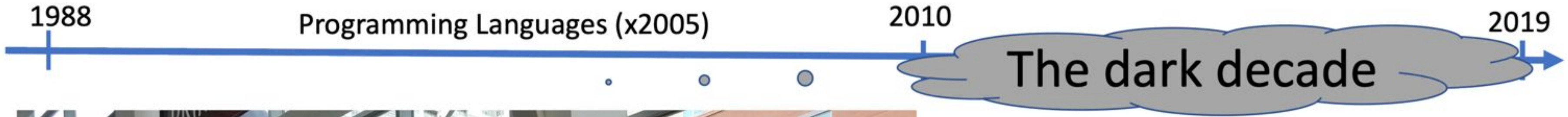
Introduction to all forms of parallel computing

Prof Alan Edelman

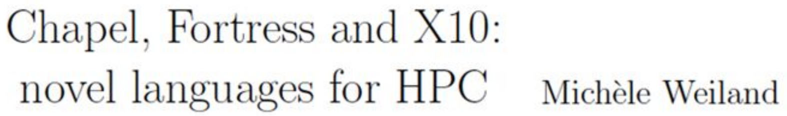
(18.s192/16.s098)

- All material on https://github.com/mitmath/Parallel-Computing-Spoke
  - Suggest bookmarking
- Canvas only used for submitting hws, nothing else



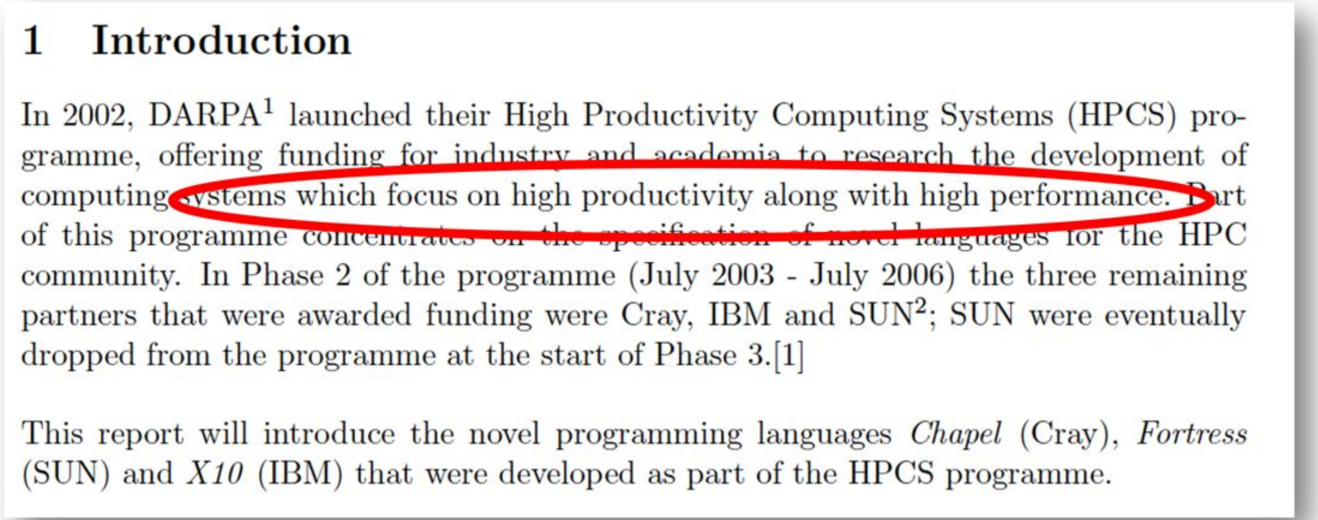
Something seems to have gone wrong!

What went wrong 23 years ago?



