Mestrado em Eng. Informática

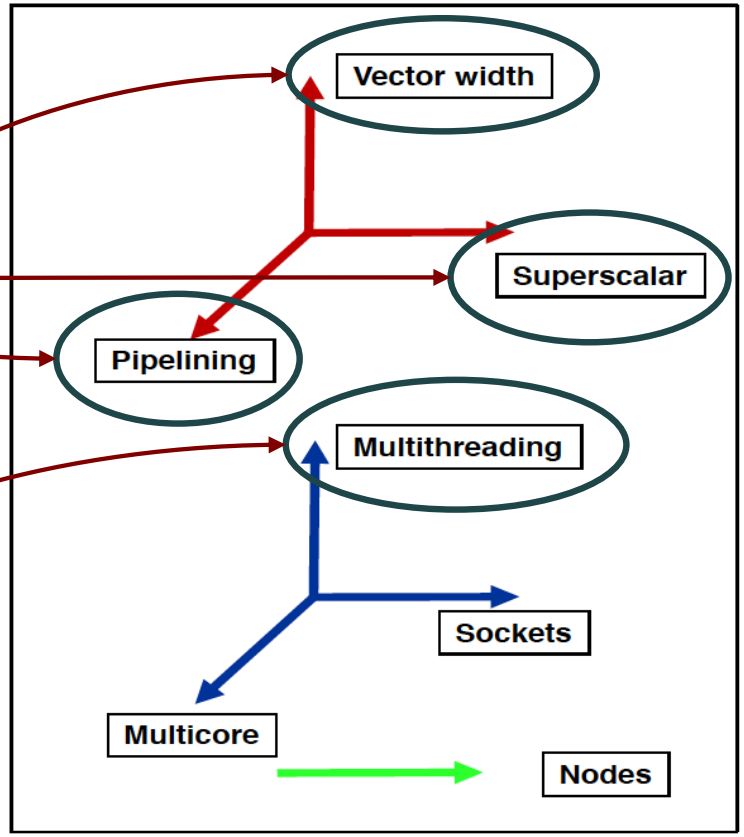
2024/25

J.L. Sobral

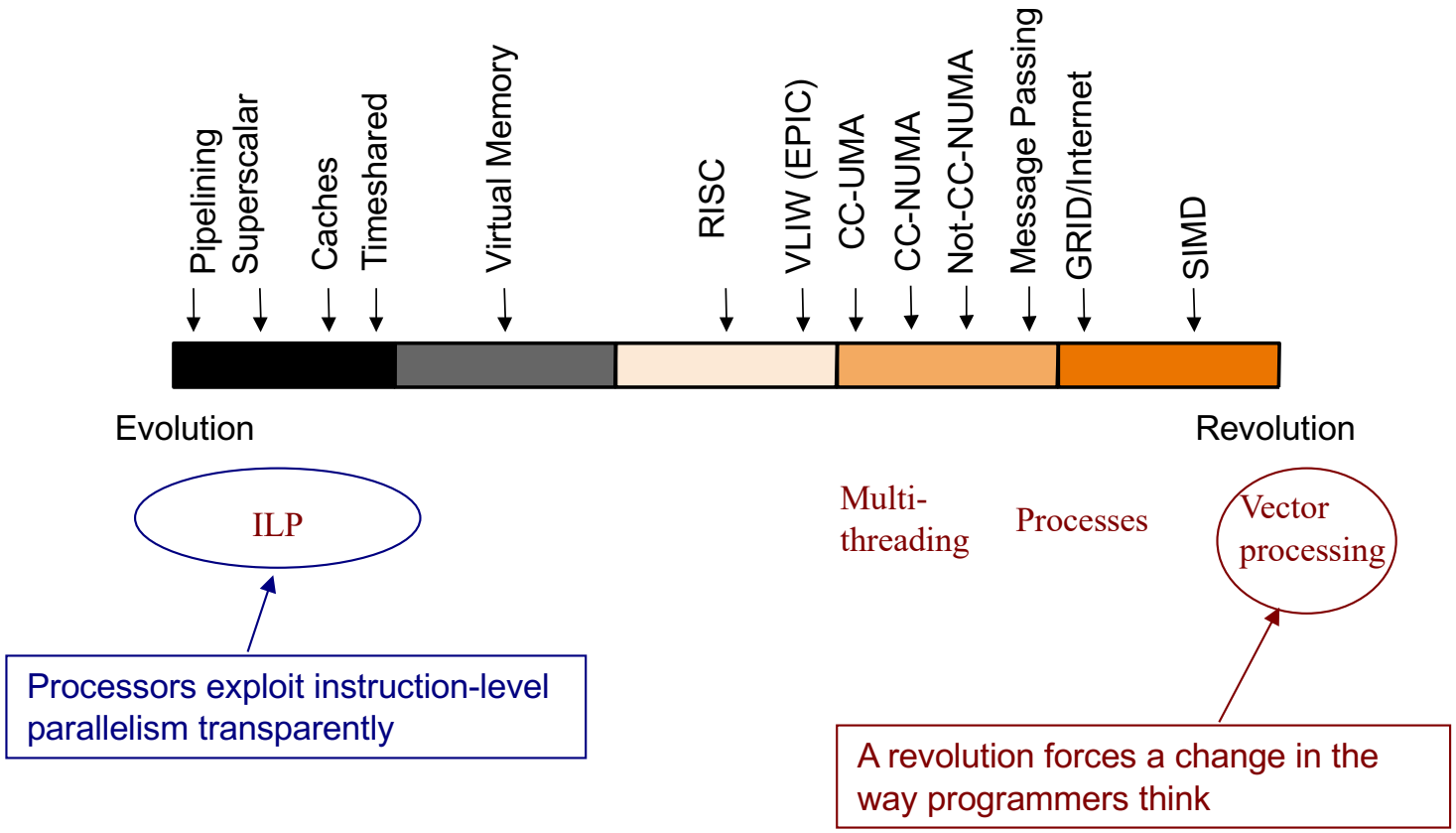
Vector Processing

Key issues for parallelism in a single-core

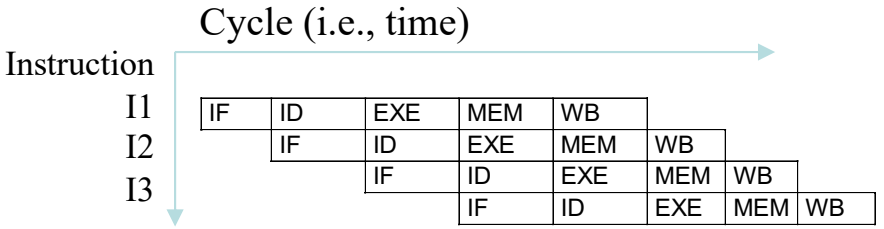
- **Currently under discussion:**
  - pipelining:
  - superscalar:
  - data parallelism: Vector processing
  - multithreading: alternative approaches



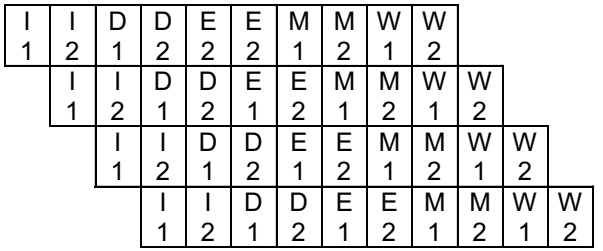
Innovations in Computer Architectures



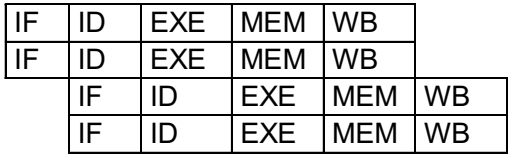
- **Pipelining**
  - Execute the instructions in stages that overlap in time



- **Super-pipelining**
  - Increase the number of stages to enable an increase in the clock cycle frequency

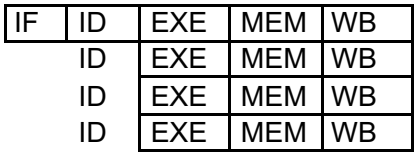


- **Superscalar**
  - Start multiple instructions per cycle



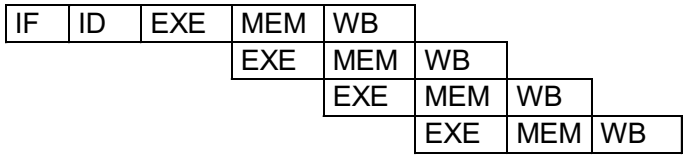
Dynamic scheduling (hardware)

- ***Very Long Instruction Word* (VLIW)**
  - Specify multiple operations per instruction

