# Out of Order Execution

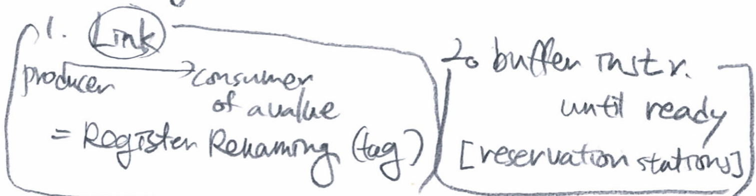
Idea ⇒ independent first, dependent last → Rest areas for dep. instr

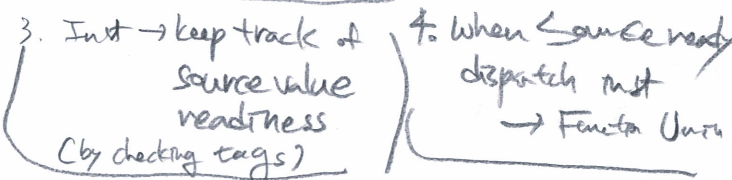
- Monitor the source "values" ⇒ Reservation stations
  ↓ in the resting area
- when all source "values" are available
  ⇒ "fire" the instruction

∴ Instr. dispatched in dataflow order

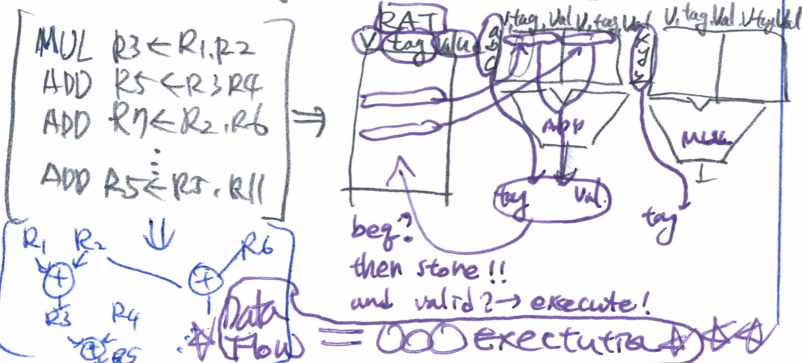
In order dispatch + precise exceptions
vs Out order dispatch + precise exception.

## ∀ Enabling OOO Execution



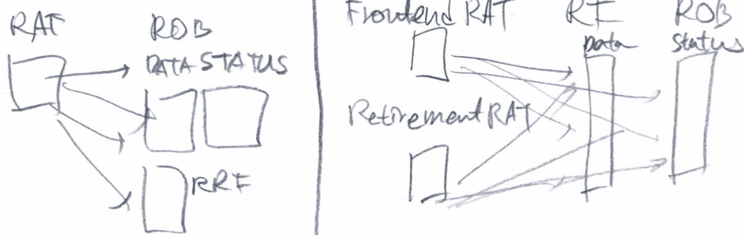

- Tomasulo's Algorithm
- Two Hump in a Modern Pipeline




# Lecture 17: Out of Order + Dataflow + Superscalar Execution.

* Most modern OoO Execution w/ precise execution
↳ ① Reorder buffer to support in-order retirement
② A single physical register file to store all register value
③ 2 register maps
- Future/Front reg. map → used for renaming
- Architectural reg. map → used for maintaining precise state.

⇒ OOO Ex = "Restricted Dataflow" (limited to instruction window)

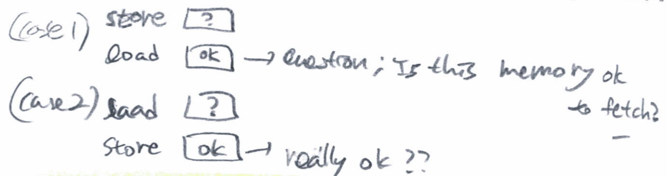



Q. Memory Dependence Handling?

⇒ Mem addr is not known until a load/store executes

↳ renaming? → difficult

- Determining dep. → after (partial) execution
- When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine




⇒ "memory disambiguation" or "unknown address problem"

Solution: ① Stall

② Aggressive; Just do it! Assume load is independent

③ Intelligent; Predict if the load is dependent on the/any unknown address store

How to detect ?