Chapel, Fortress and X10: novel languages for HPC

Michèle Weiland

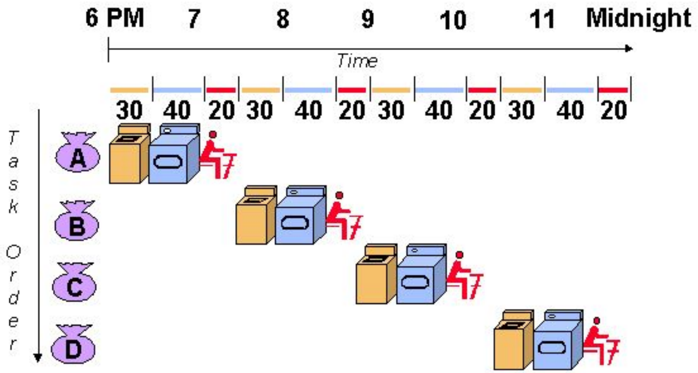


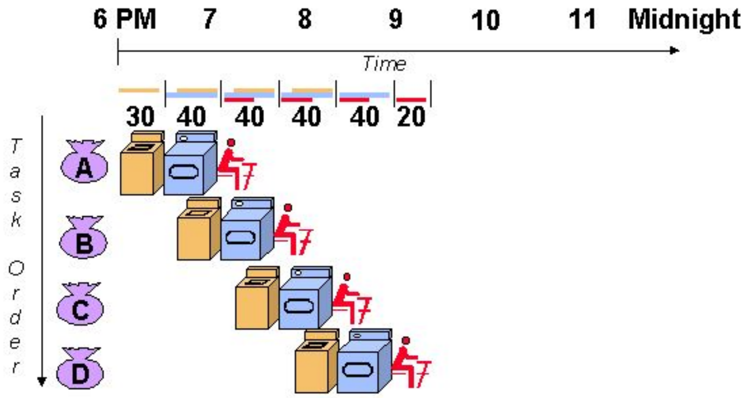
1 Introduction

In 2002, DARPA[1] launched their High Productivity Computing Systems (HPCS) programme, offering funding for industry and academia to research the development of computing systems which focus on high productivity along with high performance. Part of this programme concentrates on the specification of novel languages for the HPC community. In Phase 2 of the programme (July 2003 - July 2006) the three remaining partners that were awarded funding were Cray, IBM and SUN[2]; SUN were eventually dropped from the programme at the start of Phase 3.[1]

This report will introduce the novel programming languages *Chapel* (Cray), *Fortress* (SUN) and *X10* (IBM) that were developed as part of the HPCS programme.

Pipelining: only works if multiple resources are available



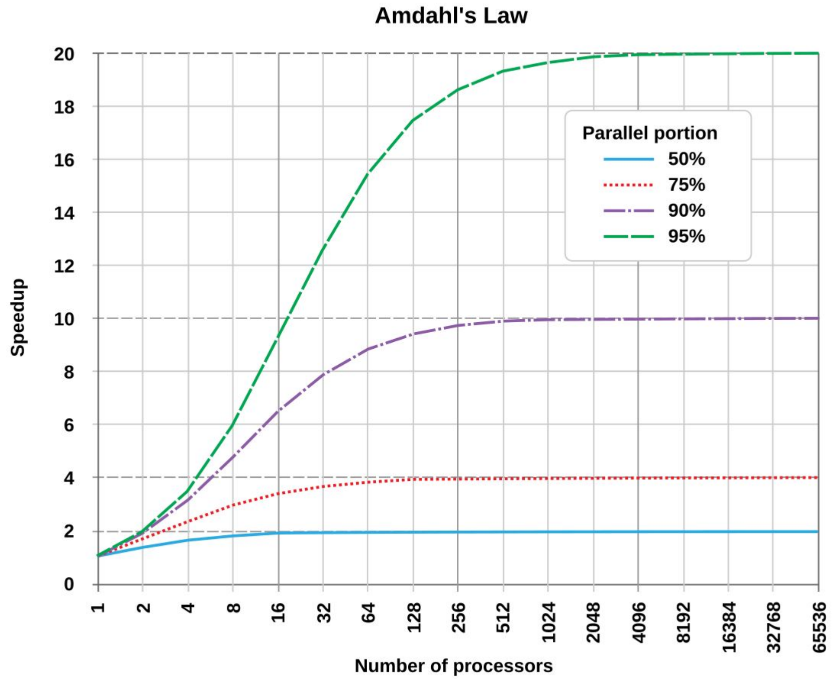


Pipeline 1: dryer
Pipeline 2: washer
Pipeline 3: iron
Threads A, B, C, D

If all threads needed only a washer, multithreading would give no benefit!

