
    - 32x(4+2)x15/2.167 = ~1330 banks



MDR = Memory Data Register
MAR = Memory Address Register

Stride

- Consider:

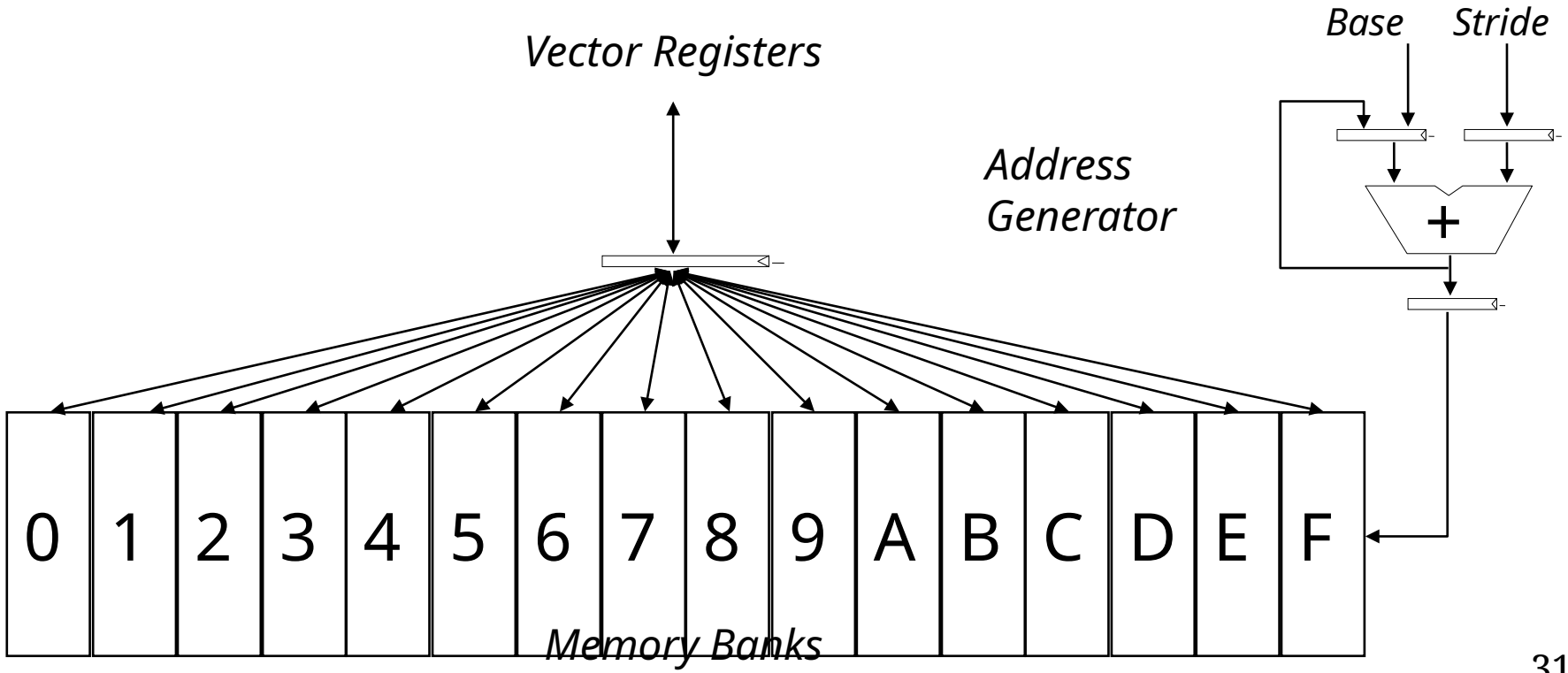
```
for (i = 0; i < 100; i=i+1)
        for (j = 0; j < 100; j=j+1) {
                A[i][j] = 0.0;
                for (k = 0; k < 100; k=k+1)
                A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time
  - LCM = Least Common Multiple

Vector Memory System

- Next address = Previous address + Stride
- If stride = = 1 & consecutive elements interleaved across banks & number of banks is big enough, (#banks > Latency cycles)

then bank latency is not a bottleneck and we can sustain 1 element/cycle throughput



Picture credit: Krste Asanovic

Scatter-Gather

- **Consider:**

  for (i = 0; i < n; i=i+1)

  A[K[i]] = A[K[i]] + C[M[i]];

- Use index vector:

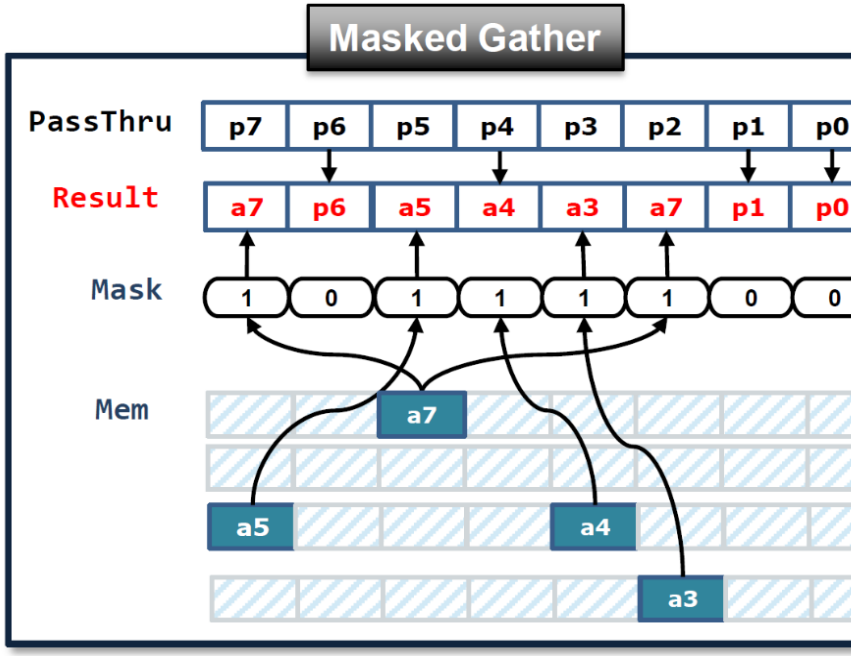

```
vsetdcfg 4*FP64                          ...tor registers
vld                                      ...K[]
vldx                                     # Load A[K[]]
vld                                      # Load M[]
vldi            v3, x6, v2               # Load C[M[]]
vadd            v1, v1, v3               # Add them
vstx            v1, x5, v0               # Store A[K[]]
vdisable                                 # Disable vector registers
```

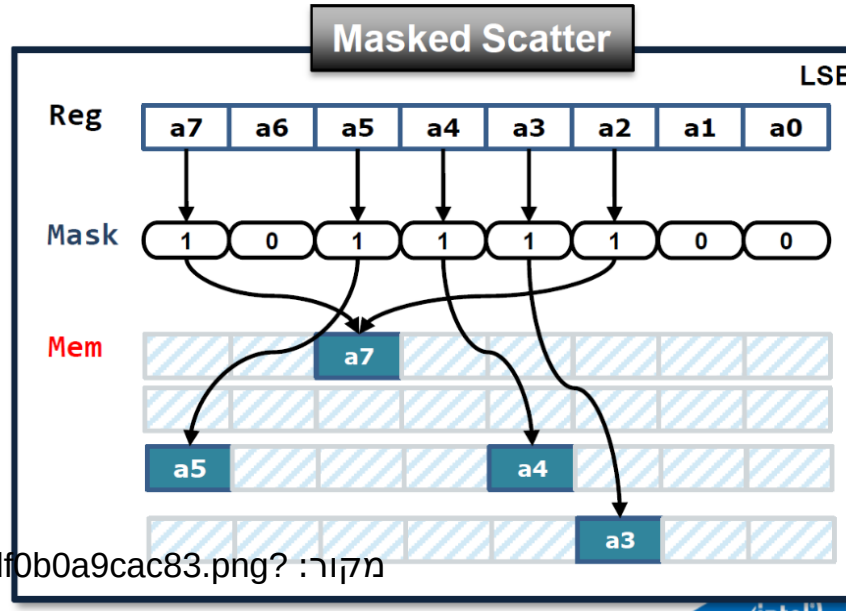
RVV 1.0 corrected

```
# Input: a0 = n, x5 = &A, x6 = &C, x7 = &K, x28 = &M
loop:
    # 1. Dynamically configure vector length for 64-bit elements
    vsetvli  t1, a0, e64, m1, ta, ma   # t1 = vl (elements processed)