Option① Wait until all previous stores committed

Option② keep a list of pending stores in a "store buffer" & check whether "load address" matches a prev. "store addr"

"Store buffer" = list of "store"s that are pending in the machine

ld addr 1000

option② How schedule ? ⇒ Option ① Assume dependent → simple & slow
② " independent → simple & need recovery
③ Predict → more accurate, still need recovery.

---

* Data Forwarding Between Store & Load

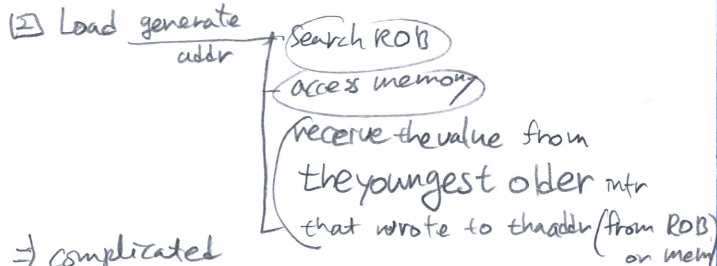
↳ Modern processors use LQ (load queue) & SQ (store queue)

→ age-based comparison !!

• Out of Order Completion of Memops

Ⅰ Store execution finish → ROB

Ⅱ Load generate addr




⇒ complicated search logic ⇒ Content addressable memory.

• Store-Load Forwarding Complexity

↳ option① CAM

option② Range Search

option③ Age-based Search

also④ Load data from [ SQ / Mem/cache

MDC⑤

# Other Approaches to Concurrency (on Instruction Level Parallelism)

① Superscalar execution ② VLIW ③ Fine-grain multithreading

④ SIMD Processing (Vector & array processors, GPUs)

⑤ Decoupled Access Execute

⑥ Systolic Arrays

## ☆ Superscalar Execution

Idea: Fetch/Decode/Execute/Retire multiple instr per cycle

• N-wide superscalar → N instr per cycle.

◦ In-Order superscalar processor

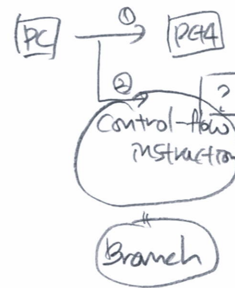
- Copies of datapath.
- Dependency make it tricky.

⇒ { Adv: high IPC
  Disadv: Dependency, more hardware.

---

# Lecture 18. Branch Prediction

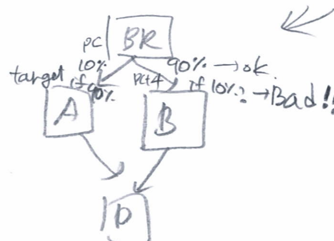
• Control Dependence Handling




| Branch types | Direction at fetch time | # of possible next pc | When next fetch resolved |
|---|---|---|---|
| Conditional | unknown | 2 | E |
| Unconditional | Always taken | 1 | D |
| Call | " | 1 | D |
| Return | " | many | E |
| Indirect | " | many | E |

## ☆ Branch Prediction ☆

if (x == 10) { A } else { B } ~ { D }




if (pointer != NULL) { A } else { B }

⇑ more probable!

→ programmer needs to try maximize probability.

◦ Guessing Next PC+4

Idea: Get rid of control flow instr.

→ ① Predicate combining

② Predicated execution.

◦ Branch Prediction.

↳ fetched instr == Branch ?

↳ Conditional Branch ?

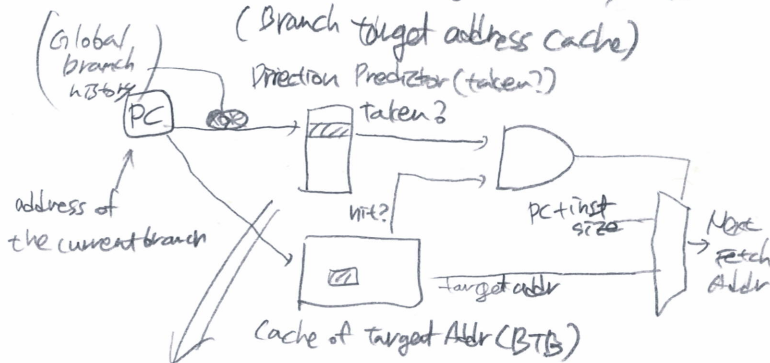
↳ Branch target addr. ?