Source: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html

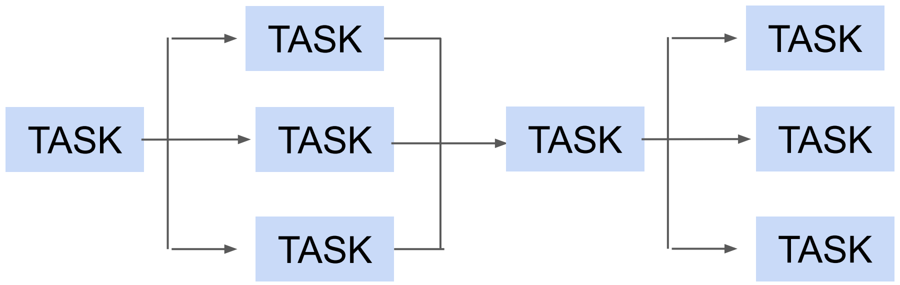
Parallel bottlenecks and Amdahl’s law



$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Serial part of job =
1 (100%) - Parallel part

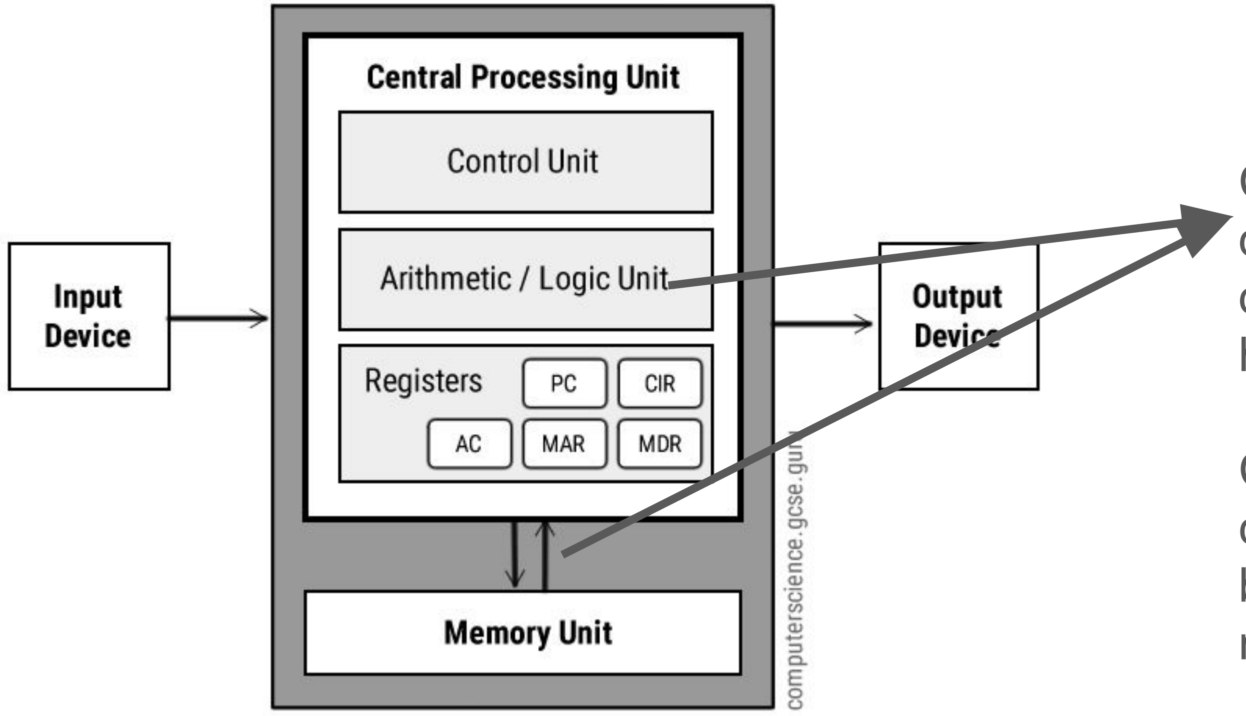
Parallel part is divide
up by N workers



https://runestone.academy/ns/books/published/welcomecs/ParallelProcessing/AmdahlsLaw.html
https://en.wikipedia.org/wiki/Amdahl%27s_law