    # 2. Load the index vector M[] and gather C[M[i]]
    vle64.v  v2, (x28)                 # v2 = Load M[] offsets
    vluxei64.v v3, (x6), v2            # v3 = C[M[i]]
    # 3. Load the index vector K[] and gather A[K[i]]
    # Changed v0 -> v4 to avoid using the reserved mask register v0
    vle64.v  v4, (x7)                  # v4 = Load K[] offsets
    vluxei64.v v1, (x5), v4            # v1 = A[K[i]]
    # 4. Perform the floating-point addition
    vfadd.vv v1, v1, v3                # v1 = A[K[i]] + C[M[i]]
    # 5. Scatter the result back to A[K[i]]
    vsuxei64.v v1, (x5), v4            # Store using v4 offsets
    # 6. Advance the index array pointers (t1 elements * 8 bytes)
    slli     t2, t1, 3                 # t2 = vl * 8 bytes
    add      x7, x7, t2                # Advance K[] pointer
    add      x28, x28, t2              # Advance M[] pointer
    # Note: Base addresses x5 (&A) and x6 (&C) do NOT change because
    # the index offsets themselves handle the jumping around memory!
    # 7. Decrement total count and repeat if elements remain
    sub      a0, a0, t1                # n -= vl
    bnez     a0, loop                  # Loop if n > 0
```

Gather And Scatter How does it work?

- Works with vector of pointers
- Access memory according to the provided mask
- The mask holds a bit per lane
- The masked-off lanes are taken from the PassThru operand.
- No memory access for all-zero mask
- Scatter with overlapping vector indices are guaranteed to be ordered from LSB to MSB





תרשימים להמחשת הרעיון

מקור: https://sw.cool3c.com/user/93262/2020/78039cec-887e-47a0-bdf1-df0b0a9cac83.png?fit=max&w=2400&q=80

In X86-64: AVX512, VSCATTER, VGATHER

https://en.wikipedia.org/wiki/AVX-512

Prefetch [edit]

AVX-512 prefetch instructions contain new prefetch operations for the new scatter and gather functionality introduced in AVX2 an

Instruction	
`VGATHERPF0DPS`, `VGATHERPF0QPS`, `VGATHERPF0DPD`, `VGATHERPF0QPD`	Using signed dword/qword indices, pre
`VGATHERPF1DPS`, `VGATHERPF1QPS`, `VGATHERPF1DPD`, `VGATHERPF1QPD`	Using signed dword/qword indices, pre
`VSCATTERPF0DPS`, `VSCATTERPF0QPS`, `VSCATTERPF0DPD`, `VSCATTERPF0QPD`	Using signed dword/qword indices, pre
`VSCATTERPF1DPS`, `VSCATTERPF1QPS`, `VSCATTERPF1DPD`, `VSCATTERPF1QPD`	Using signed dword/qword indices, pre

סט שלם של הוראות המאפשרות מניפולציות על וקטורים:

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-xeon-processor-d-2100-product-family-technical-overview.html

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

MMX TECHNOLOGY EXTENSION TO THE INTEL ARCHITECTURE

Alex Peleg
Uri Weiser
Intel Israel Design Center

Designed to accelerate multimedia and communications software, MMX technology improves performance by introducing data types and instructions to the IA that exploit the parallelism in these applications.

MMX technology extends the Intel architecture (IA) to improve the performance of multimedia, communications, and other numeric-intensive applications. It uses a SIMD (single-instruction, multiple-data) technique to exploit the parallelism inherent in many algorithms, producing full application performance of 1.5 to 2 times faster than the same applications run on the same processor without MMX. The extension also maintains full compatibility with existing IA microprocessors, operating systems, and applications while providing new instructions and data types that applications can use to achieve a higher level of performance on the host CPU.

We formed a group of microprocessor architects and select software developers that worked together to define MMX technology with clear guidelines and specific goals in mind. We aimed not only to improve performance and retain backward compatibility but also to design an innovative, state-of-the-art architecture that would scale with future higher frequency, advanced-microarchitecture processors.

We analyzed a wide range of software applications, including graphics, MPEG video, music synthesis, speech compression, image processing, games, speech recognition, and videoconferencing. The analysis results revealed that although the applications cross different domains, in most cases the computationally intensive, time-consuming routines within them have common characteristics. These are small native data types (for example, 8-bit pixels, 16-bit audio samples), localized recurring operations, and parallelism. This pointed us in the direction of a SIMD architecture that would exploit the available parallelism. The SIMD technique lets one instruction perform the same operation on multiple data elements, in parallel. It is the fundamental reason for the speedups achieved by MMX technology.

The imperative requirement that MMX technology retain full compatibility with existing operating systems and software placed some interesting constraints on our design. For example, we couldn't introduce new states, such as a register set, additional control registers, or new condition codes. Also, we couldn't permit new events (for example, new numeric exceptions) that existing operating systems would not acknowledge. Last but not least, we had to ensure that applications or operating systems would not rely on the invalid-opcode exception that present-day CPUs generate when executing the new MMX opcodes (for example, to internally signal some special event).

Basically, we used the existing floating-point registers as the MMX registers to maintain compatibility. We made this decision 1) to enjoy the existing 64-bit width possible with the floating-point registers and 2) as the main means to maintain full IA compatibility with existing applications and operating systems.

MMX technology and other processors

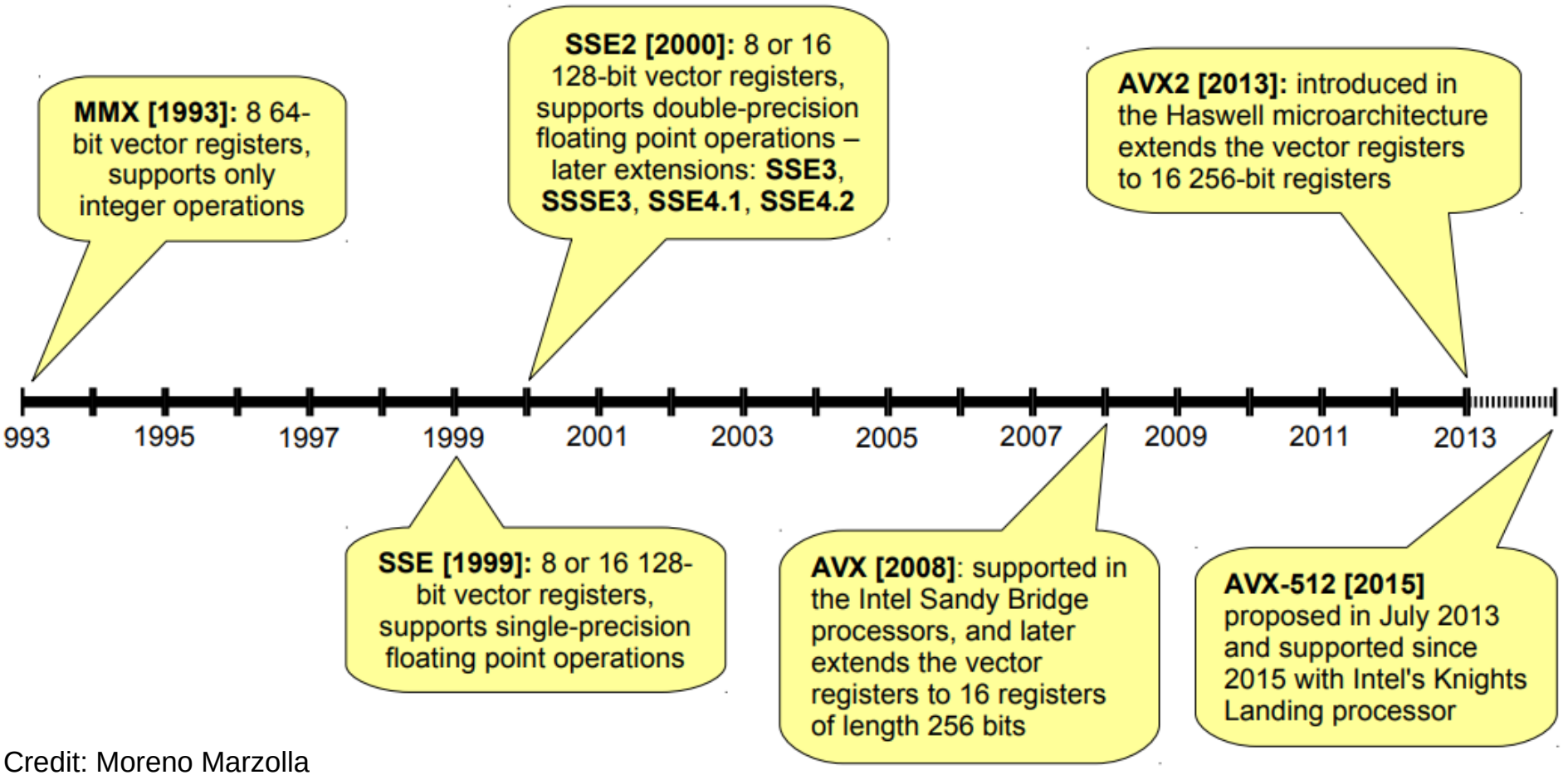
MMX instructions have been implemented in other general-purpose processor architectures. The 8607 and MC88110[2] processors implemented a limited set of SIMD graphics instructions targeted to support a few basic algorithms in 3D graphics, for example, Z buffering. More recently, the PA-7100LC processor[3] together with the SIMD instruction support in the PA-RISC 2.0 Architecture,[3] and the UltraSparc processors[4] have added a set of extensions to accelerate multimedia applications. Like MMX technology, both these architectures support parallel operations on multiple small packed data elements.

Like the UltraSparc design, MMX technology shares the floating-point registers between the floating-point and MMX instruc-

42 IEEE Micro

0272-1732/96/$5.00 © 1996 IEEE

Intel SIMD extensions timeline



Credit: Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
http://www.moreno.marzolla.name/

Example SIMD Code

- Example DXPY:

```
        fld       f0,a          # Load scalar a
        splat.4D  f0,f0         # Make 4 copies of a
        addi      x28,x5,#256   # Last addr
Loop:   fld.4D    f1,0(x5)      # Lo
        fmul.4D   f1,f1,f0
        fld.4D    f2,0(x        Y[i+3]
        fadd.4D                 X[i]+Y[i]...
                                # a x X[i+3]+Y[i+3]
        fsd.4D                  # 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
```

out of date

Corrected RVV 1.0 DAXPY

```
# Input: fa0 = a, a0 = n, x5 = &X, x6 = &Y

Loop:
    # 1. Dynamically configure vector length for 64-bit float elements (e64)
    # t0 will receive the actual number of elements granted by hardware this
iteration.
    vsetvli t0, a0, e64, m1, ta, ma
    # 2. Vector Load X and Y elements
    vle64.v v1, (x5)           # Load X elements into v1
    vle64.v v2, (x6)           # Load Y elements into v2
    # 3. Vector-Scalar Fused Multiply-Add: Y = a * X + Y
    # vfmacc.vf multiplies scalar fa0 by vector v1, adds vector v2,
    # and destructive-writes the result directly into v2.
    vfmacc.vf v2, fa0, v1