→ store the target addr. (if taken)

from prev. instance & access it with the PC.

◦ Called Branch Target Buffer, BTB

(Branch target address cache)




So, How can we predict the branch? direction..

- Compile time (static) — Always Not Taken
  - Always Taken
  - BTFN (Back Taken, Foward Not taken)
  - Profile based
  - Program analysis based
  - pragma → if(likely(x)) / if(unlikely(x))
- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two bit counter based precition
  - Two-level prediction
  - Hybrid (Dynamic + Static)
  - Advanced algorithms (perceptrons)

Intel Pentium Pro Branch Predictor

⇒ Two level global branch predictor

- 4-bit global history register
- Multiple pattern history tables (of 2bit counters)

# Lecture 19: Branch Prediction + VLIW + Fine-Grained Multithreading

MOOC ⑥

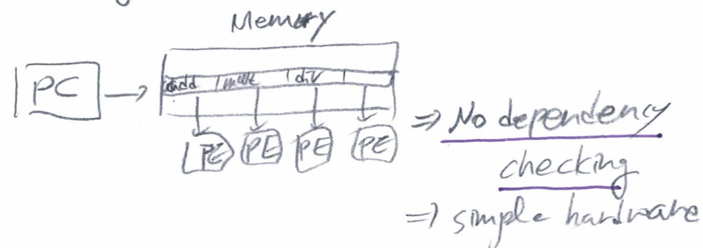
Other control dependence handling

- → Stall
- → Branch Prediction
- → Branch Delay Slot
- → Fine-grained multithreading
- → predicated execution
- → multipath execution

VLIW very large instruction word

⇔ superscalar; multiple instructions & check dependencies between them

Software packs independent instructions in a larger "instruction bundle"




⇒ No dependency checking

⇒ simple hardware

(But complex compiler)

Q. what about variable latency operation?

- VLIW philosophy

⇒ ✓RISC, ✓Intel IA-64, ✓Superblock ✓Impact.

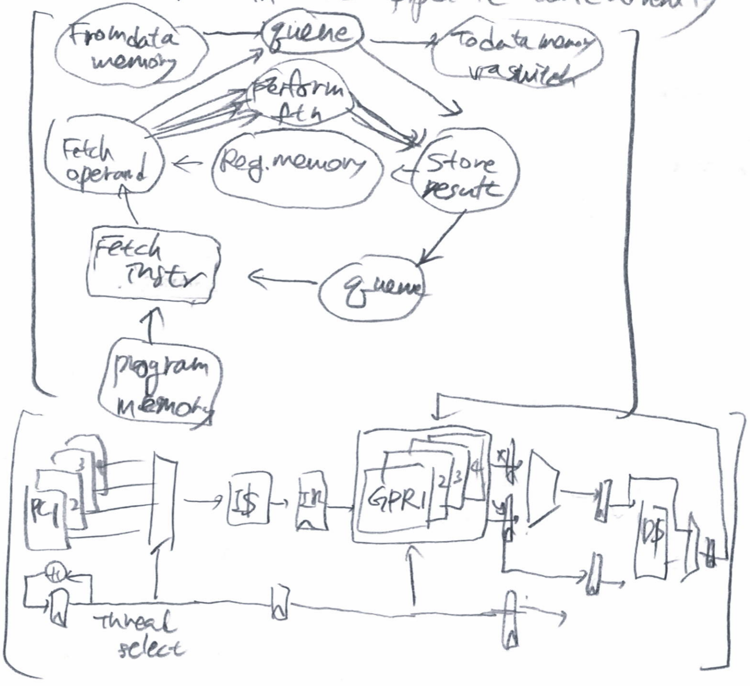
- Trade-off — Adv: No need for
  - dynamic scheduling hardware
  - dependency checking
  - instr. align

Disadv: Compiler more complex!!

