Scheduling Array Operations

Juuso Haavisto (University of Oxford), October 2022

Static Semantics

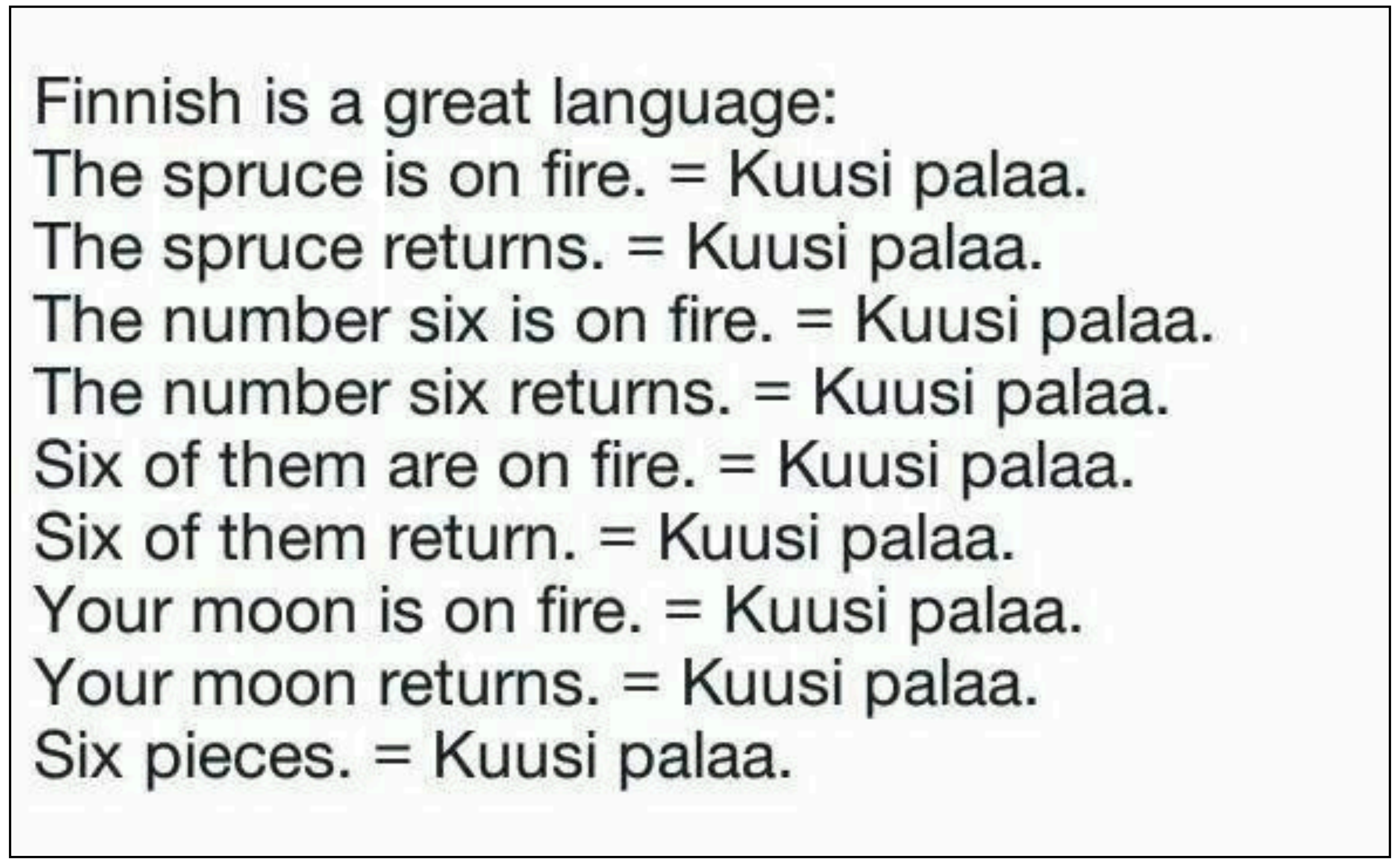
Ability to detect correctness errors before execution

Statically absurd situation:
Waterfal (Escher, 1961)

Rank Polymorphism

Language property where semantically same thing means multiple things depending on the context