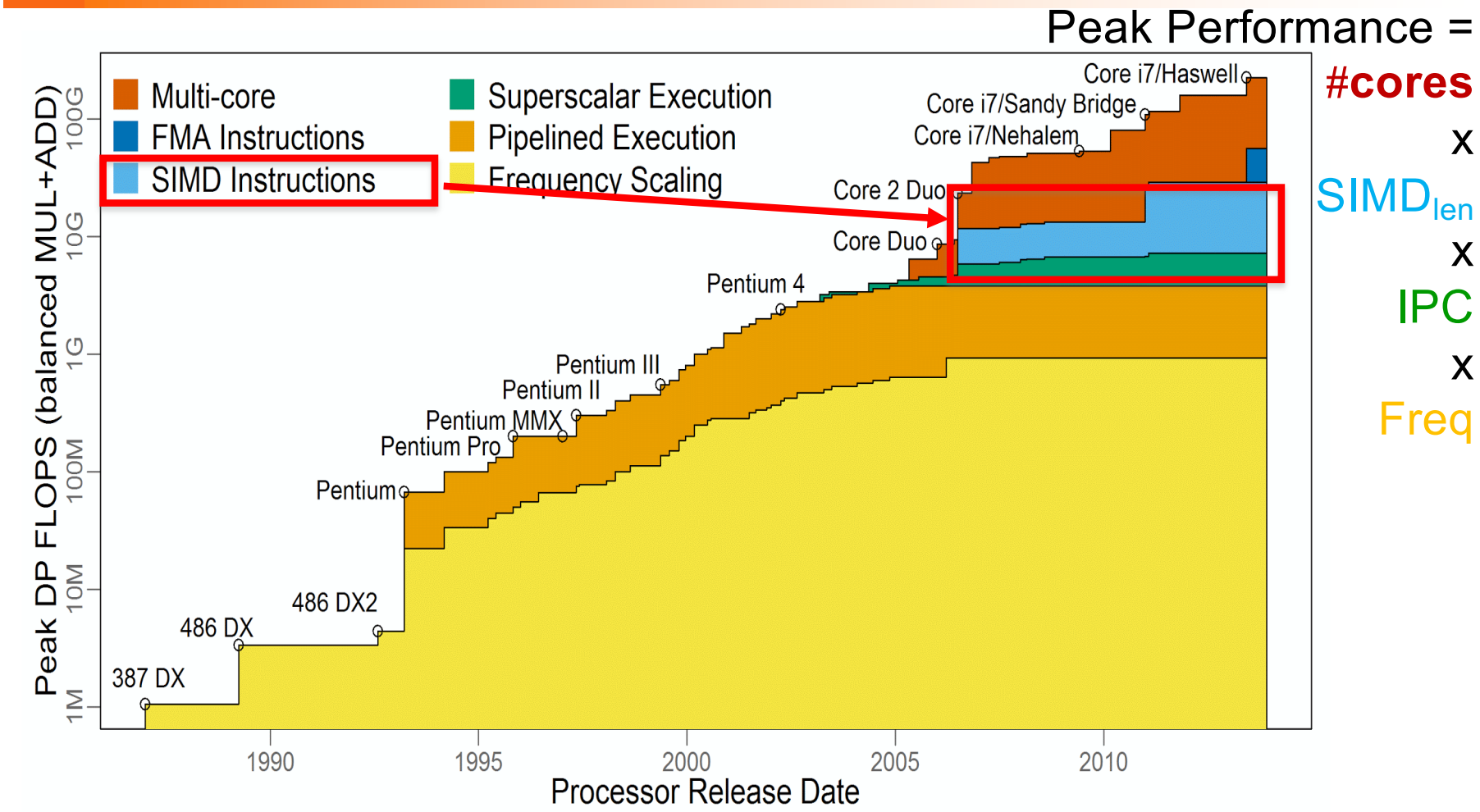


Static scheduling (compiler)

- **Vector Instructions**
  - A series of operations to be performed on multiple data in a single instruction (SIMD)



Evolution of Peak performance



Peak Performance =

#cores

x

$SIMD_{len}$

x

IPC

x

Freq

Texe (ideal) = $\#I_{vect}$ x CPI x Tcc / **#cores** ($\#I_{vect} = \#I/SIMD_{len}$)

Side: Loop-unrolling VS vector processing

-O2	-O2 -funroll-all-loops	-O3 – vetorized 4x - Array

1e00: 83 00 01 addl $1, (%eax)
1e03: 83 c0 08 addl $8, %eax
1e06: 39 c1 cmpl %eax, %ecx
1e08: 75 f6 jne -10
	1c90: 83 00 01 addl $1, (%eax)
	1c93: 83 40 08 01 addl $1, 8(%eax)
	1c97: 83 40 10 01 addl $1, 16(%eax)
	1c9b: 83 40 18 01 addl $1, 24(%eax)
	1c9f: 83 40 20 01 addl $1, 32(%eax)
	1ca3: 83 40 28 01 addl $1, 40(%eax)
	1ca7: 83 40 30 01 addl $1, 48(%eax)
	1cab: 83 40 38 01 addl $1, 56(%eax)
	1caf: 83 c0 40 addl $64, %eax
	1cb2: 39 c1 cmpl %eax, %ecx
	1cb4: 75 da jne -38
		L7:
		vpaddd (%eax), %ymm1, %ymm0
		addl $32, %eax
		vmovdqu %ymm0, -32(%eax)
		cmpl %ecx, %eax
		jne L7

Original

```
for(i=0;i<=N;i++)
    v[i].a++;
...
```

Unrolled 8x

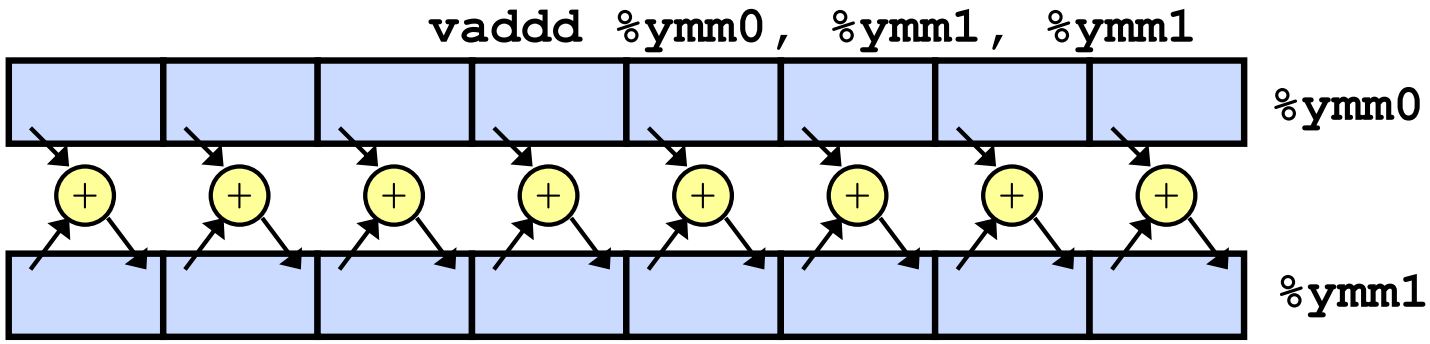
```
for(i=0;i<=N;i+=8) {
    v[i].a++;
    v[i+1].a++;
  ...
```

Modified to support vectorization….