Intro to hardware for parallelism

Hardware details: von Neumann architecture

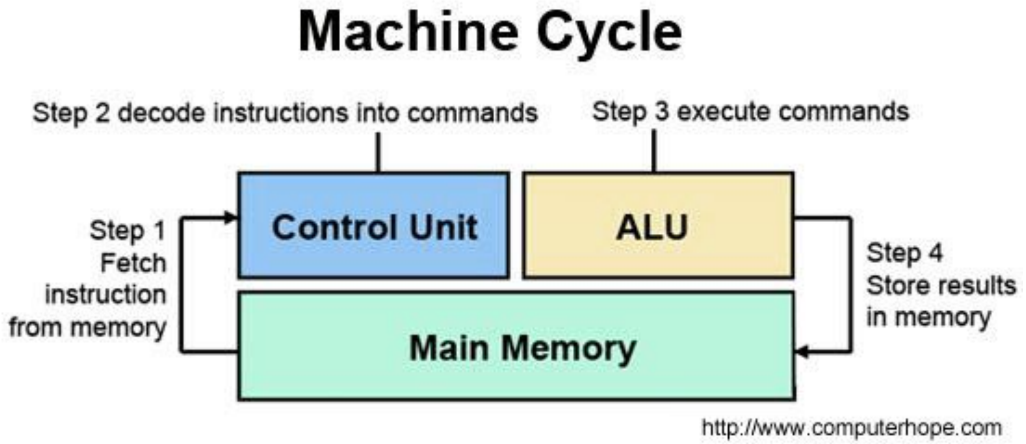


Computation and data retrieval are different physical hardware locations

Computation and communication can be overlapped on the macro level!

https://www.computerscience.gcse.guru/theory/von-neumann-architecture

How many clock cycles does it take to execute 1+1?

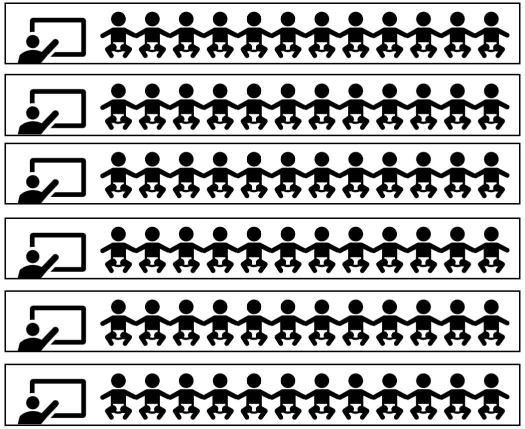

Machine Cycle
Step 2 decode instructions into commands
Step 3 execute commands
Control Unit
ALU
Step 1 Fetch instruction from memory
Step 4 Store results in memory
Main Memory
http://www.computerhope.com

At least 4!