Challenge: building connection between static semantics

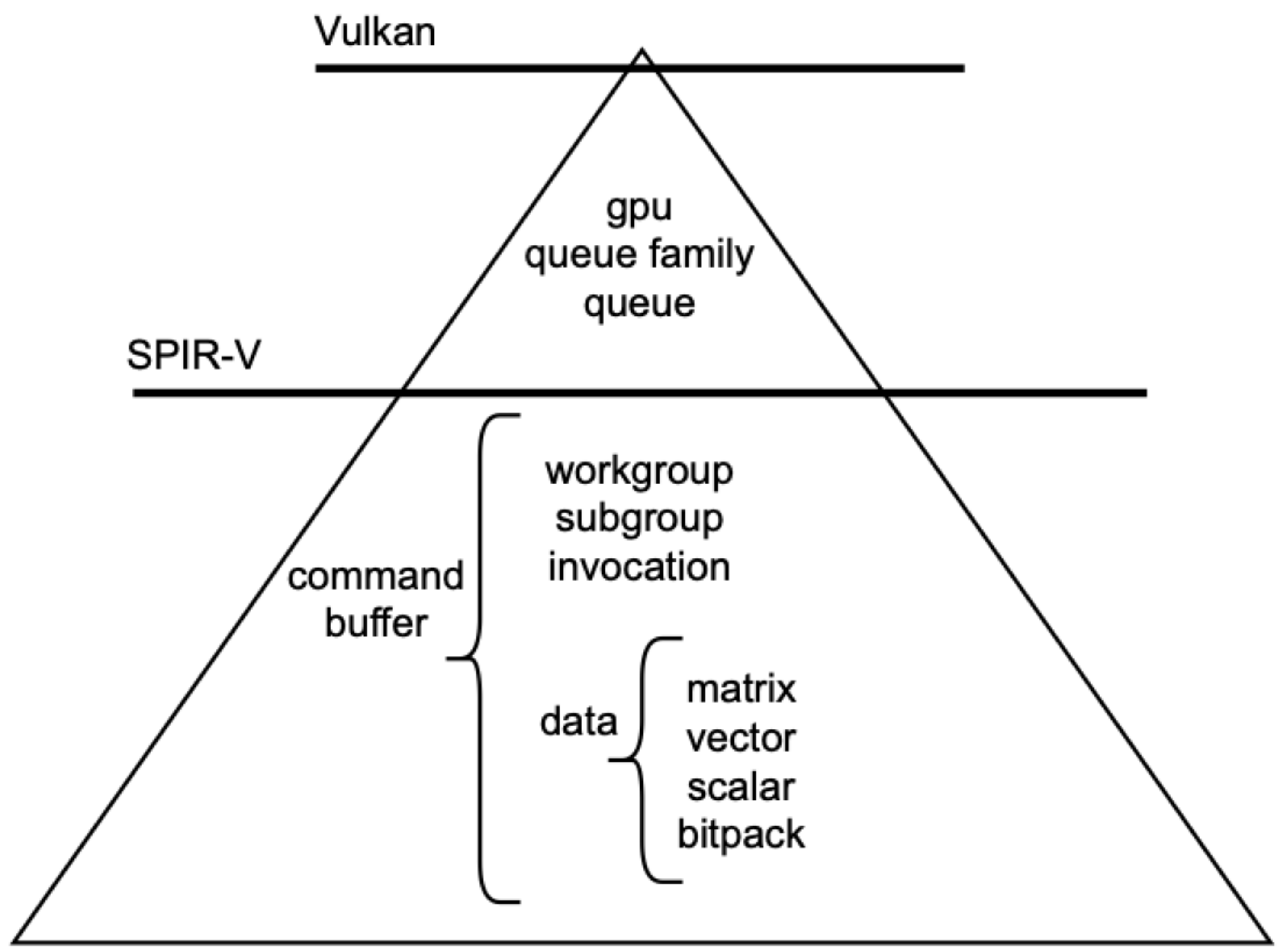


Finnish is a great language:
The spruce is on fire. = Kuusi palaa.
The spruce returns. = Kuusi palaa.
The number six is on fire. = Kuusi palaa.
The number six returns. = Kuusi palaa.
Six of them are on fire. = Kuusi palaa.
Six of them return. = Kuusi palaa.
Your moon is on fire. = Kuusi palaa.
Your moon returns. = Kuusi palaa.
Six pieces. = Kuusi palaa.