## [Fine-Grained Multithreading]

idea ① Fetch engin fetchs from a diff. thread with hardware having multiple thread contexts

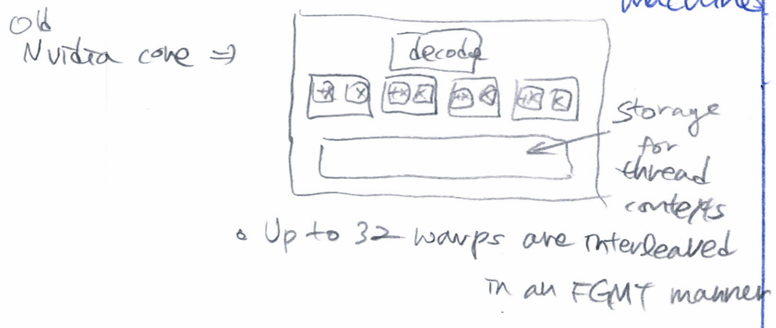
idea ②; Switch to another thread every cycle s.t no two instr from a thread are in the pipeline concurrently




- Sun Niagara ; The first multicore machine
  - Adv = • No need for dep check, branch prediction
    - • otherwise bubble cycles used for useful instr. from different threads
  - Disadv
    - Extra hardware complexity
    - Reduced single thread performance

★ Modern GPUs are fine grained multicore machines

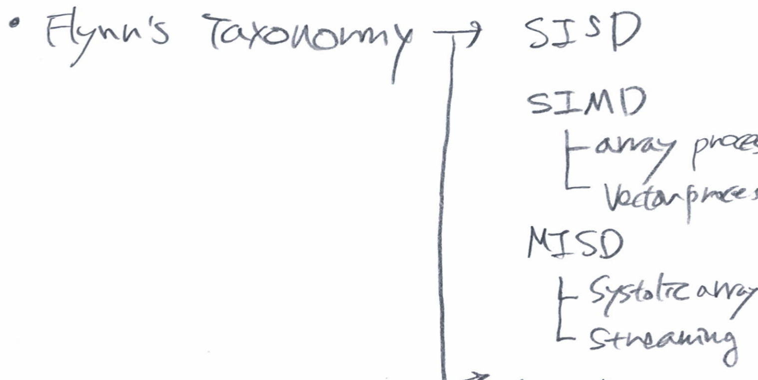
old Nvidia core ⇒




- Up to 32 warps are interleaved in an FGMT manner

## Lecture 20: SIMD Processors

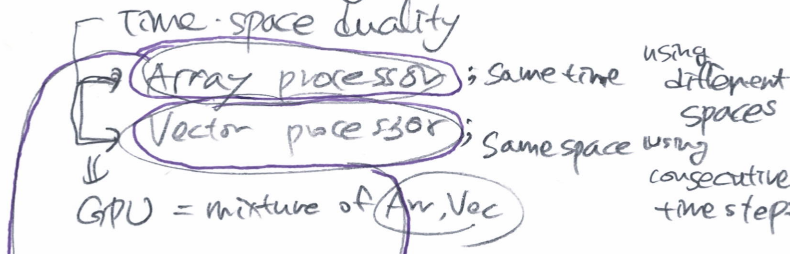
MPCN




- Flynn's Taxonomy

- Data parallelism.
  - ex) dot product of two vectors
  - * Same operation on different pieces of data (≠ Dataflow)

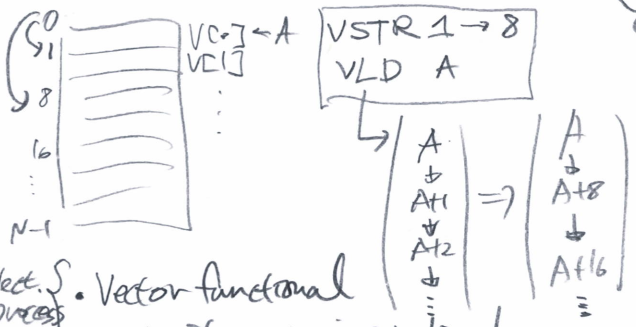
- SIMD processing
  - Time - space duality




multiple processing element

easy..

sequential execution w/
1. Vector registers
2. Vector length register (VLEN)
3. Vector stride register (VSTR)

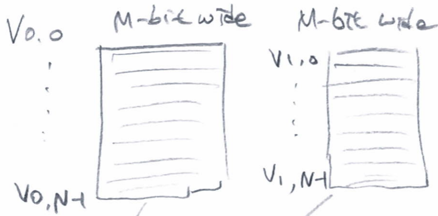



Vect. process

- Vector functional units are pipelined
- Each pipeline stage operates on diff data element
- Deeper pipeline ⇒ No intra-vector dep.
- Must be regular
  - No control flow.
  - Known stride → easy addr calc
  - ⇒ prefetching