Types of processors: CPU vs GPU - MIT students vs kindergarten classes

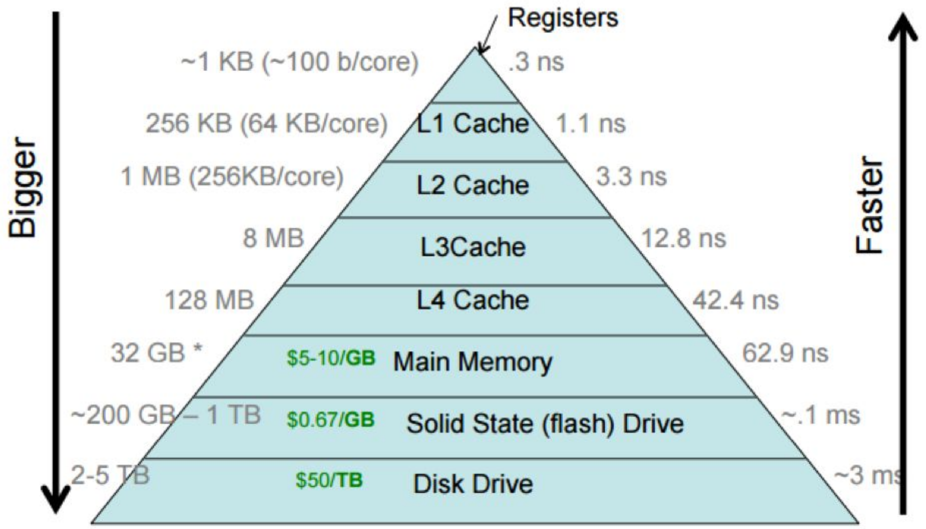
CPU

GPU



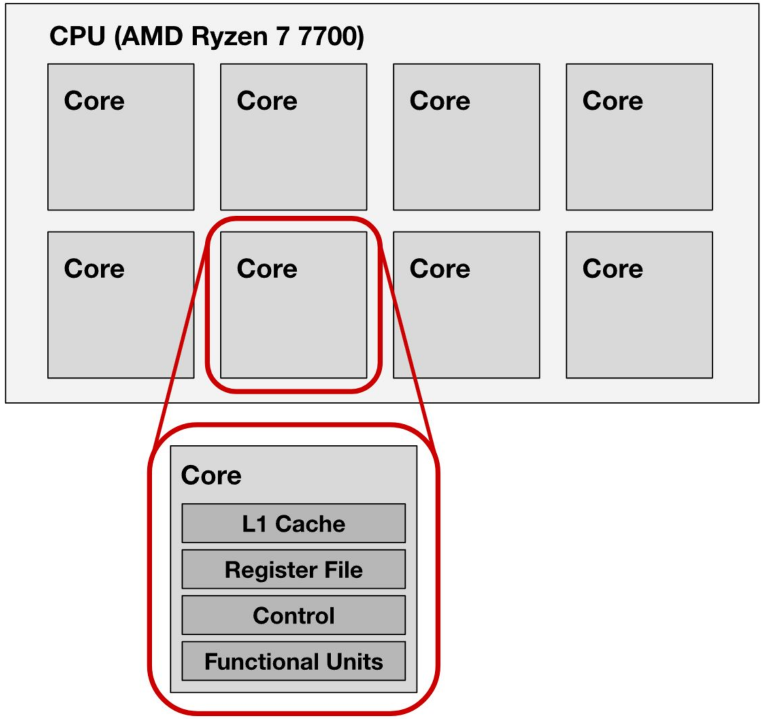
- Kindergarten classes operate in groups of 32, directed by a teacher (large quantity)
- Grad students operate independently (large quality)
- Classes on the GPU need to receive an instruction from a grad student what to do!

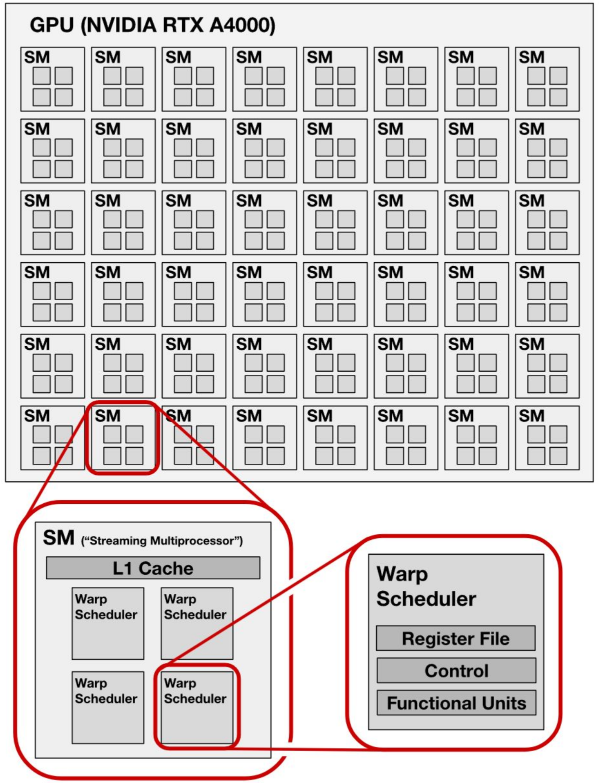
Cache levels and memory locations: the farther the data, the higher the latency



https://cs.brown.edu/courses/csci1310/2020/notes/l10.html

CPU vs GPU: the technical





https://accelerated-computing-class.github.io/fall24/labs/lab2/

Intro to julia for parallelism

HPC Tooling

julia

OpenCilk

(based on Tapir/LLVM)

+ Enzyme

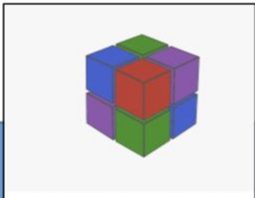
+ Tiramisu

hardware

Q-Chem, Molpro

QuantumESPRESSO
Terachem
VASP

LAMMPS



GitHub - eth-cscs/ImplicitGlobalGrid.jl ...
github.com

Accelerators

CUDA.jl

AMDGPU.jl

OneAPI.jl

JuliaGPU/
GPUArrays.jl

Reusable array functionality for Julia's various GPU backends.

https://github.com › JuliaGPU › KernelAbstractions

KernelAbstractions.jl - Heterogeneous programming in Julia

Heterogeneous programming in **Julia**. Contribute to JuliaGPU/KernelAbstractions.jl development by creating an ... **JuliaGPU** / **KernelAbstractions**.jl Public.