A certain kind of rank polymorphism in action

Scheduling

With static rank polymorphism, we can give GPU a data structure that it understands, and from which to derive scheduling schemes for multi-core use

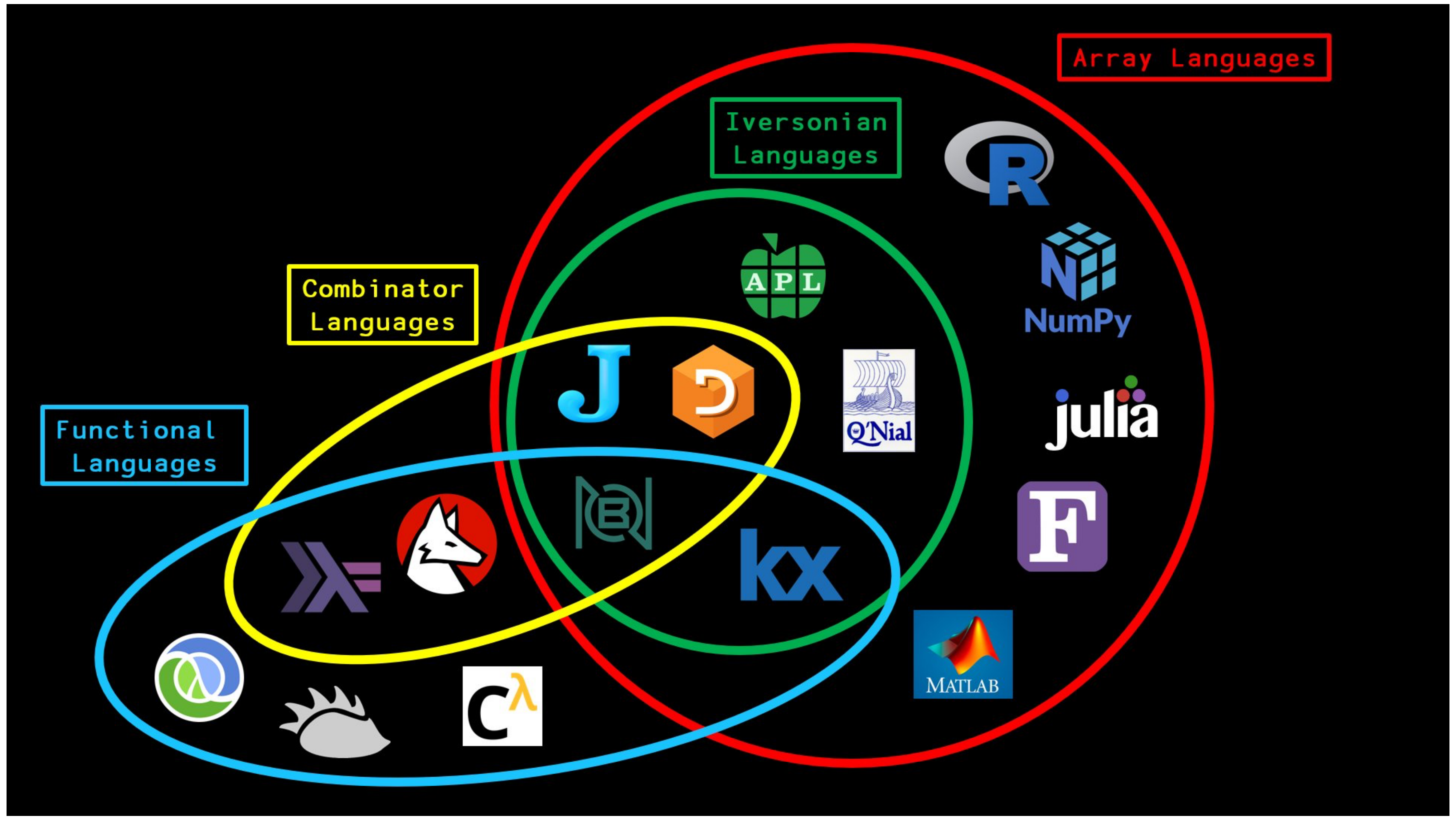


Hierarchy of GPU

Big picture

“Moving Haskell towards Dyalog APL”

- APL programmers “see” a lot about their programs
- but the computer… not so much
- my research is about revisiting what the computer can “see” without adding new language semantics