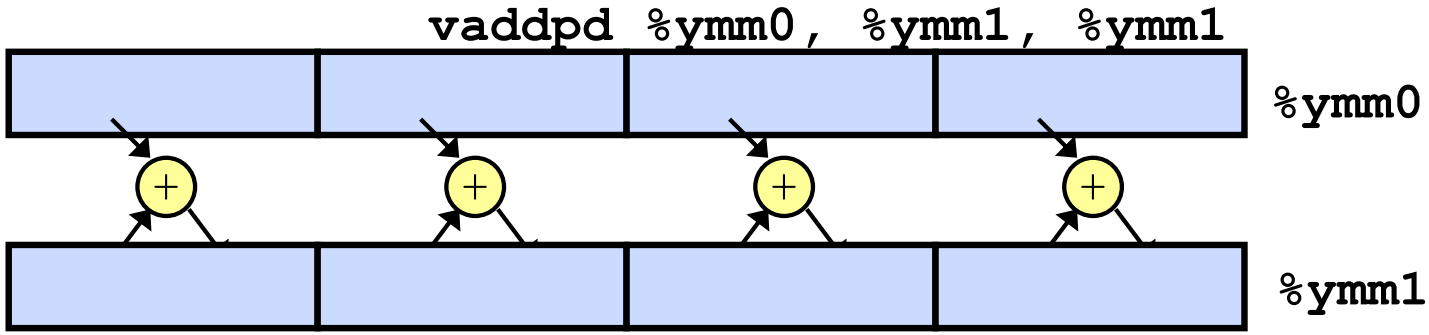
```
for(i=0;i<=N;i++)
    va[i]++;
```

SIMD Operations (Vetorization)

- SIMD Operations: Int (8 x 4 bytes = 32 bytes)



- SIMD Operations: Double Precision



Instruction and Data Streams

Flynn's Taxonomy of Computers *

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Multicore devices

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

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

Introduction

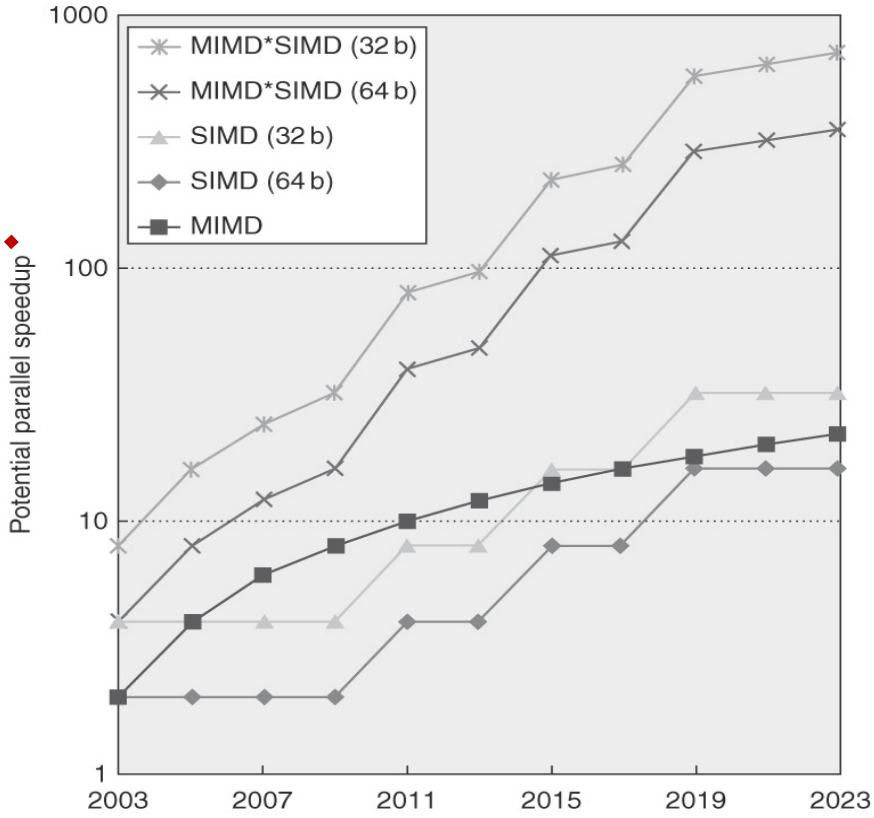
- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
  - Machine learning algorithms
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs) *(in another set of slides)*

- For x86 processors:
  - Expect 2 additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup: SIMD 2x that from MIMD!