    # 4. Vector Store updated Y elements back to memory
    vse64.v v2, (x6)
    # 5. Calculate byte offset for the next iteration (t0 elements * 8 bytes)
    slli    t1, t0, 3          # t1 = t0 * 8 (shift left by 3)
    add     x5, x5, t1         # Advance pointer X
    add     x6, x6, t1         # Advance pointer Y
    # 6. Decrement total element count and loop if elements remain
    sub     a0, a0, t0         # n = n - elements_processed
    bnez    a0, Loop           # Repeat if n > 0
```

להראות דוגמה לקימפול עם וקטוריזציה

`~/.../lecture/.../code/simd.c and simd2.c`
using Intel icx compiler (`source setvars.sh` first)

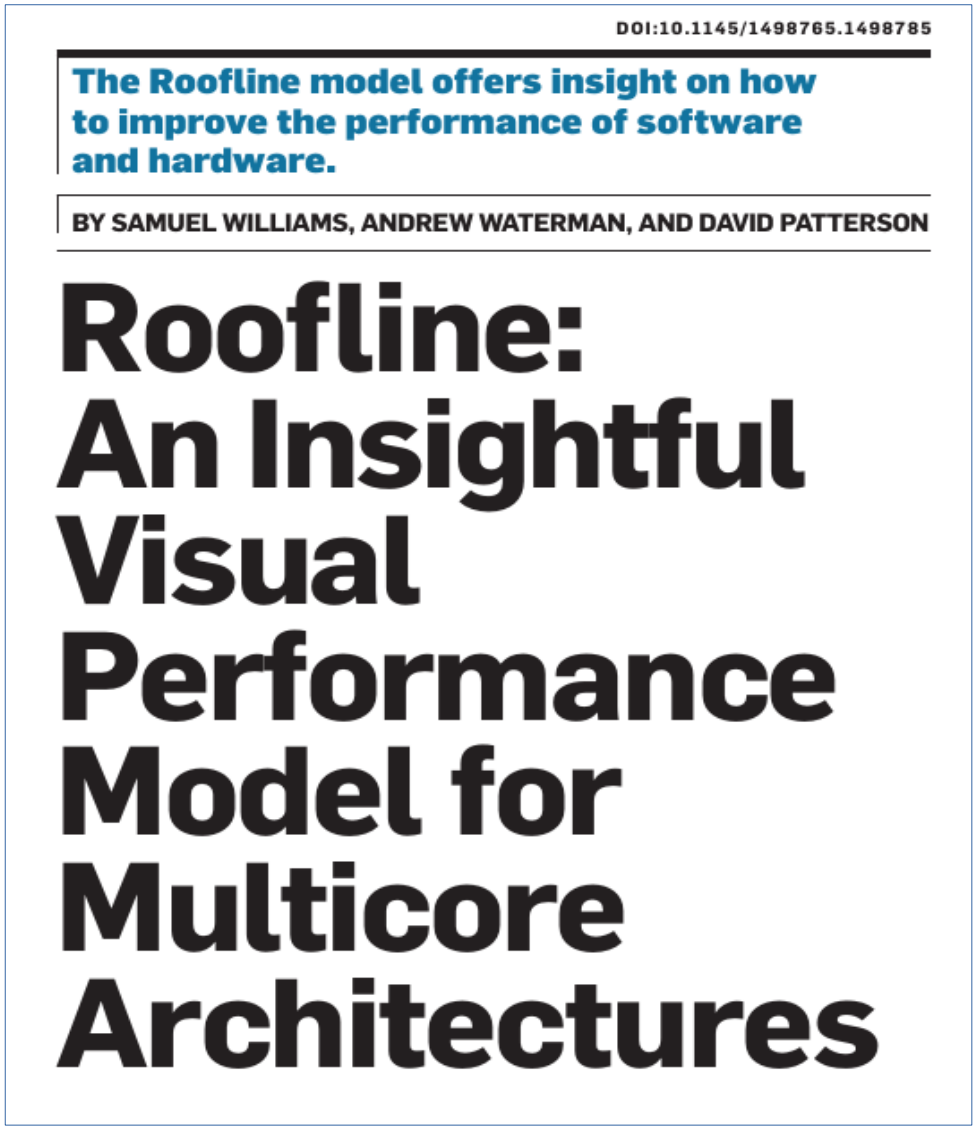
`simd` – a simple version
`simd2` – a vectorized version

use `runme.sh` to build both (including the assembly codes)

The Roof line model,

https://dl.acm.org/doi/pdf/10.1145/1498765.1498785

"The Roofline sets an upper bound on performance of a kernel depending on the kernel's operational intensity. If we think of operational intensity as a column that hits the roof, either it hits the flat part of the roof, meaning performance is compute-bound, or performance is ultimately memory-bound."



DOI:10.1145/1498765.1498785

The Roofline model offers insight on how to improve the performance of software and hardware.

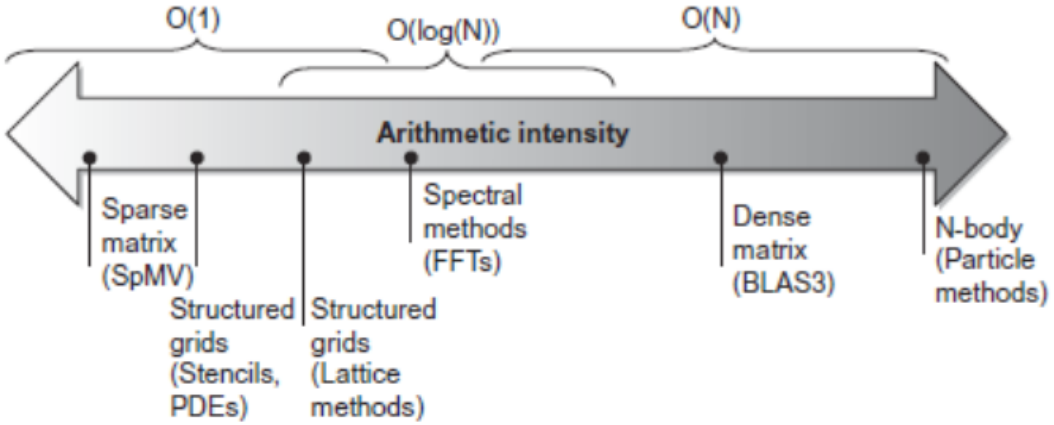
BY SAMUEL WILLIAMS, ANDREW WATERMAN, AND DAVID PATTERSON

Roofline: An Insightful Visual Performance Model for Multicore Architectures

Reference: Communications of the ACM | April 2009 | vol. 52 | no. 4

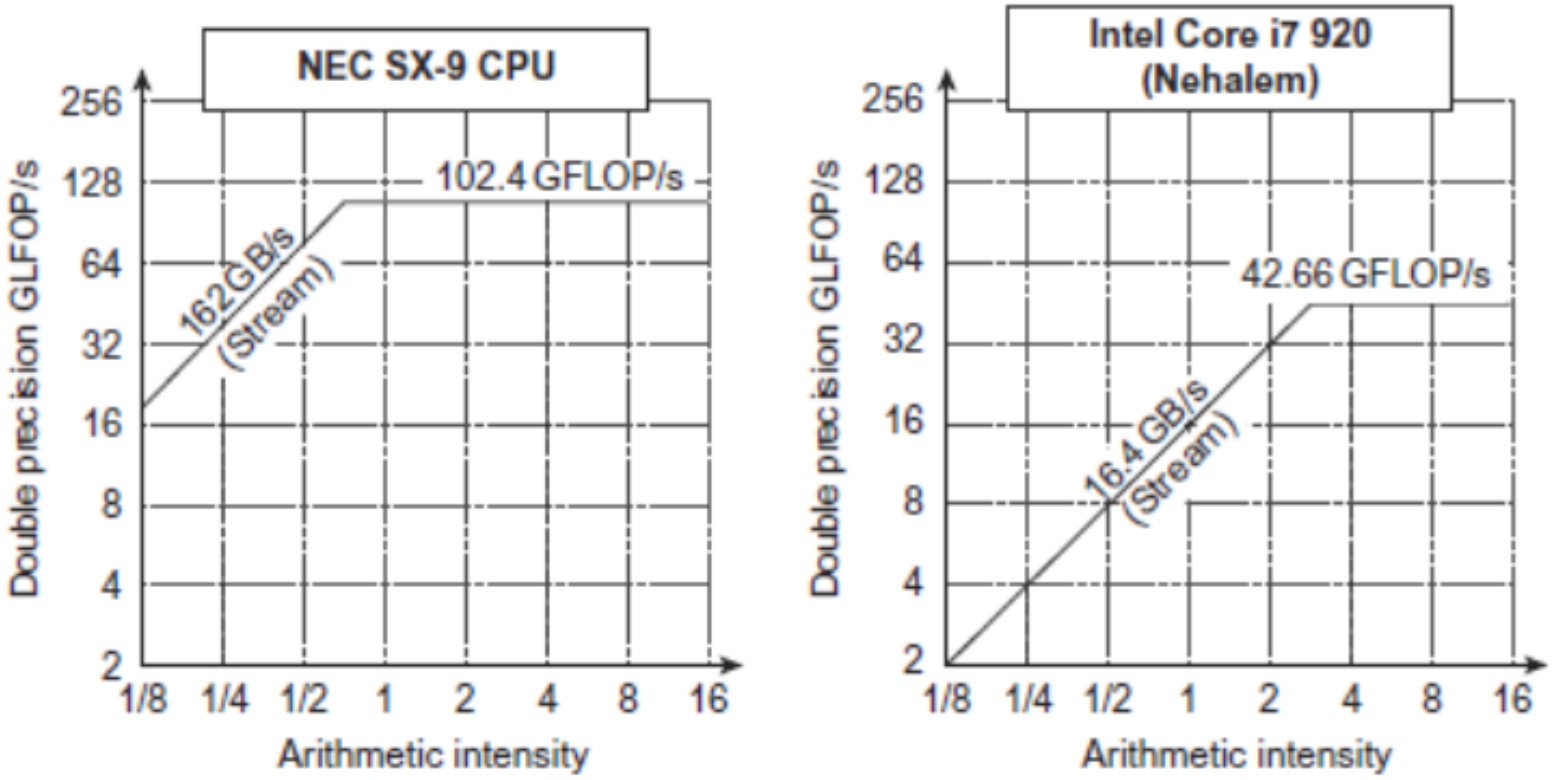
Roofline Performance Model

- 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



Examples

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



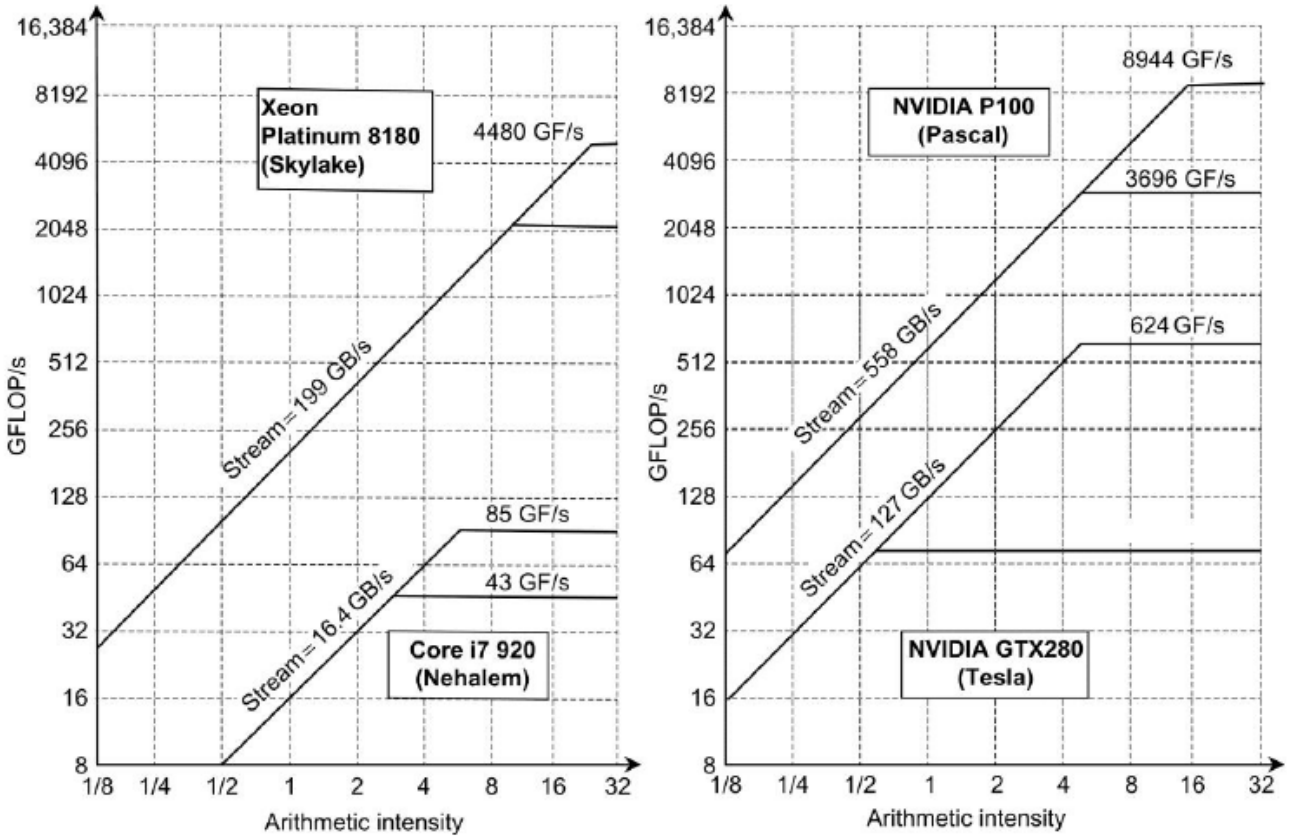
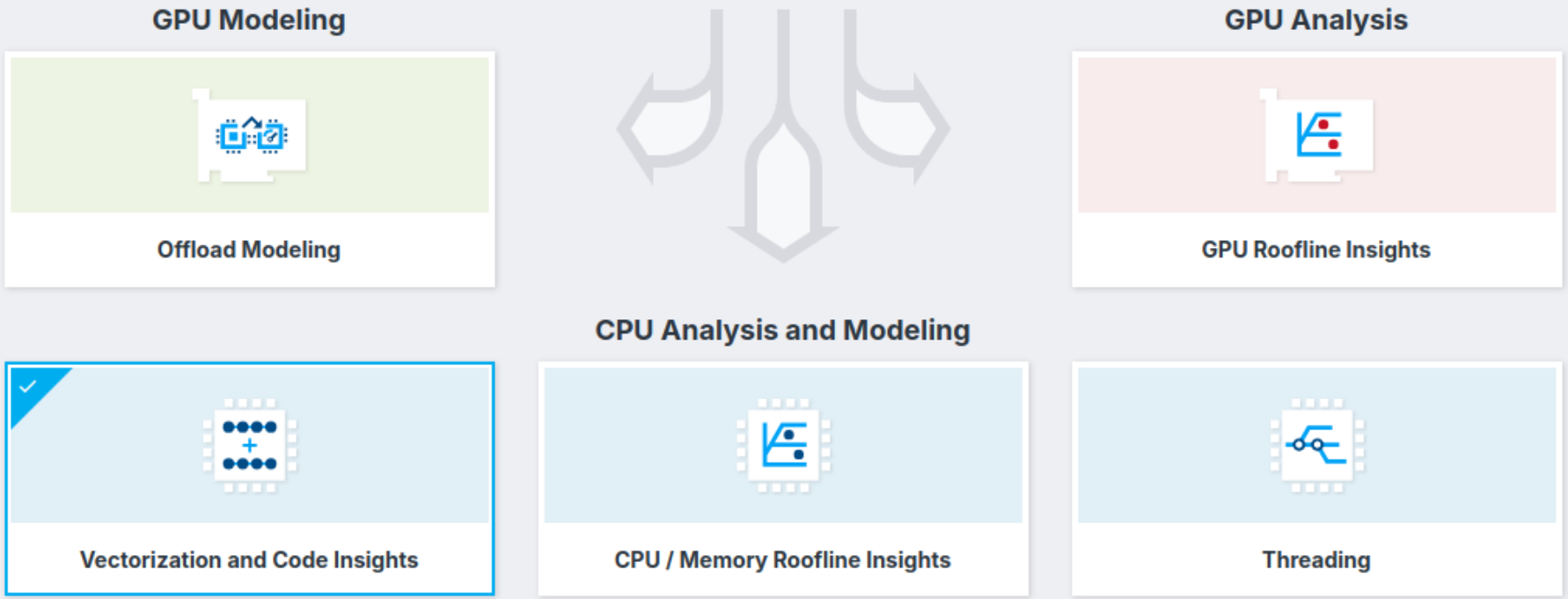


Figure 4.33 Roofline models of older and newer CPUs versus older and newer GPUs. The higher roofline for each computer is single-precision floating-point performance, and the lower one is double-precision performance.

CPU Roofline



tool: **Advisor** : **advixe-gui**

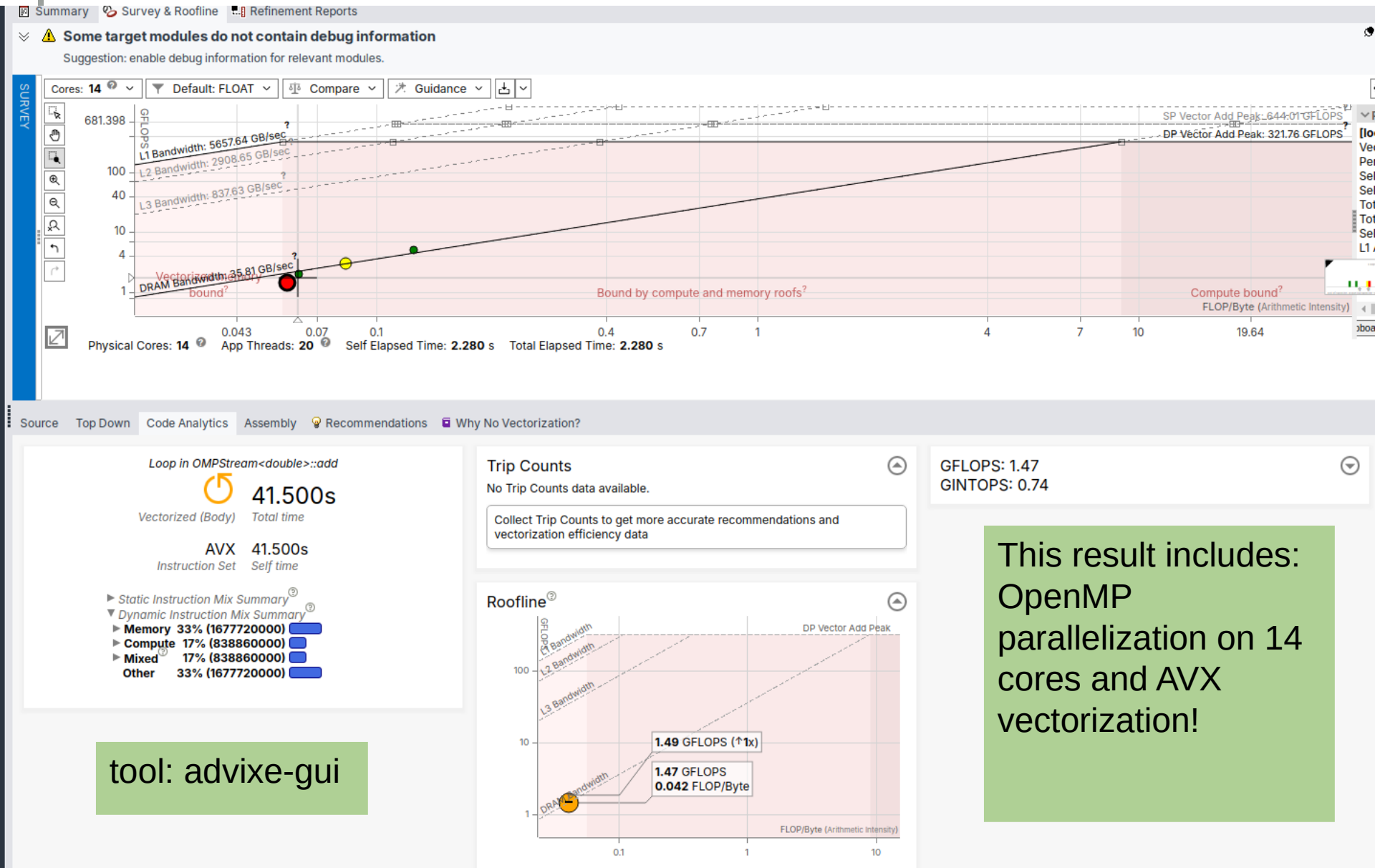
```
cmake .. \ -DMODEL=omp \ -DCMAKE_CXX_COMPILER=icpx \ -DCMAKE_C_COMPILER=icx \ -DCXX_EXTRA_FLAGS="-O3 -xHost -qopenmp"
```

CPU stream

Intel OneAPI 2024 with OpenMP and Vectorization on 14 i9 cores