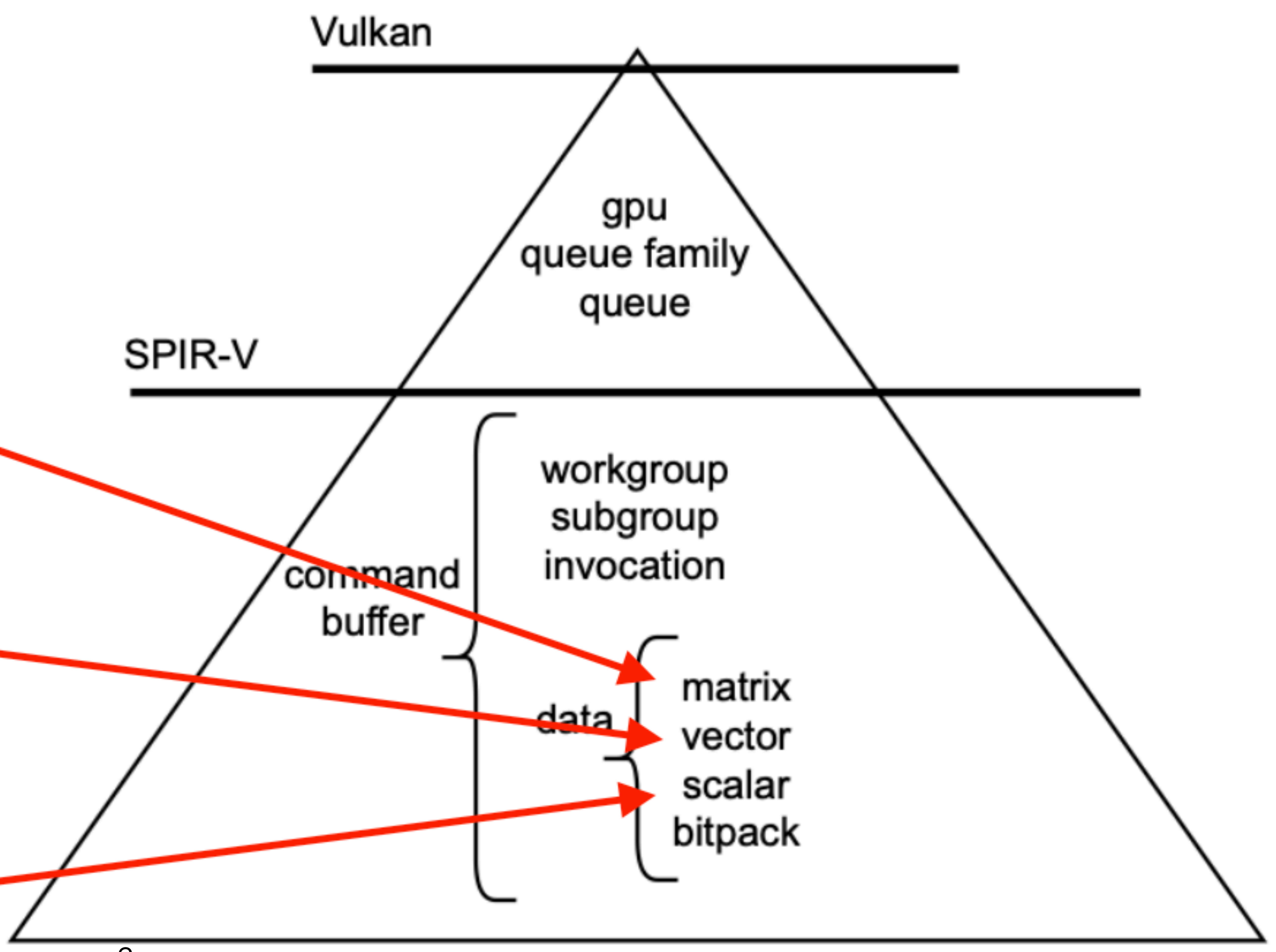


“Array Languages vs Iversonian Languages vs Combinator Languages vs Functional Languages” - Conor Hoekstra @code_report

Max Pooling example, Rodrigo Serrão

MX←{⌈⌿[2],[2 3]{ω}⌺(2 2ρ2)⊃Z[α]←⊂ω}

```
      M
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
      {⊂⍵}⌺(2 2)⊢M
┌─────┬─────┬─────┐
│1 2  │2 3  │3 4  │
│5 6  │6 7  │7 8  │
├─────┼─────┼─────┤
│5  6 │ 6  7│ 7  8│
│9 10 │10 11│11 12│
├─────┼─────┼─────┤
│ 9 10│10 11│11 12│
│13 14│14 15│15 16│
└─────┴─────┴─────┘
      {⊂⌈⌿⍵}⌺(2 2)⊢M
┌─────┬─────┬─────┐
│5 6  │6 7  │7 8  │
├─────┼─────┼─────┤
│9 10 │10 11│11 12│
├─────┼─────┼─────┤
│13 14│14 15│15 16│
└─────┴─────┴─────┘
      {⊂⌈⌿⌈⌿⍵}⌺(2 2)⊢M
 6  7  8
10 11 12
14 15 16
```