* Vector Processor Limitation
  - Data should be regular!
    very inefficient if irregular
  - Memory (Bandwidth) can be a bottleneck

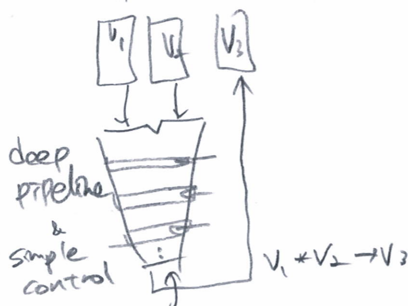
- Vector registers



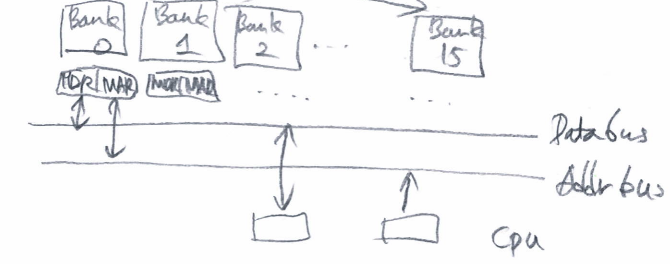

- Vector control register = VLEN, VSTR, VMASK

VMASK[i] = (Vk[i]==0)

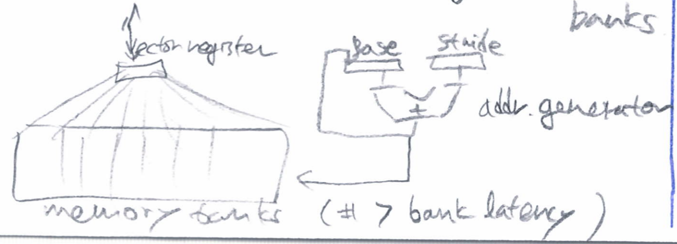
- Vector Functional Units

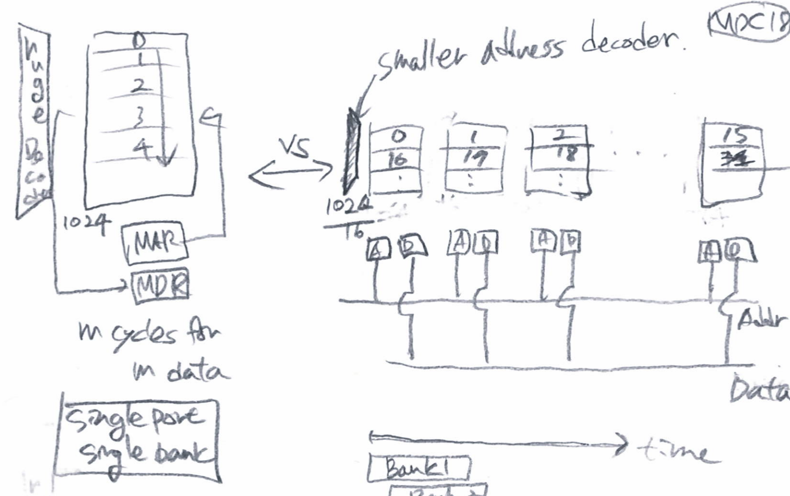


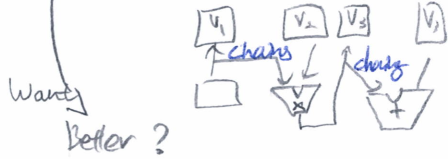

- memory banks ⇒ Loading/storing Vectors from/to memory
  Multiple (16)
  ⇒ 1 element for 1 cycle!
  ⇒ Need multiple mem. banks!!