```
BabelStream
Version: 5.0
Implementation: OpenMP
Running  Classic kernels 100 times in  Classic order
Number of elements: 33554432
Precision: double
Array size: 268.4 MB
Total size: 805.3 MB
Function    MB/s         Min (sec)   Max          Average
Copy        30049.964    0.01787     0.02117      0.01976
Mul         28506.778    0.01883     0.02250      0.02019
Add         30378.315    0.02651     0.03048      0.02836
Triad       30571.905    0.02634     0.03046      0.02825
Dot         35804.680    0.01499     0.01879      0.01611
```

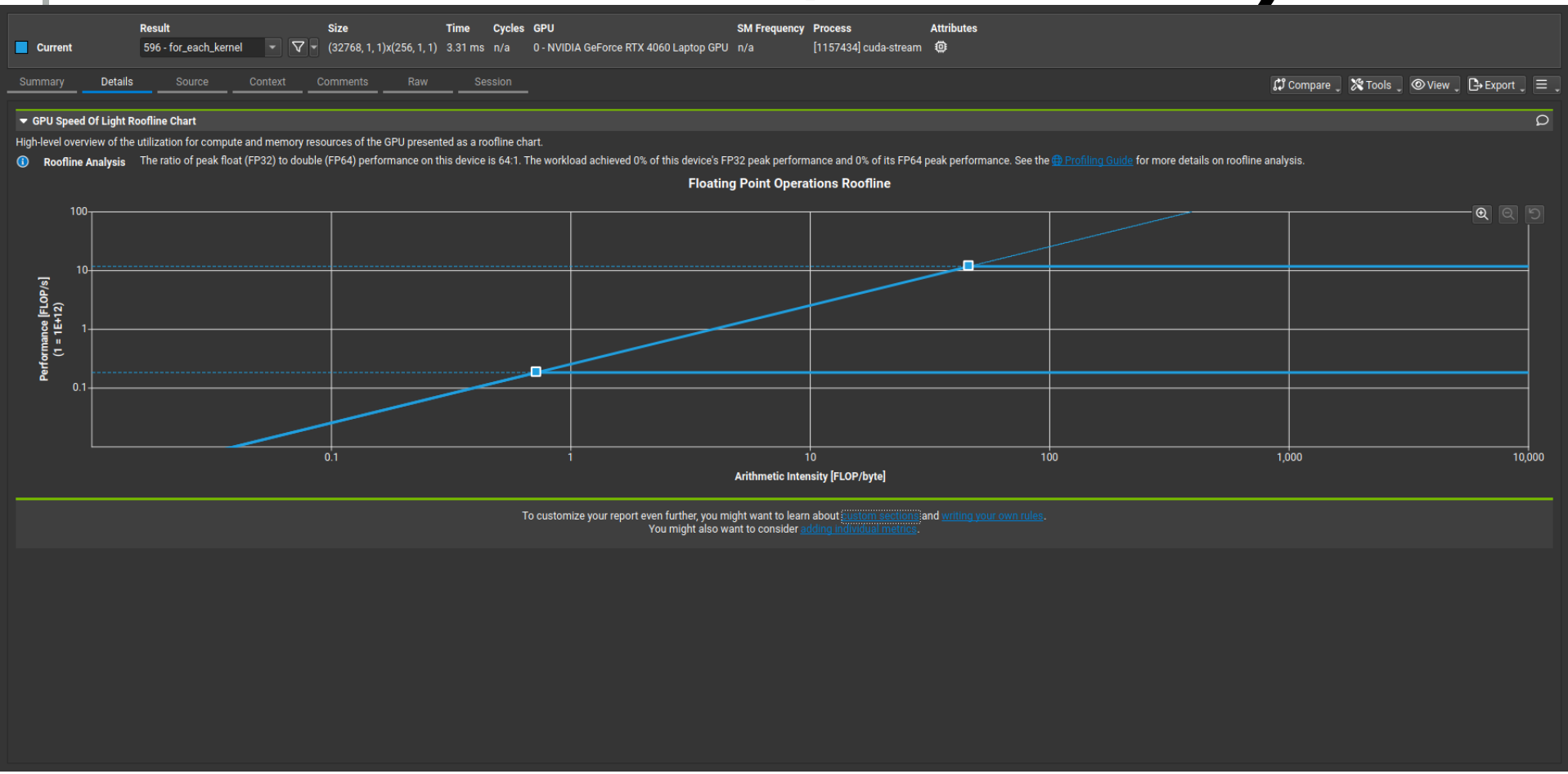
CPU Roofline (Intel OneAPI)



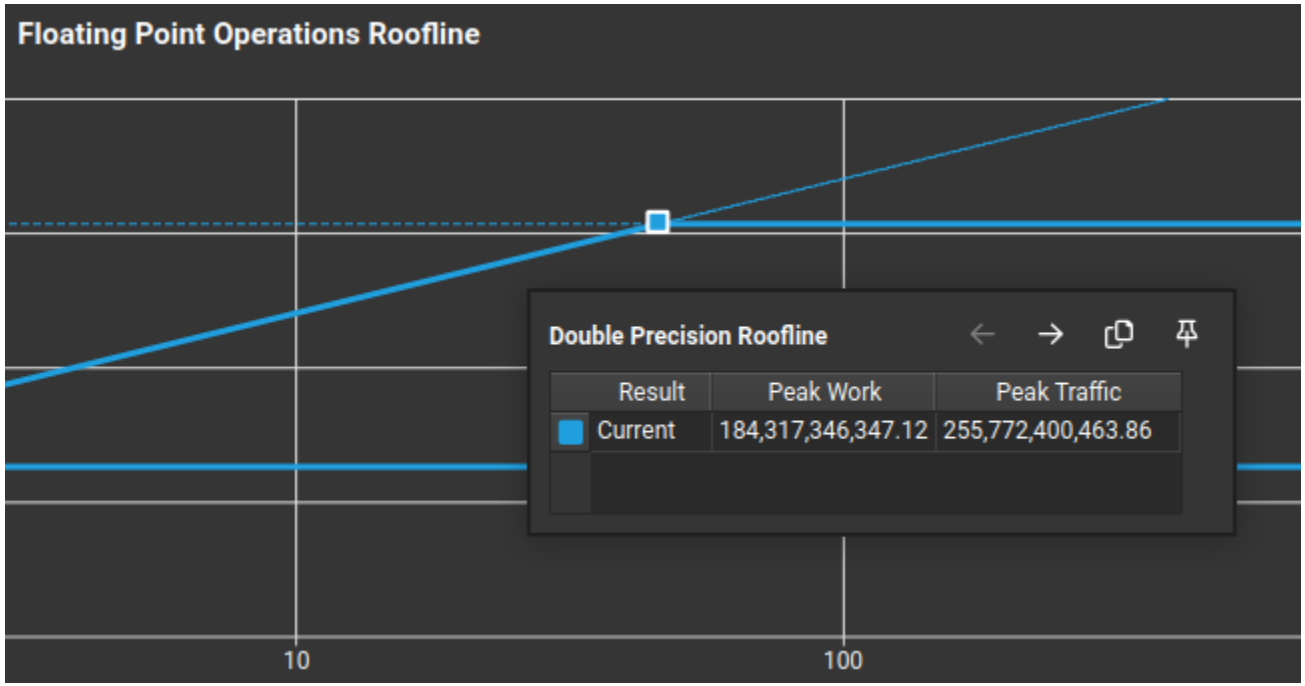
tool: advixe-gui

This result includes: OpenMP parallelization on 14 cores and AVX vectorization!

The stream benchmark (for Nvidia GPU, RTX4060)



RTX4060



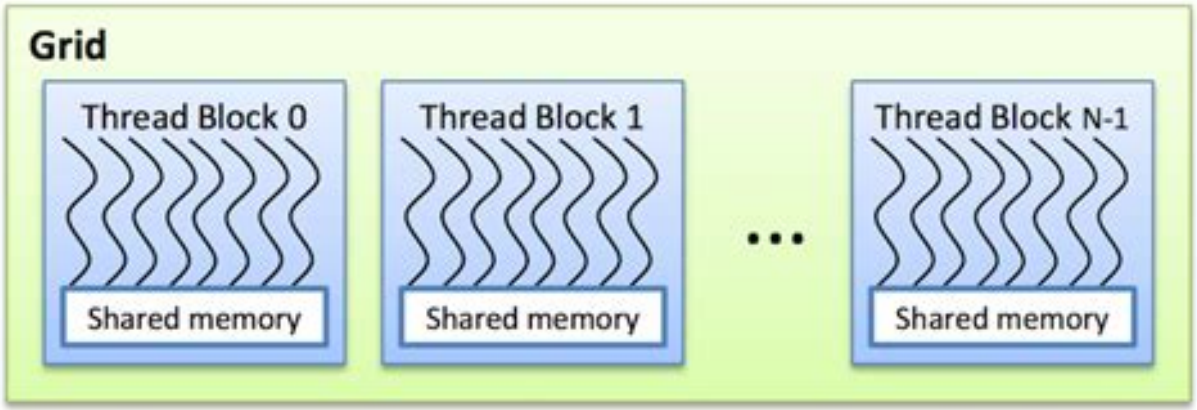
Peak compute ~184GFLOPS/s
Peak memory traffic ~255 GB/s

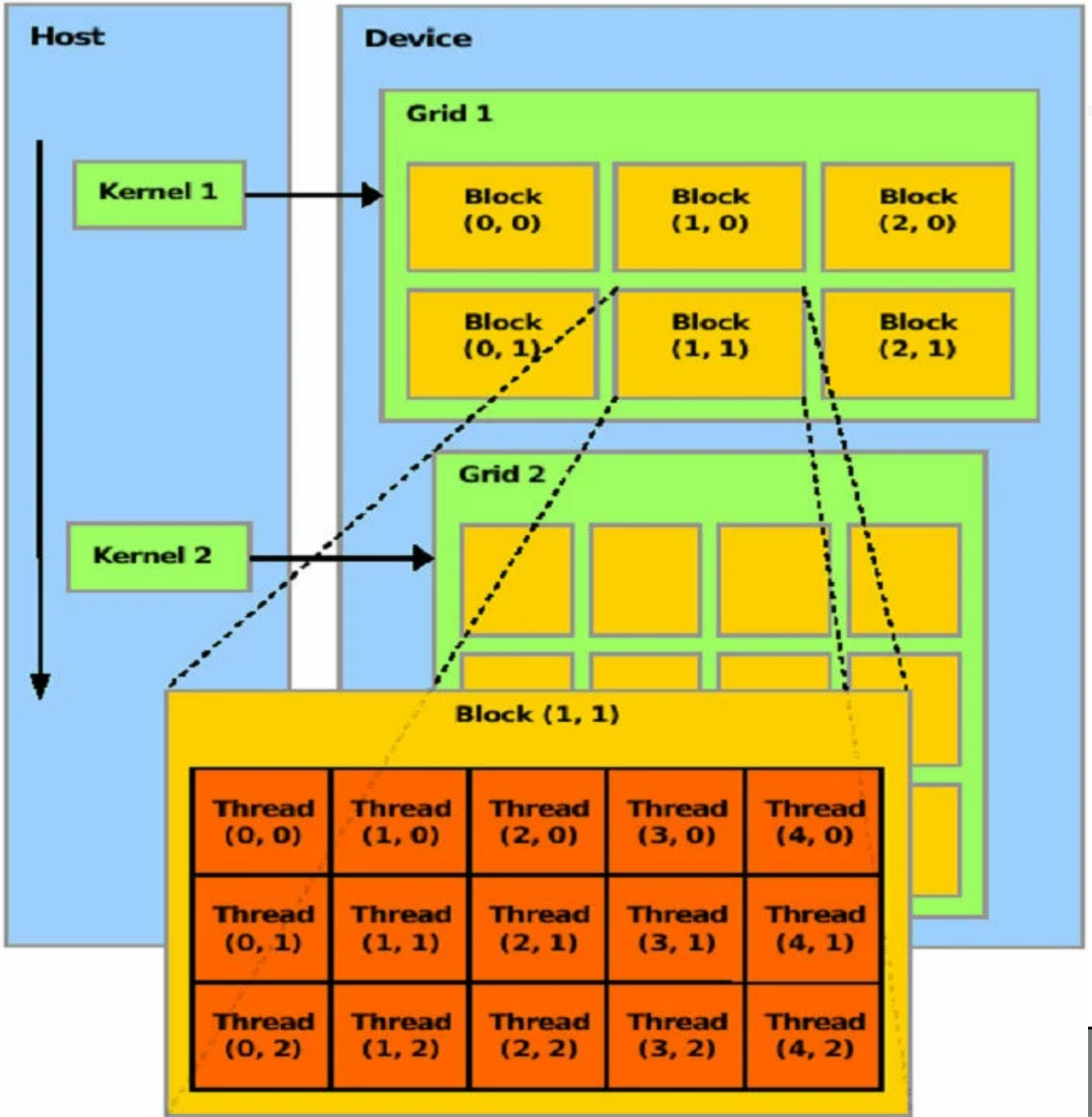
Graphical Processing Units

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

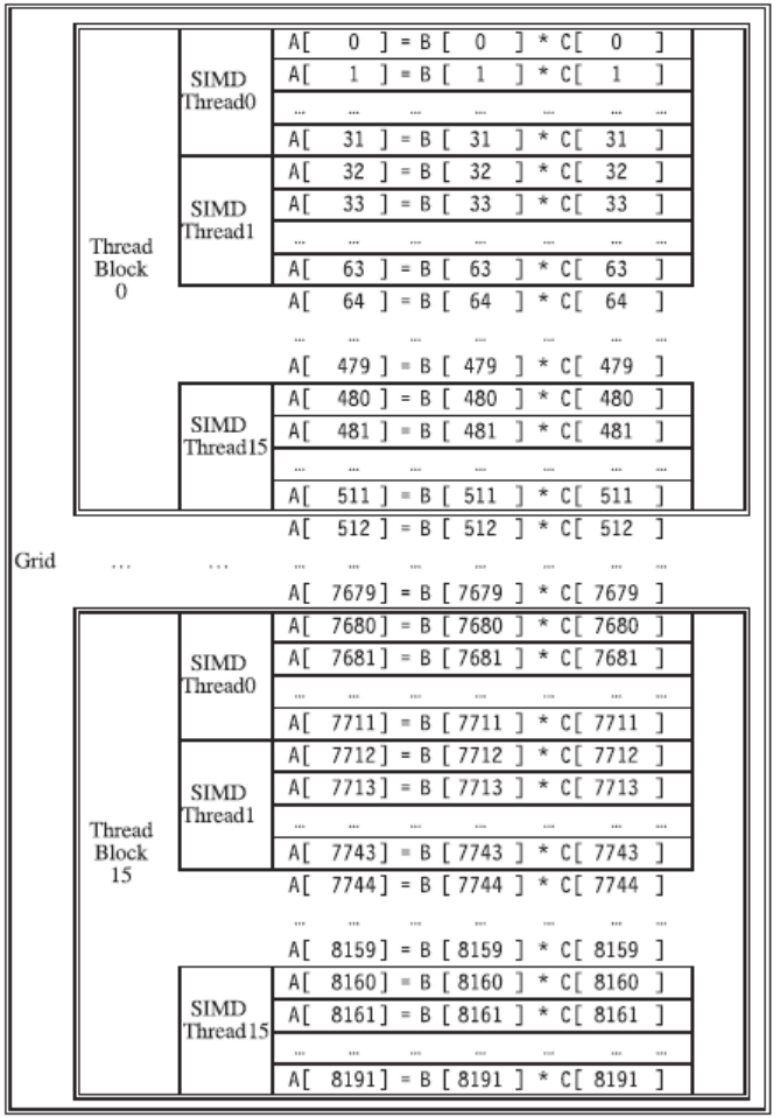
Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid
- GPU hardware handles thread management, not applications or OS




Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Example



Logical concepts mapped to HW

Software/Logical Concept	Hardware/Physical Concept (The Silicon)
Grid (The entire problem)	GPU (The entire chip)