Shared Mem

JuliaFolds/**FLoops.jl**

Fast sequential, threaded, and distributed for-loops for Julia—fold for humans™

Announcing composable multi-threaded parallelism in Julia

23 July 2019 | **Jeff Bezanson (Julia Computing), Jameson Nash (Julia Computing), Kiran Pamnany (Intel)**

Base / Multi-Threading

Multi-Threading

`Base.Threads.@threads` — Macro

Julia Atomics Manifesto

This proposal aims to define the memory model of Julia and to provide certain guarantees in the presence of data races, both by default and through providing intrinsics to allow the user to specify the level of guarantees required. This should allow native implementation in Julia of simple system primitives (like mutexes), interoperate with native system code, and aim to give generally explainable behaviors without incurring significant performance cost. Additionally, it strives to be general-purpose and yet clear about the user's intent—particularly with respect to ensuring that an atomic-type field is accessed with proper care for synchronization.

Distributed

JuliaParallel/**MPI.jl**

MPI wrappers for Julia

JuliaParallel/
Dagger.jl

A framework for out-of-core and parallel execution

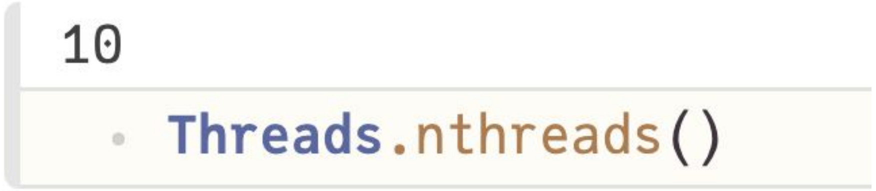
Standard Library / Distributed Computing

Distributed Computing

Appendix

Multithreading in Julia

- Pluto is automatically multithreading enabled
- hyperthreading = usually pretending a physical core is two virtual cores
- Pluto's approach: system cores/2 = https://github.com/fonsp/Pluto.jl/blob/f9f8542af30491f89606e490d46102ae1406511f/src/Configuration.jl#L149-L158



```
10
Threads.nthreads()
```

- See https://docs.julialang.org/en/v1/manual/multi-threading/

Serial Julia Performance Tips

- https://docs.julialang.org/en/v1/manual/performance-tips/
- Most important is
    - type instability (tool: @code_warntype)

  https://docs.julialang.org/en/v1/manual/performance-tips/#Write-%22type-stable%22-functions

    - and memory allocations (tool: @time)
        - pre-allocate
        - static arrays
        - consider views

- specifying types often not critical for performance
- vectorization itself is not required but fused broadcasting can help (@. love it or hate it?)

What is SIMD?

- Simple at a high level - details are full of acronyms
- Sometimes “called vectorization” or “vectorization the hardware level”
  - software vectorization is when users write codes to mimic the hardware because the compiler or programming language can not, or because it is more elegant, or more explicit
- SIMD = Single Instruction Multiple Data
  - e.g. performing four multiplies one after another vs at the same time
-

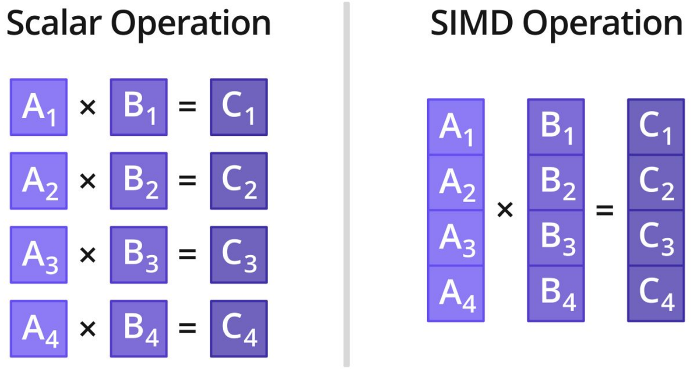


Image source: https://medium.com/wasmer/webassembly-and-simd-13badb9bf1a8

Understanding SIMD in depth – which depth?

- Hardware Level
  - Your laptop cores can perform SIMD
  - GPUs have major SIMD units
  - Economics: SIMD is cheap, don’t need to reproduce logic, big bang for the buck if computations allow
- Instruction Level
  - @code_native in julia shows you the instructions, learning to read them is another matter, but google is your friend here
- Julia Level
  - May be automatic, can use @SIMD, may use broadcasting (the dot “.”)
  - Julia finicky about not be automatic when an operation is not explicitly exactly associative

Might be fun to learn a bit about your hardware and instruction sets, but anyway it will change in a few years when you buy a new one, and it is a bit of an endless rathole. I like to poke a little then stop.