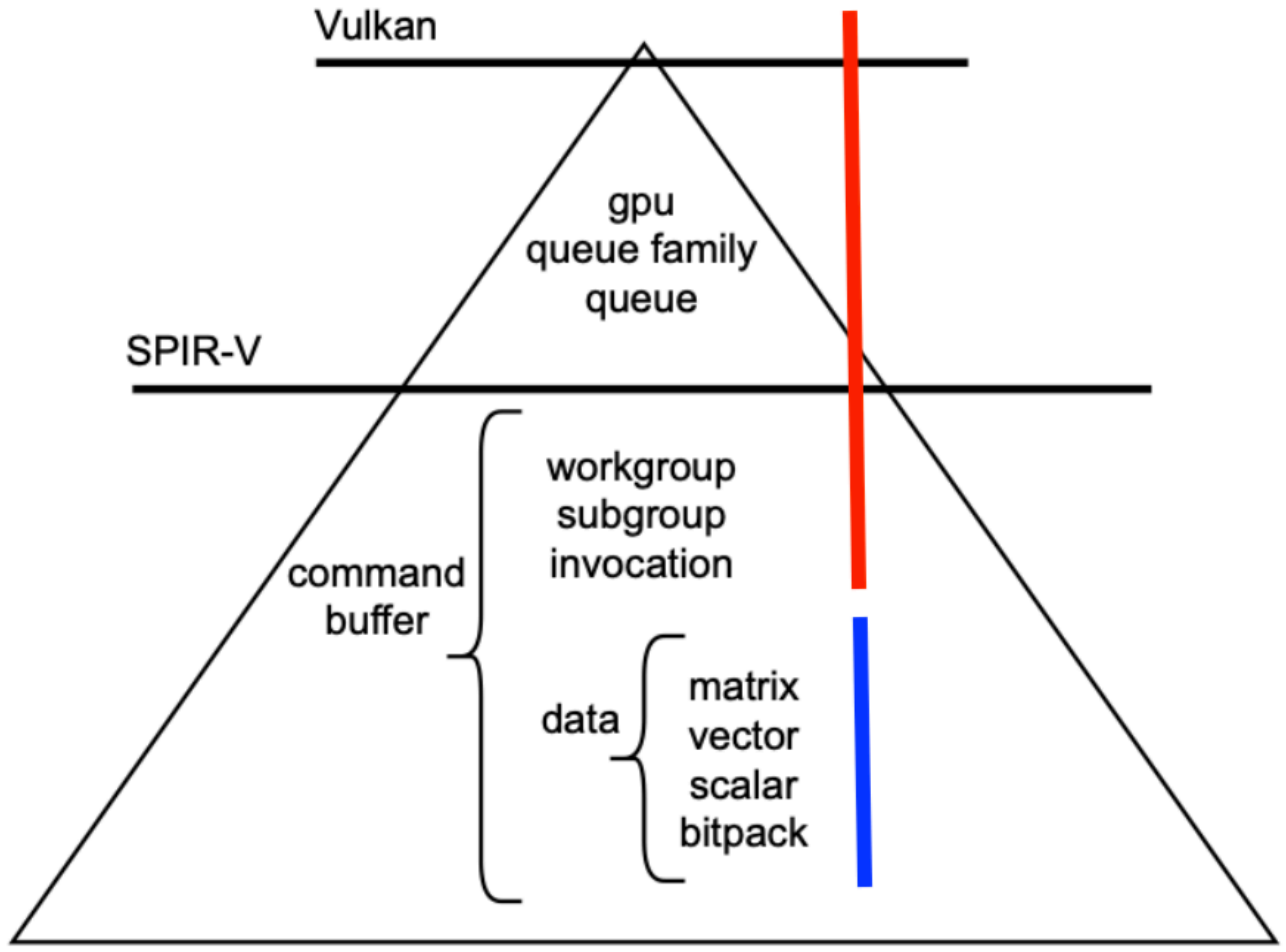


Modern Hardware

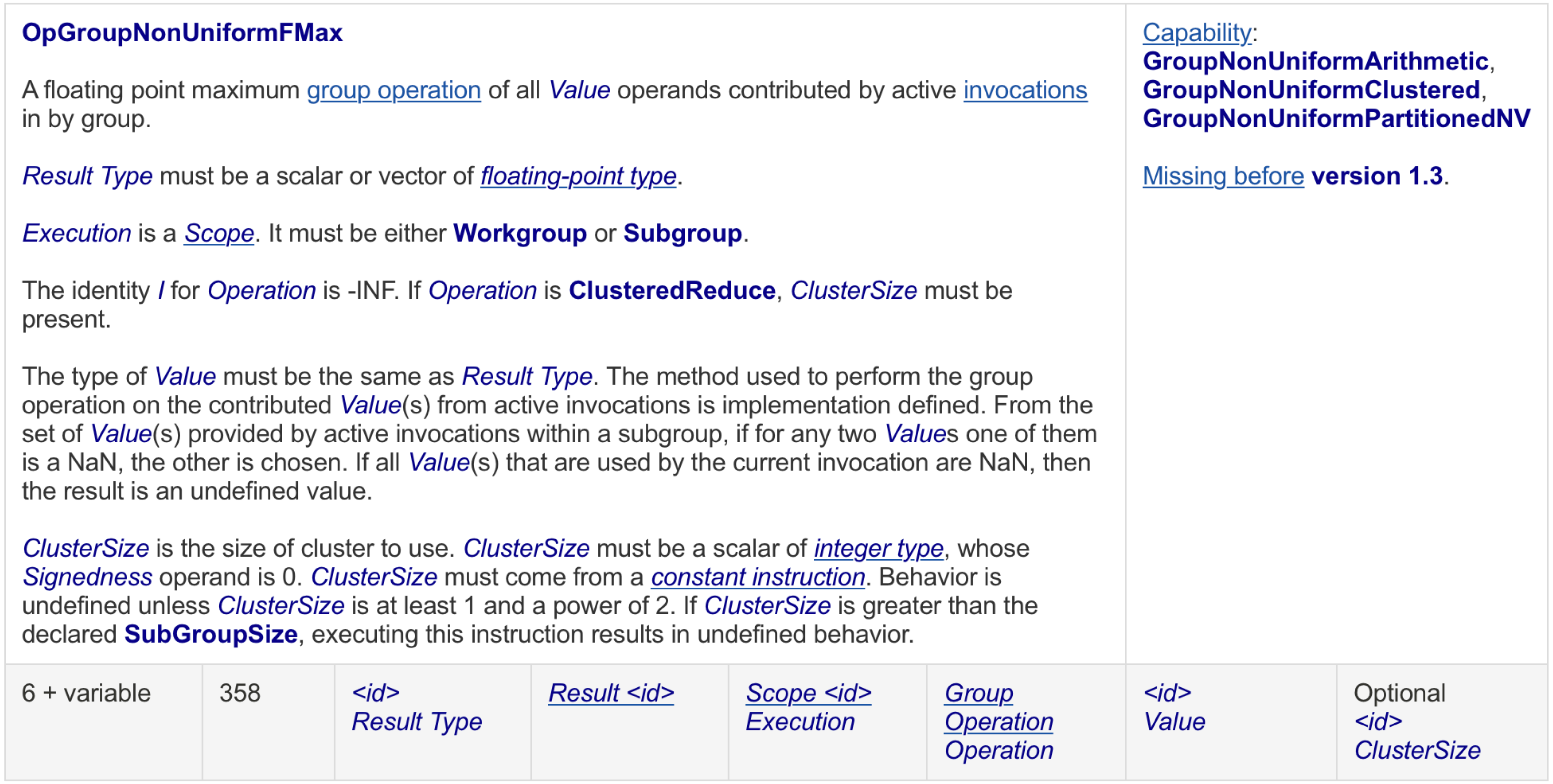
Separation in “what” and “how” we compute things

- how we compute — the red part
  -> build on top of data

- what we compute — the blue part
  -> APL programmers excel here



- weirdly enough GPU people have created instructions for APL operations already
- … but using the correct instruction with correct parameters requires compilation



OpGroupNonUniformFMax

A floating point maximum group operation of all *Value* operands contributed by active invocations in by group.

Result Type must be a scalar or vector of *floating-point type*.

Execution is a *Scope*. It must be either **Workgroup** or **Subgroup**.

The identity *I* for *Operation* is -INF. If *Operation* is **ClusteredReduce**, *ClusterSize* must be present.

The type of *Value* must be the same as *Result Type*. The method used to perform the group operation on the contributed *Value*(s) from active invocations is implementation defined. From the set of *Value*(s) provided by active invocations within a subgroup, if for any two *Value*s one of them is a NaN, the other is chosen. If all *Value*(s) that are used by the current invocation are NaN, then the result is an undefined value.

ClusterSize is the size of cluster to use. *ClusterSize* must be a scalar of *integer type*, whose *Signedness* operand is 0. *ClusterSize* must come from a *constant instruction*. Behavior is undefined unless *ClusterSize* is at least 1 and a power of 2. If *ClusterSize* is greater than the declared **SubGroupSize**, executing this instruction results in undefined behavior.