Thread Block (A workgroup), can contain several warps	SM (Streaming Multiprocessor) (Executes 1 or more blocks)
Warp (32 threads executing together, i.e. lockstep = same instruction, same cycle)	Warp Scheduler (Issues instructions to 32 cores at once)
Thread (An individual data element)	CUDA Core / Execution Unit (Processes 1 thread's math)

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

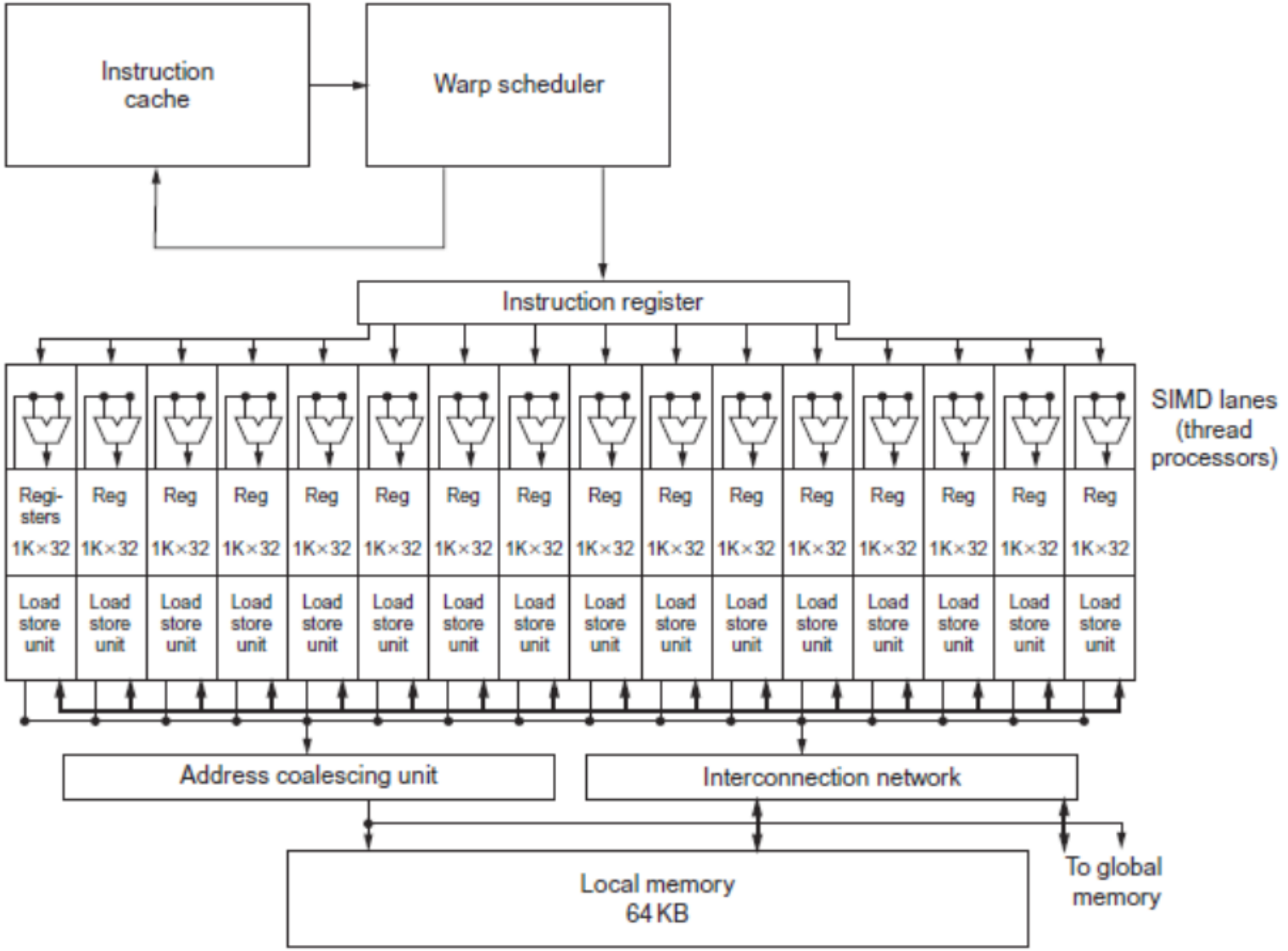
Example

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
  - 512 threads per block
- SIMD instruction executes 32 elements at a time (1 warp)
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Current-generation GPUs have 7-15 multithreaded SIMD processors

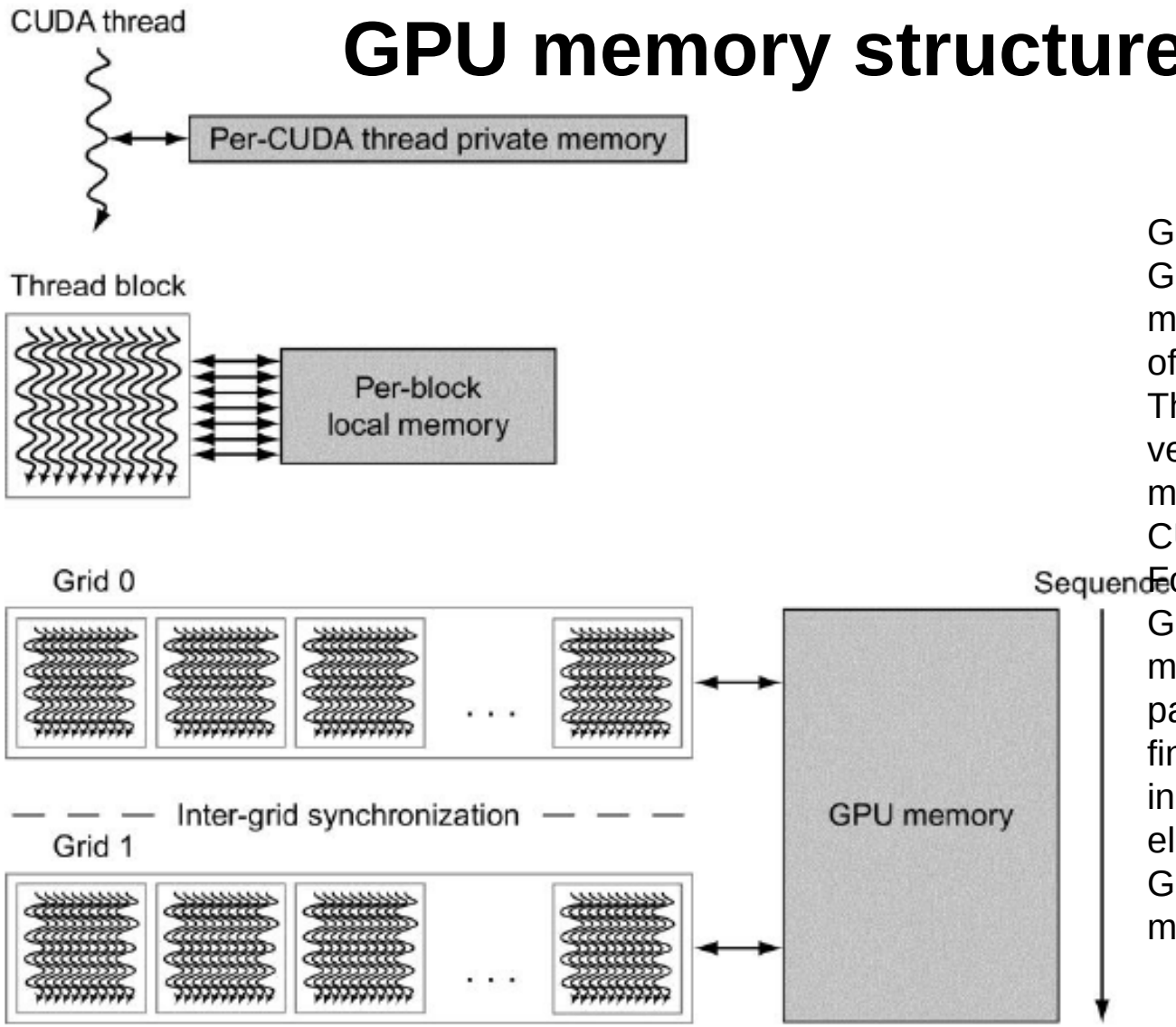
Terminology

- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or “warp”
  - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - Each warp has its own PC
  - 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:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

GPU Organization



GPU memory structures



GPU memory is shared by all Grids (vectorized loops), local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread.

For completeness sake, the GPU can also access CPU memory via the PCIe bus. This path is commonly used for a final result when its address is in host memory. This option eliminates a final copy from the GPU memory to the host memory.