Outdated…



Vector Architectures

- Basic idea:
  - Read sets of data elements *(gather from memory)* into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory *(scatter)*
- Registers are controlled by the compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 cycles
    - Floating-point divide => 20 cycles
    - Vector load => 12 cycles
- Improvements:
  - > 1 element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices (mem accesses with nonunit strides)
  - Sparse matrices
  - Programming a vector computer

~~Data dependencies~~
- only usable for data-parallel loads

Control dependencies
- solutions:
  - predication => additional work
  - masking => unused PEs
- support for any vector length

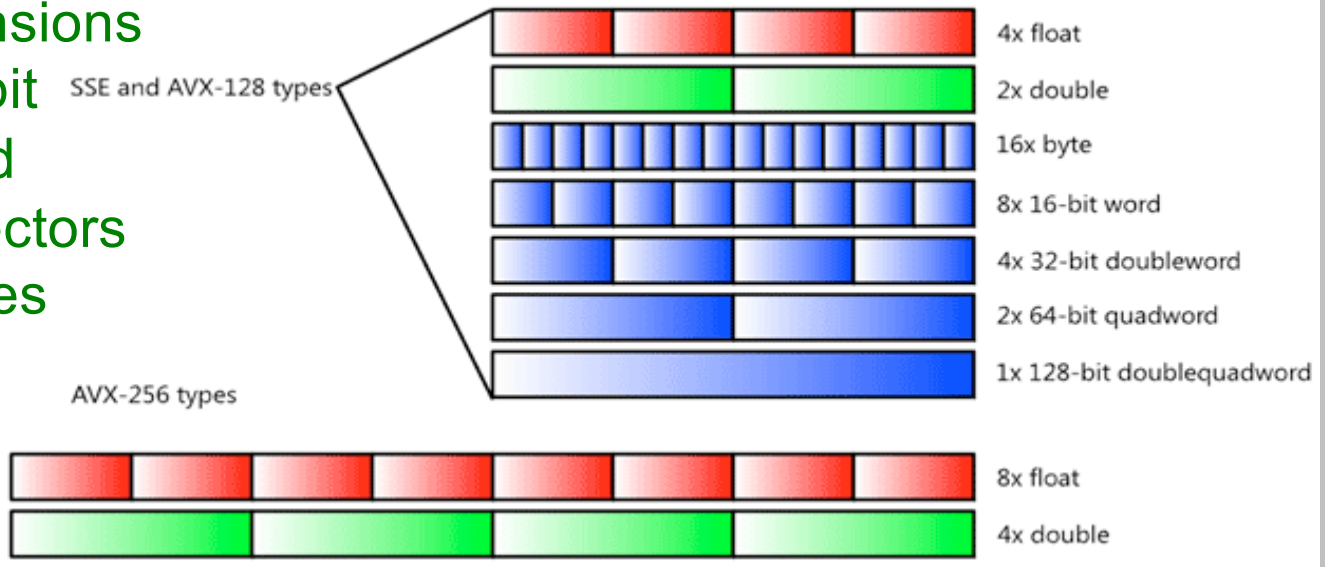
Data locality
- Maximize spatial locality
- Data alignment in memory

Vector Programming

- Compilers are a key element to give hints on whether a code section will vectorize or not
- Check if loop iterations have data dependencies and/or `if…then` statements, otherwise vectorization is compromised
- Vector architectures have a too high cost, but simpler variants are currently available on off-the-shelf devices, as extensions to the scalar processor; **however**:
  - most do not support **non-unit stride** => care must be taken in the design of data structures
  - same applies for **mask register**, **gather-scatter...**

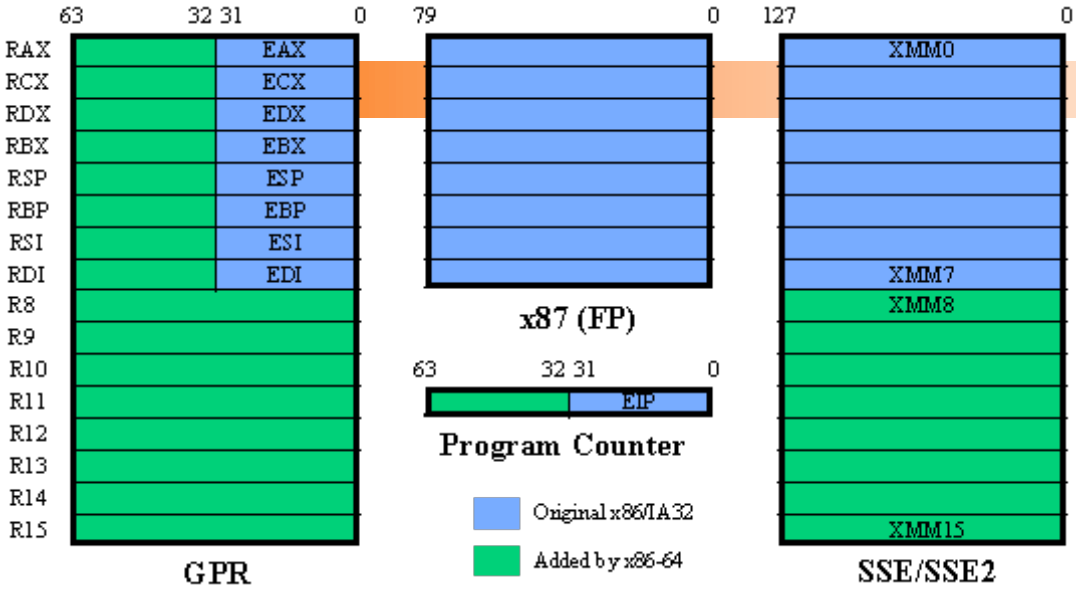
SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Intel SIMD extensions started with 64-bit wide vectors and grew to wider vectors and more facilities
  - Current AVX generation is 512-bit wide