Capability:
GroupNonUniformArithmetic, **GroupNonUniformClustered**, **GroupNonUniformPartitionedNV**

Missing before **version 1.3**.

6 + variable	358	*<id>* *Result Type*	*Result <id>*	*Scope <id>* *Execution*	*Group Operation* *Operation*	*<id>* *Value*	Optional *<id>* *ClusterSize*

About me

My APL short-story

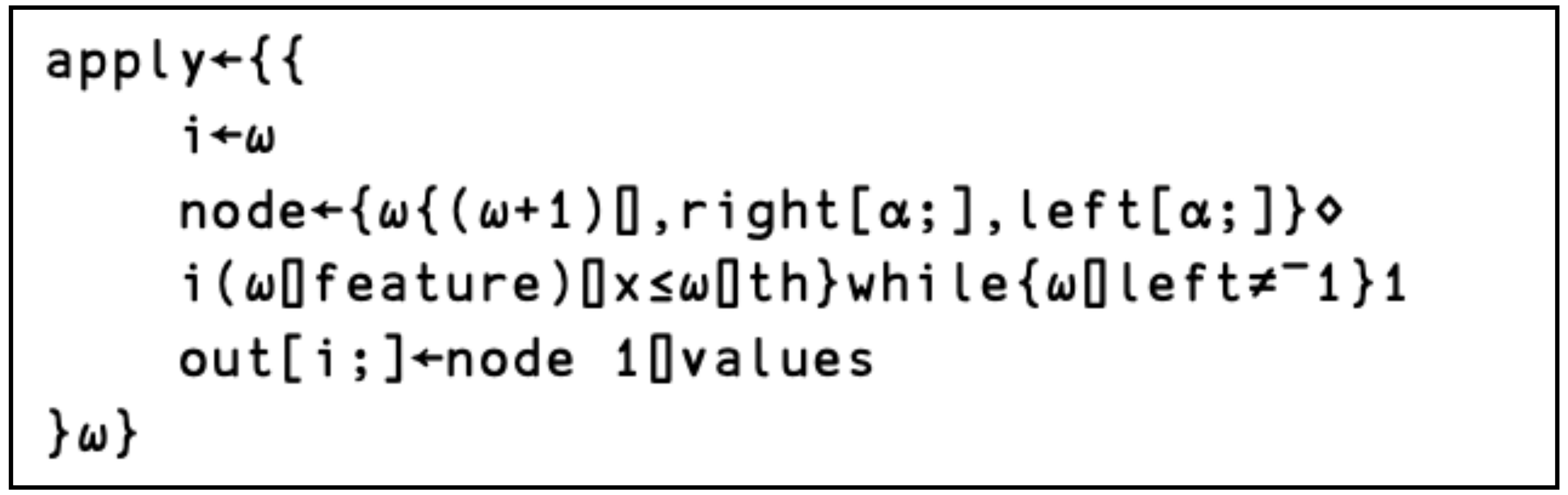
- learned about APL in 2018
- actually looked into it in 2019
- => love on first IDE (Dyalog)
- highly influenced by Aaron Hsu's YouTube talks and thesis

ELEMENTARY ALGEBRA

IBM PHILADELPHIA SCIENTIFIC CENTER
Data Processing Division
TECH. REPORT NO. 320-3001 JUNE 1971

K. E. IVERSON

Seemingly, I first learned about APL from this PDF book: Elementary Algebra, K.E. Iverson