```
$ nvidia-smi
Fri Jun 26 12:20:00 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 595.71.05              Driver Version: 595.71.05      CUDA Version: 13.2     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4060 ...    Off |   00000000:01:00.0  On |                  N/A |
| N/A   51C    P8              4W /   80W |     862MiB /   8188MiB |     18%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            6111      G   /usr/lib/xorg/Xorg                        65MiB |
|    0   N/A  N/A         1005340      C   ...local/lib/ollama/llama-server         768MiB |
+-----------------------------------------------------------------------------------------+
```

Compute Capability

```
$ nvidia-smi --query-gpu=name,compute_cap --format=csv
name, compute_cap
NVIDIA GeForce RTX 4060 Laptop GPU, 8.9
```

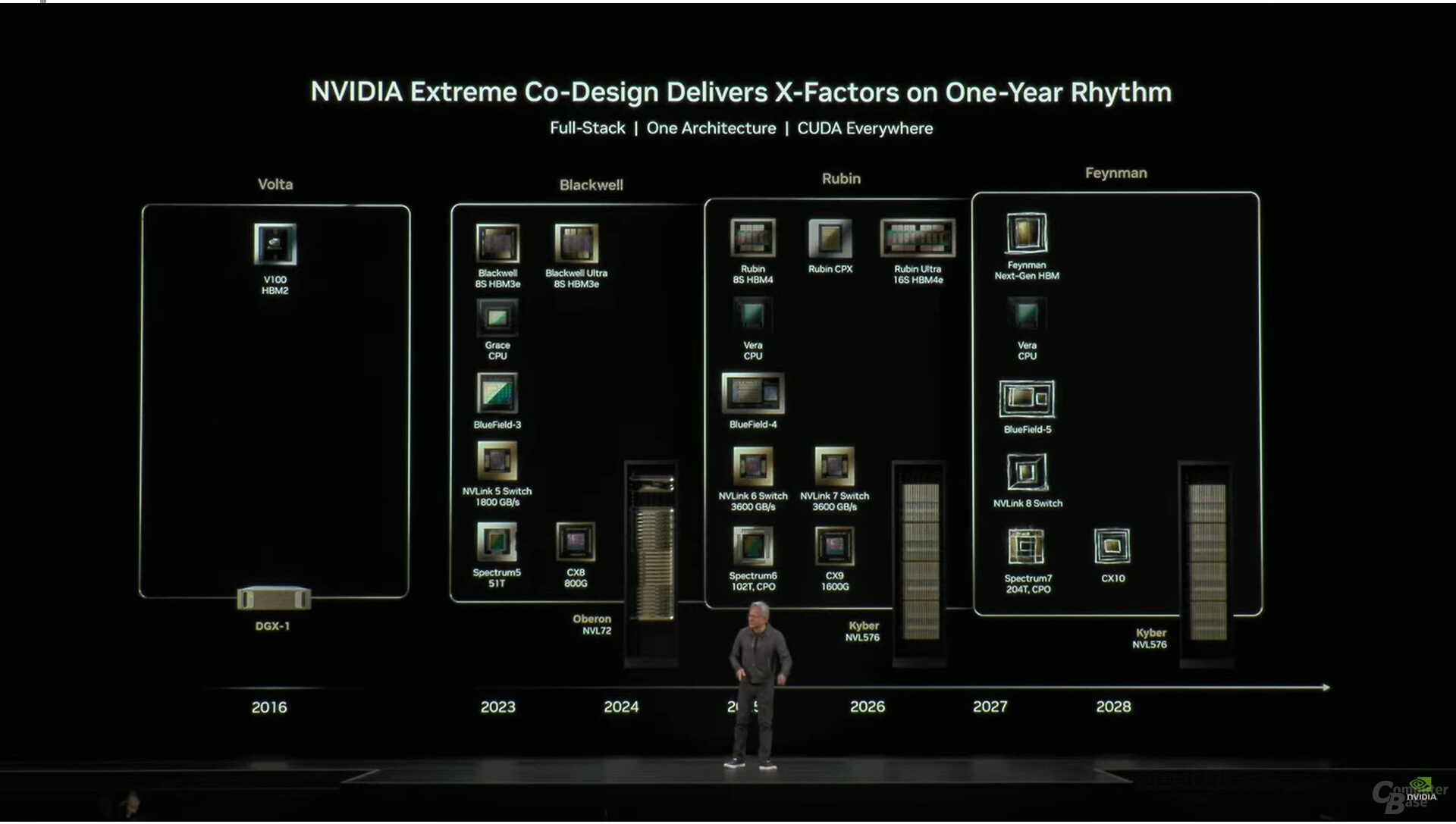
GPU	Architecture	Compute Capability
RTX 4060	**Ada Lovelace (AD107)**	**8.9** my laptop
RTX 4060 Ti	Ada Lovelace (AD106)	8.9
RTX 4090	Ada Lovelace (AD102)	8.9
RTX 3090/4090	Ampere	8.6
RTX 2080	Turing	7.5