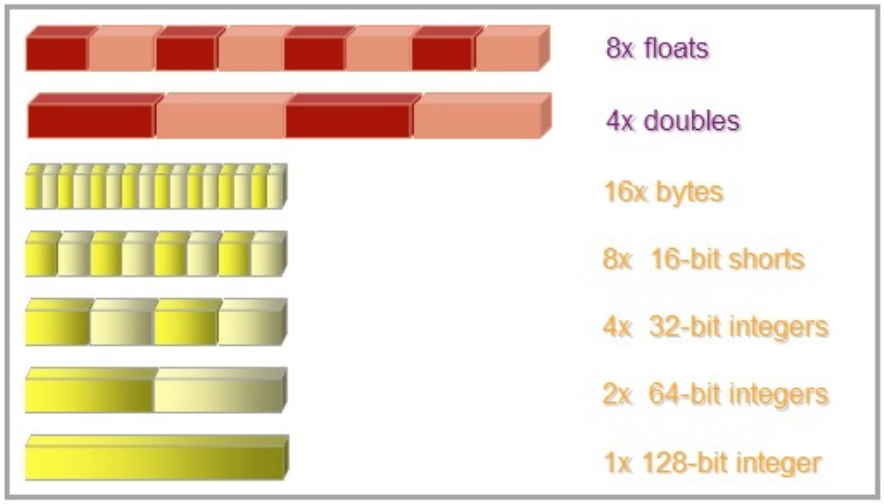


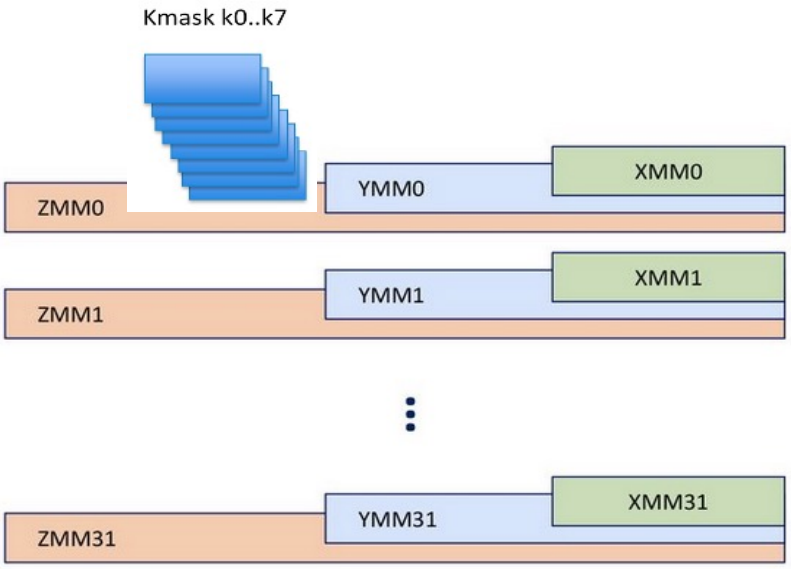
- Limitations, compared to vector architectures:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather, but…)
  - No mask registers