⇒ sustain N parallel access if all N go to different banks




- Vectorizable Loop. (depends on the stride)

- Vector Chaining; Data forwarding from one vector functional unit to another

Wanty Better ?
Vector chaining w/ 2 ports per bank

QI: what if # of data element > # elements in a vector register?
AI: Break loops!
  ↳ Vector Stripmining

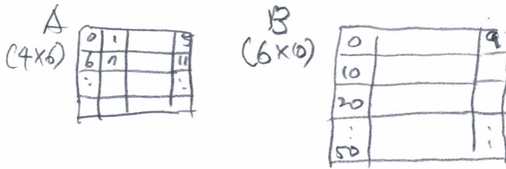
QII: If irregular data?
AII: Use indirection to combine/pack elements into vector register
  ↳ Scatter/Gather operations

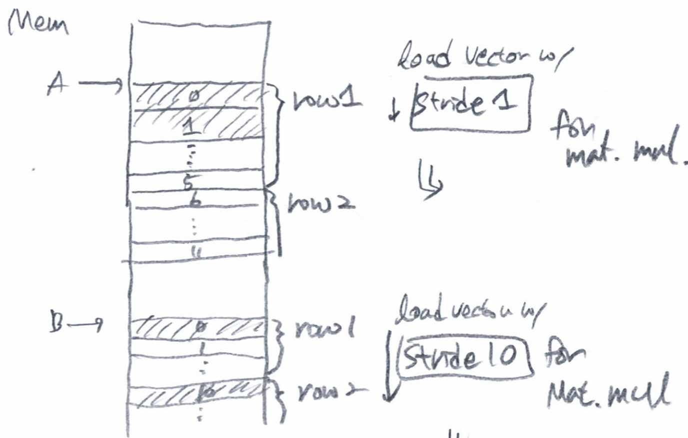
QIII: If Conditional Operations in a loop (Forbid Operation for some element)
AIII: Masked operation (VMASK)
  → "predicated execution"

# Lect 21 SIMD & GPUs

- Stride & banking issues (should be relatively prime)
- Storage of a matrix.
  - Row major: store data row by row
  - Column major: store data column by column.

- Matrix multiply A×B




Different strides can lead to "bank conflicts"

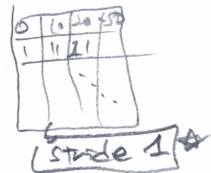
minimizing bank conflicts

① more banks

② Better Data layout → matmul A×B (transpose) $B^T$ (i.e, column major)

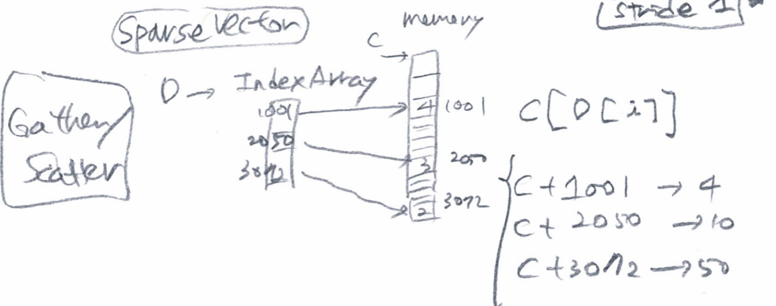
③ Better mapping of address to bank.

E.g) randomized mapping




Sparse vector

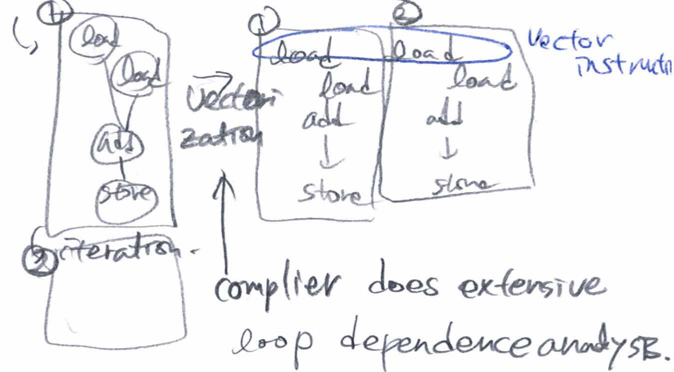
Gather / Scatter




- Modern SIMD = mixture of array processor & vector processor.

ex) Vector Instn Level Parallelism.

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




- SIMD ISA Extensions (graphics)
- Intel Pentium MMX Operations.
  - ↳ Image Overlaying (w/ Bit mask)

## GPU (Graphic Processing Units)

- programming is done using threads, NOT SIMD Instructions.
- Programming Model vs. Hardware Execution Model.

↳ Next page!

Q. How to calculate

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