```
nvcc -arch=sm_89  # For RTX 4060 (Ada Lovelace)
```

Pascal Multithreaded SIMD Proc.

לכל אחד מ-64 נתיבי ה-SIMD (ליבות) יש יחידת נקודה צפה בצינור, יחידת מספר שלם בצינור, לוגיקה מסוימת לשליחת הוראות ואופרנדים ליחידות אלו, ותור להחזקת תוצאות. 64 נתיבי ה-SIMD מקיימים אינטראקציה עם 32 יחידות ALU (DP) בעלות דיוק כפול המבצעות חשבון נקודה צפה של 64 סיביות, 16 יחידות טעינה-אחסון (LD/ST) ו-16 יחידות פונקציה מיוחדת (SFU) המחשבות פונקציות כגון שורשים ריבועיים, גומלין, סינוסים וקוסינוסים.

סדרות של מאיצים מתוצרת אנבידיה



NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
    - opcode. type d,a,b,c;
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
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])
```

להסברים עבור קוד האסמבלי של CUDA ראו:
https://docs.nvidia.com/cuda/inline-ptx-assembly/index.html

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

Example

```
if (X[i] != 0)
            X[i] = X[i] – Y[i];
else X[i] = Z[i];

ld.global.f64       RD0, [X+R8]       ; RD0 = X[i]
setp.neq.s32        P1, RD0, #0       ; P1 is predicate register 1
@!P1, bra ELSE1, *Push                ; Push old mask, set new mask bits
                                      ; if P1 false, go to ELSE1
ld.global.f64       RD2, [Y+R8]       ; RD2 = Y[i]
sub.f64   RD0, RD0, RD2               ; Difference in RD0
st.global.f64       [X+R8], RD0       ; X[i] = RD0
@P1, bra  ENDIF1, *Comp               ; complement mask bits
                                      ; if P1 true, go to ENDIF1
ELSE1:    ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
          st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1:   <next instruction>, *Pop    ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Pascal Architecture Innovations

- Each SIMD processor has
  - Two or four SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Two threads of SIMD instructions are scheduled every two clock cycles
- Fast single-, double-, and half-precision
- High Bandwith Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

Chime = the latency of a GPU execution unit

Vector Architectures vs GPUs

- SIMD processor analogous to vector processor, both have MIMD
- Registers
  - RV64V register file holds entire vectors
  - GPU distributes vectors across the registers of SIMD lanes
  - RV64 has 32 vector registers of 32 elements (1024)
  - GPU has 256 registers with 32 elements each (8K)
  - RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles
  - SIMD processor chime is 2 to 4 cycles
  - GPU vectorized loop is grid
  - All GPU loads are gather instructions and all GPU stores are scatter instructions

SIMD Architectures vs GPUs

- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

- Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4:

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

- Example 5:

```
for (i=1;i<100;i=i+1) {
        Y[i] = Y[i-1] + Y[i];
}
```

Reductions

- Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note: assumes associativity!

Vector add demo

folder: ~/science/Teaching/CPU/lectures/13/code/simd/CUDA

מעודכן לגרסת 26.5

:compiling #

~~opt/nvidia/hpc_sdk/Linux_x86_64/25.5/compilers/bin/nvcc -Wno-deprecated-/ gpu-targets -o VecAdd ./VecAdd.cu~~

```
opt/nvidia/hpc_sdk/Linux_x86_64/26.5/compilers/bin/nvcc -Wno-deprecated-/
gpu-targets -o VecAdd ./VecAdd.cu
```

profiling #

~~sudo /opt/nvidia/hpc_sdk/Linux_x86_64/25.5/profilers/Nsight_Compute/ncu --section SpeedOfLight_RooflineChart -k vecAdd -o vadd_roofline ./VecAdd~~

```
sudo /opt/nvidia/hpc_sdk/Linux_x86_64/26.5/profilers/13.2/Nsight_Compute/
ncu --section SpeedOfLight_RooflineChart -k vecAdd -o vadd_roofline
./VecAdd
```

:visualization #

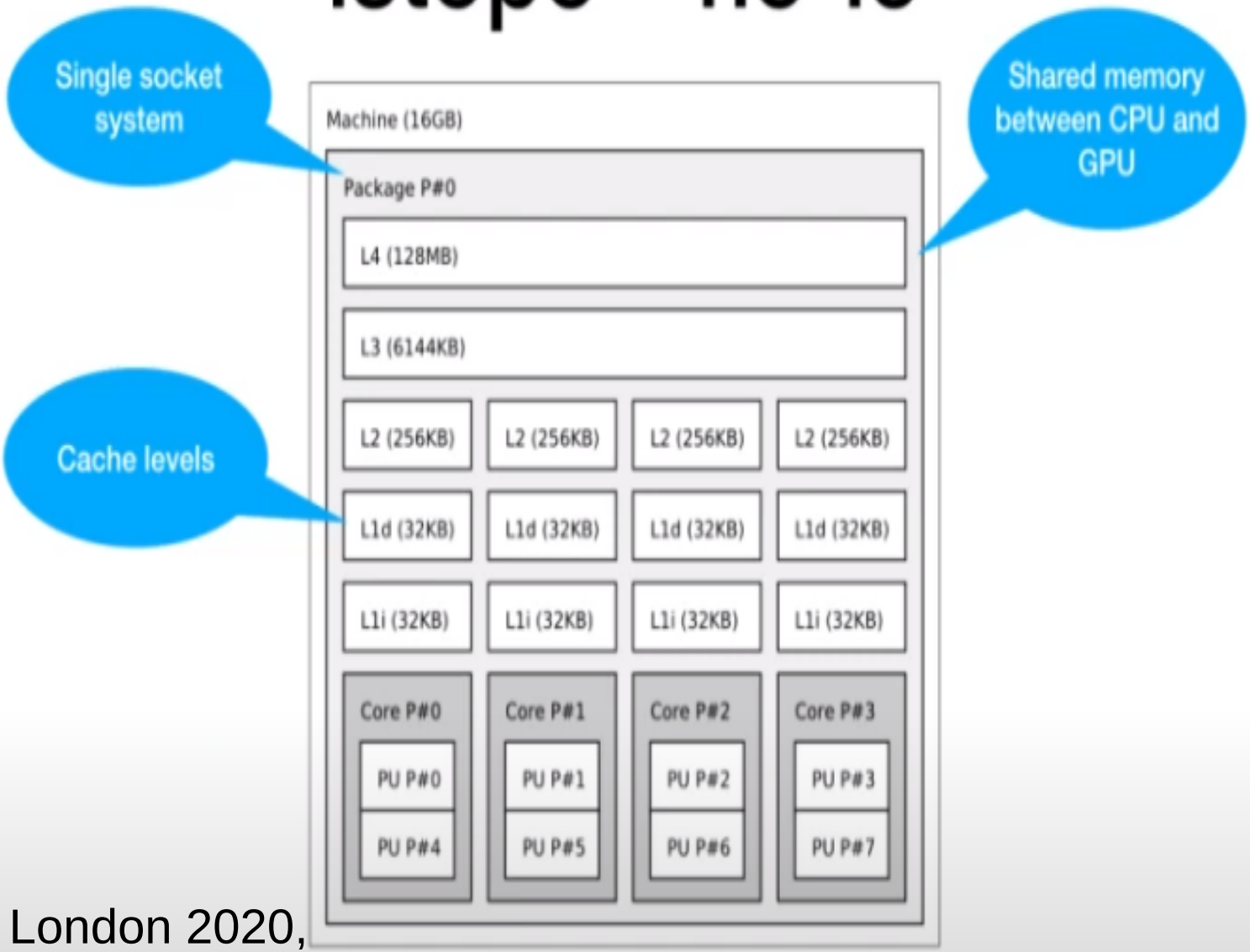
~~opt/nvidia/hpc_sdk/Linux_x86_64/25.5/profilers/Nsight_Compute/ncu-ui/~~