Registers for vector processing in Intel 64

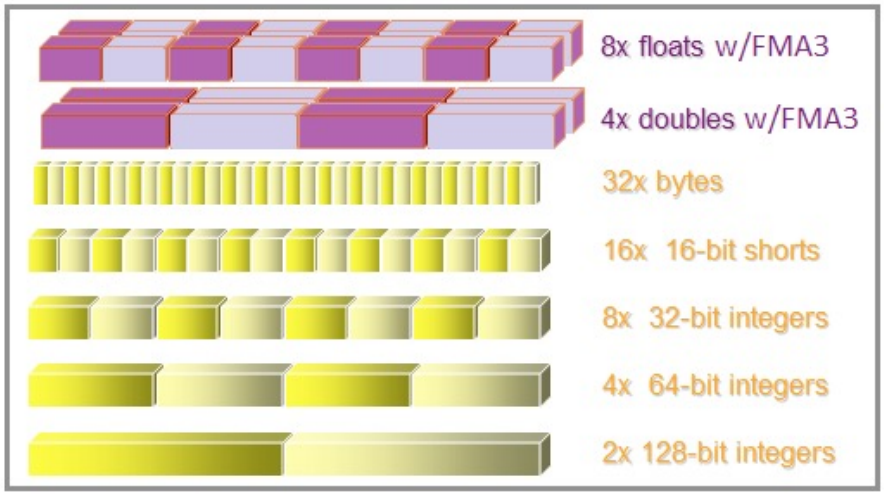


Advanced Vector eXtensions, AVX 1.0





- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512



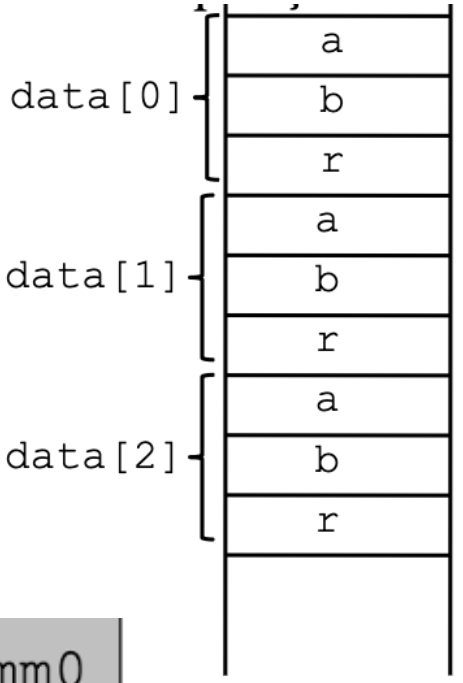
AVX 2.0

Data layout: AoS vs. SoA (1)

A typical C code program defines the following data layout as an Array of Structures (AoS):

```
struct {
  float a, b, r;
} data[100];
```

This array will be stored in memory like this:



Now consider this instruction:

```
movaps (%eax, %ecx, 4), %xmm0
```

It expects to find a vector of 4 floats in consecutive memory cells: namely `a[0],a[1],a[2],a[3]` but it finds a non-unit stride…

Conclusion: the compiler will not vectorize this code!

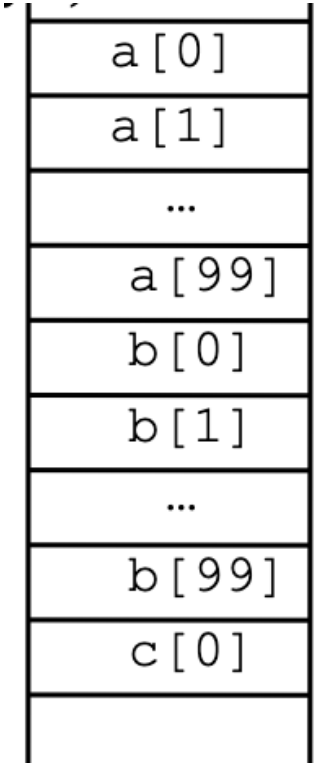
Data layout: AoS vs. SoA (2)

Solution

The C code should redefine the
data layout as a Structure of Arrays (SoA):

```
struct {
   float a[100], b[100], r[100];
} data;
```

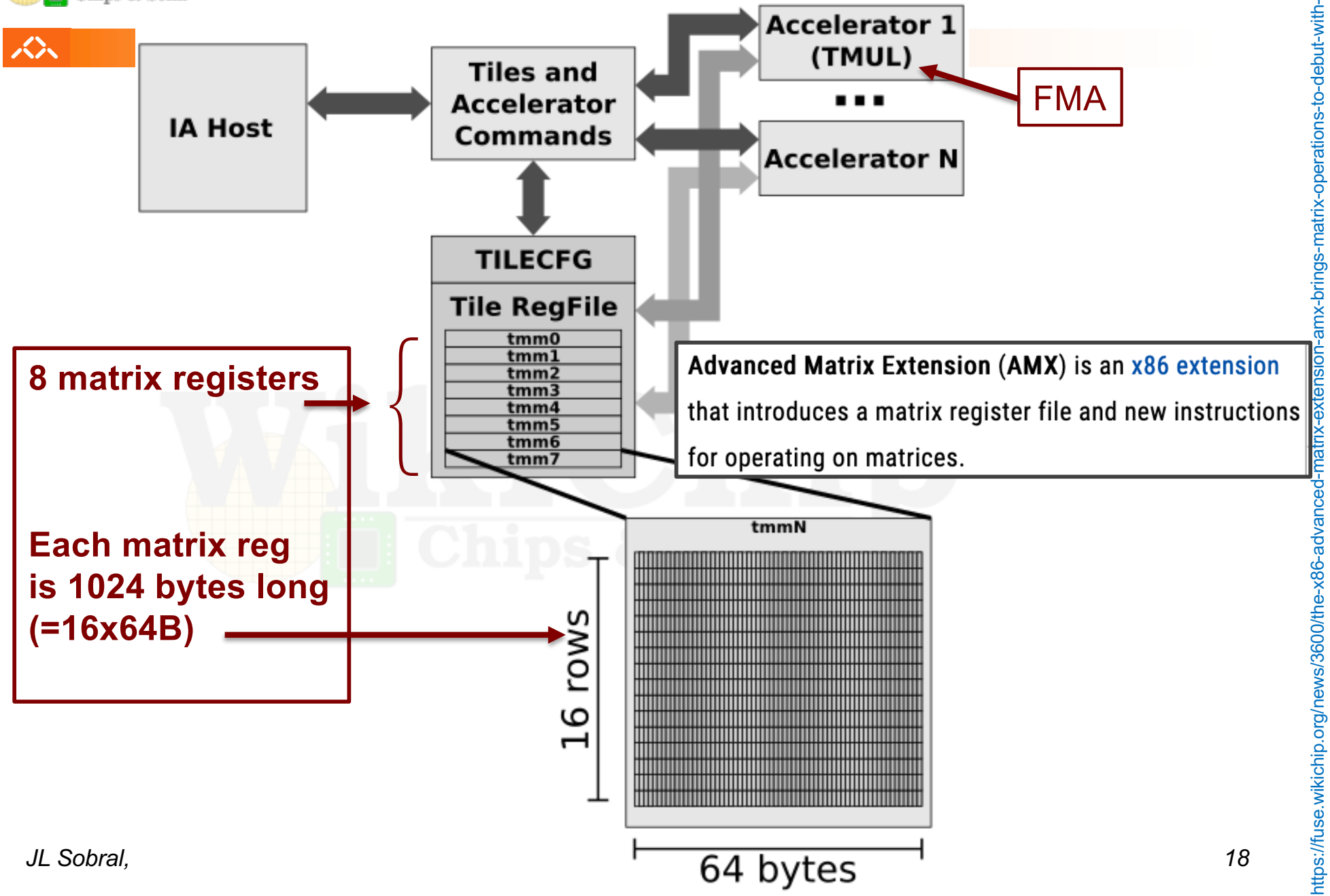
Now the structure of arrays is
stored in memory like this:



Conclusion: the compiler now vectorizes that code!

Intel Advanced Matrix Extension (AMX)

(expected in 2023)



https://fuse.wikichip.org/news/3600/the-x86-advanced-matrix-extension-amx-brings-matrix-operations-to-debut-with-sapphire-rapids/

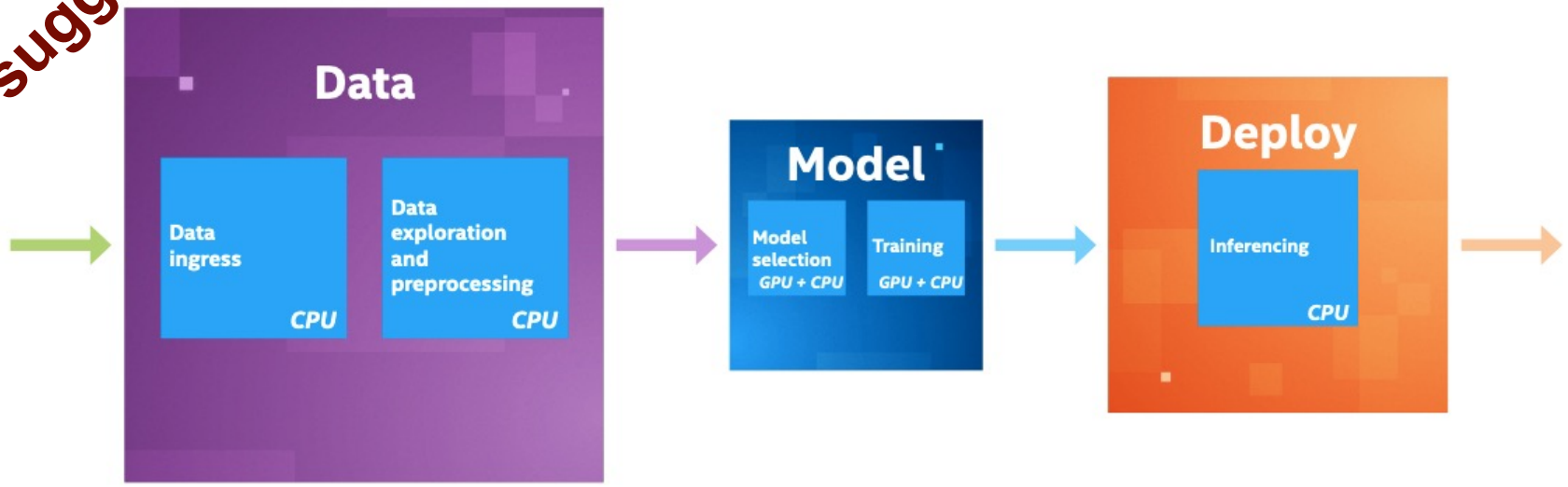
Solution Brief

Artificial Intelligence (AI)
Intel® Advanced Matrix Extensions (Intel® AMX)

Accelerate Artificial Intelligence (AI) Workloads with Intel Advanced Matrix Extensions (Intel AMX)

Reading suggestion

The AI pipeline



The three outer boxes represent AI pipeline stages.
The five inner boxes represent AI workloads.
Box sizes indicate relative levels of processor activity within the AI pipline.

https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/ai-solution-brief.html

Beyond vector extensions

- Vector/SIMD-extended architectures are hybrid approaches
  - mix (super)**scalar + vector** op capabilities on a single device
  - **highly pipelined** approach to reduce memory access penalty
  - **tightly-closed access to shared memory**: lower latency
- Evolution of vector/SIMD-extended architectures
  - **computing accelerators optimized for number crunching (GPU)**
  - **add support for matrix multiply + accumulate operations**; **why?**
    - most scientific, engineering, AI & finance applications use matrix computations, namely the dot product: multiply and accumulate the elements in a row of a matrix by the elements in a column from another matrix
    - manufacturers typically call these extension **Tensor Processing Unit** (TPU)
  - **support for half-precision FP & 8-bit integer; why?**
    - machine learning using neural nets is becoming very popular; to compute the model parameter during training phase, intensive matrix products are used and with very low precision (is adequate!)