```
opt/nvidia/hpc_sdk/Linux_x86_64/26.5/profilers/13.2/Nsight_Compute/ncu-ui/
```

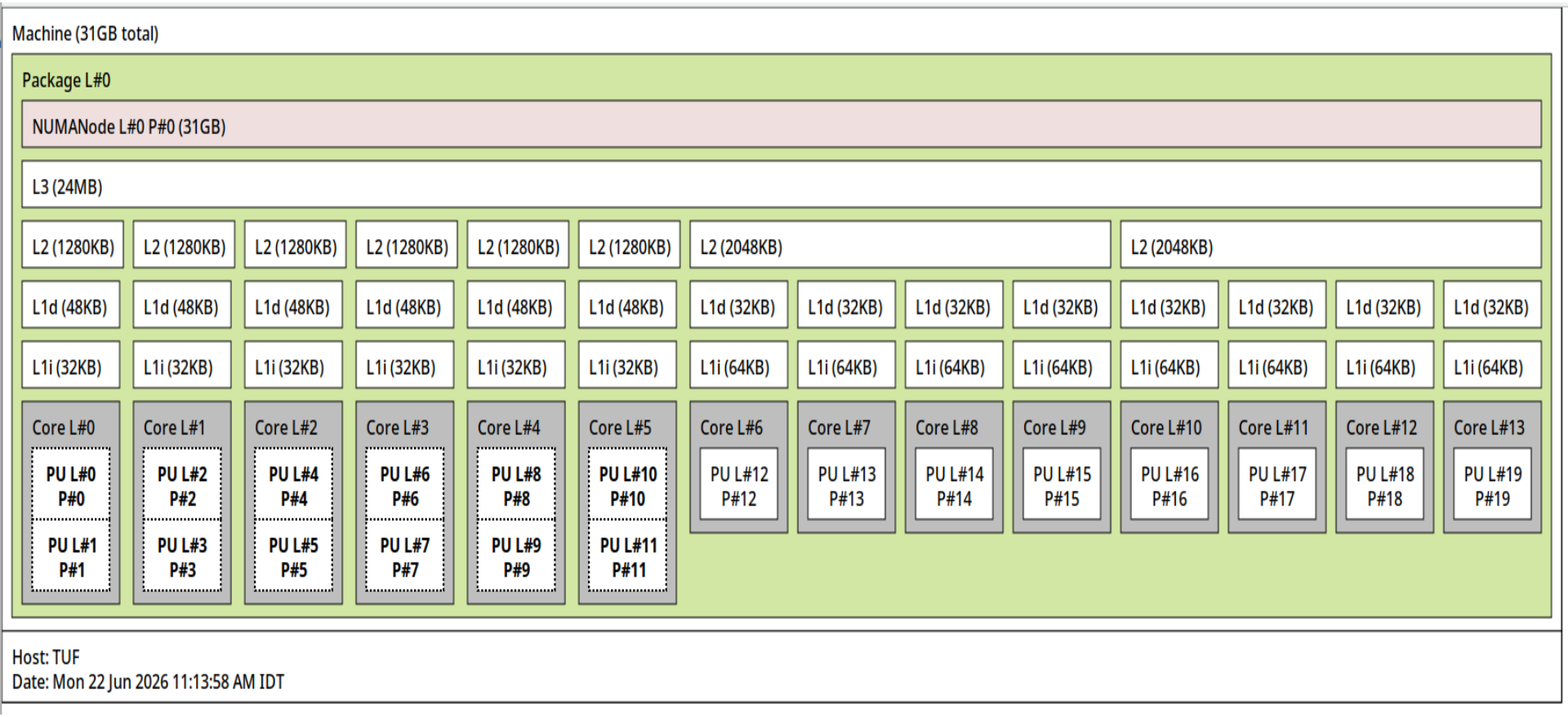
עוד כמה דברים לסיכום

QCon LONDON by InfoQ

lstopo --no-io



Source: Qcon London 2020, https://www.youtube.com/watch?v=rgImJ6Xyj1c


Machine (31GB total)
Package L#0
NUMANode L#0 P#0 (31GB)
L3 (24MB)
L2 (1280KB)
L2 (1280KB)
L2 (1280KB)
L2 (1280KB)
L2 (1280KB)
L2 (1280KB)
L2 (2048KB)
L2 (2048KB)
L1d (48KB)
L1d (48KB)
L1d (48KB)
L1d (48KB)
L1d (48KB)
L1d (48KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
L1i (64KB)
Core L#0
PU L#0 P#0
PU L#1 P#1
Core L#1
PU L#2 P#2
PU L#3 P#3
Core L#2
PU L#4 P#4
PU L#5 P#5
Core L#3
PU L#6 P#6
PU L#7 P#7
Core L#4
PU L#8 P#8
PU L#9 P#9
Core L#5
PU L#10 P#10
PU L#11 P#11
Core L#6
PU L#12 P#12
Core L#7
PU L#13 P#13
Core L#8
PU L#14 P#14
Core L#9
PU L#15 P#15
Core L#10
PU L#16 P#16
Core L#11
PU L#17 P#17
Core L#12
PU L#18 P#18
Core L#13
PU L#19 P#19
Host: TUF
Date: Mon 22 Jun 2026 11:13:58 AM IDT

a test code

```
telzur@TUF:~/science/Teaching/CPU/lectures/13/code/
perf_stat_base64$ cat ./hello.c
#include <stdio.h>
double x,y,z;
int main() {
        int i;
       long MAX = 10000000000;
        for (i=0;i<MAX;i++)
//              printf("Hello World!\n");
                x = i;
                y = x * x * x * x;
                z = 4.0*(x + y)/(x+1);
       return 0;
}
```

On my (TUF) laptop

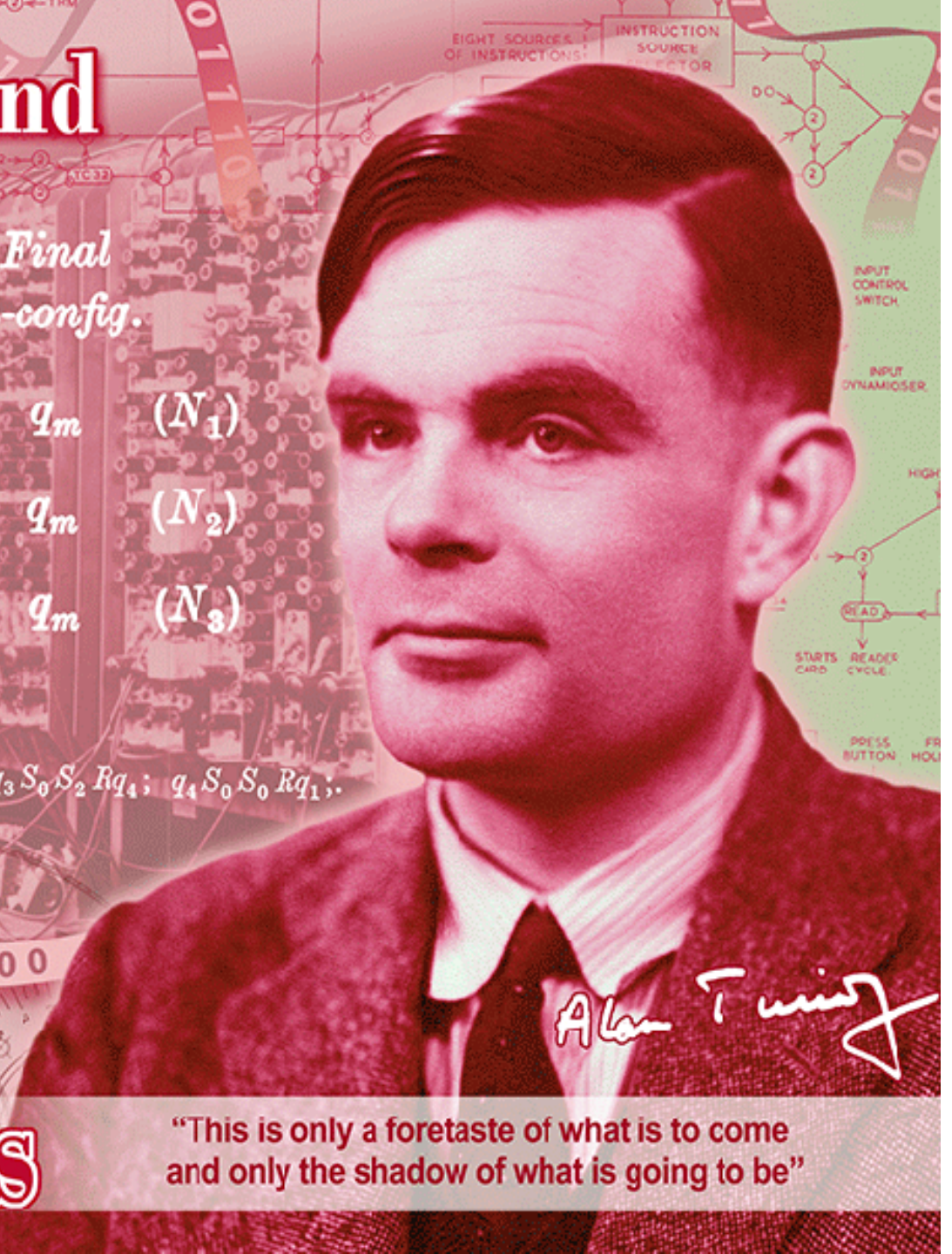
```
telzur@TUF:~/science/Teaching/CPU/lectures/13/code/perf_stat_base64$ perf stat base64 <(echo hello)
aGVsbG8K

 Performance counter stats for 'base64 /dev/fd/63':

            330,523      task-clock                       #    0.394 CPUs utilized
                  0      context-switches                 #    0.000 /sec
                  0      cpu-migrations                   #    0.000 /sec
                 73      page-faults                      #  220.862 K/sec
            982,811      cpu_atom/instructions/           #    0.79  insn per cycle
     <not counted>      cpu_core/instructions/
(0.00%)
          1,247,441      cpu_atom/cycles/                 #    3.774 GHz
     <not counted>      cpu_core/cycles/
(0.00%)
            182,693      cpu_atom/branches/               #  552.739 M/sec
     <not counted>      cpu_core/branches/
(0.00%)
              8,427      cpu_atom/branch-misses/          #    4.61% of all branches
     <not counted>      cpu_core/branch-misses/
(0.00%)
 #     17.4 %  tma_bad_speculation
                                                   #     20.3 %  tma_retiring
 #     27.9 %  tma_backend_bound
                                                   #     34.4 %  tma_frontend_bound

       0.000838364 seconds time elapsed

       0.000828000 seconds user
       0.000000000 seconds sys
```

Bank of England

m-config.	Symbol	Operations	Final m-config.	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

$q_1 S_0 S_1 Rq_2;\ q_2 S_0 S_0 Rq_3;\ q_3 S_0 S_2 Rq_4;\ q_4 S_0 S_0 Rq_1;.$

10101111111100101100111000



Alan Turing

Fifty Pounds

"This is only a foretaste of what is to come
and only the shadow of what is going to be"

המלצות לחיים...

- ללמוד היטב למבחן
- לעשות מנוי על כתב-עת מקצועי
- להצטרף לאגודה מקצועית כגון:

  IEEE Computer Society
- לקרוא כל הזמן חומרים חדשים
- בהמשך לצאת לכנסים

בהצלחה בבחינות וקיץ נעים !!!

עד כאן מצגת זו