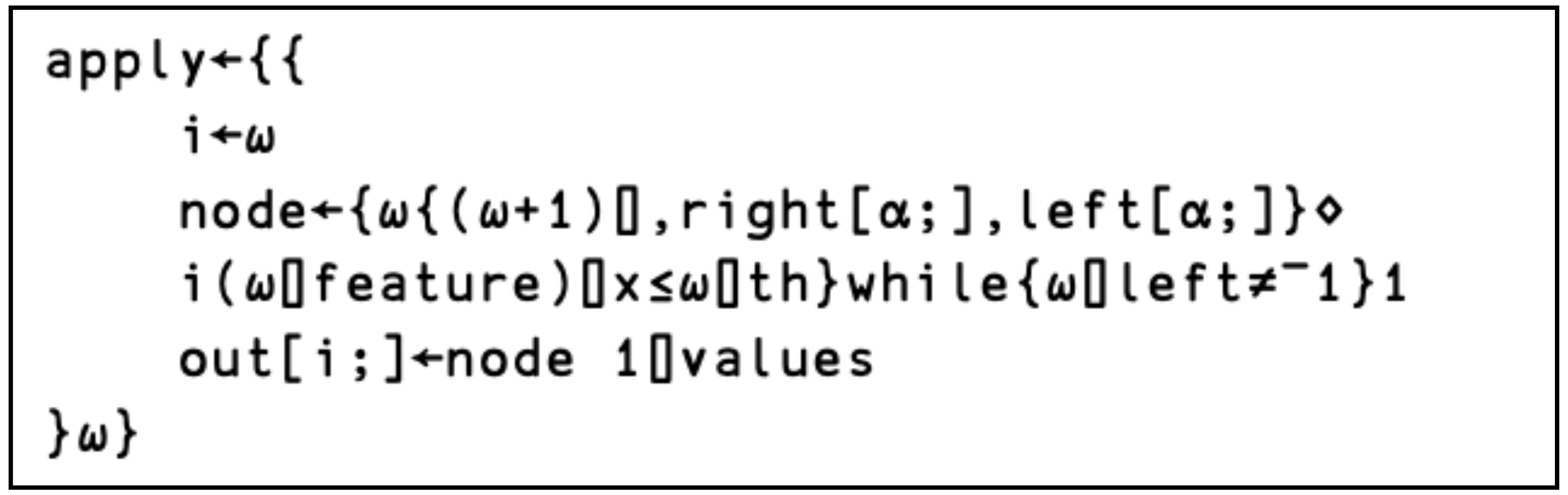

```
apply←{{
    i←ω
    node←{ω{(ω+1)⌷,right[α;],left[α;]}⋄
    i(ω⌷feature)⌷x≤ω⌷th}while{ω⌷left≠¯1}1
    out[i;]←node 1⌷values
}ω}
```

My first APL program: random forest traversal

Case study: random forest prediction

Performance aspect

- Modelling of Python scikit with APL
- Translating APL to SPIR-V
- Running SPIR-V with Vulkan



```
apply←{{
    i←ω
    node←{ω{(ω+1)⌷,right[α;],left[α;]}⋄
    i(ω⌷feature)⌷x≤ω⌷th}while{ω⌷left≠¯1}1
    out[i;]←node 1⌷values
}ω}
```

Random forest binary tree traversal in APL

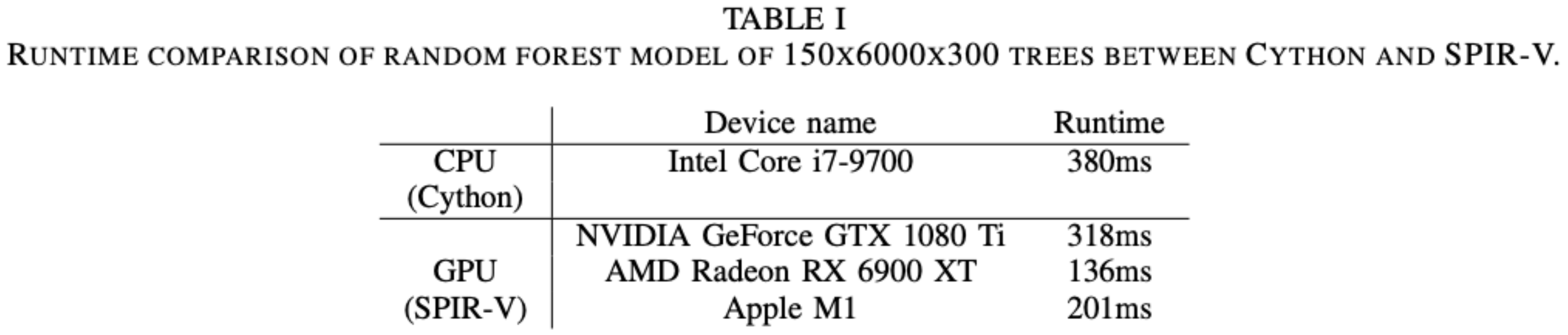


TABLE I
RUNTIME COMPARISON OF RANDOM FOREST MODEL OF 150X6000X300 TREES BETWEEN CYTHON AND SPIR-V.

	Device name	Runtime
CPU (Cython)	Intel Core i7-9700	380ms
GPU (SPIR-V)	NVIDIA GeForce GTX 1080 Ti	318ms
	AMD Radeon RX 6900 XT	136ms
	Apple M1	201ms

Results for small programs, memory copying remains the big bottleneck

Findings from the GPU world

- **what we compute**: APL programmers already think about data in a way that multi-core devices would want all programmers to think
- **how we compute**: machine-solvable drudgery that builds on top of the data representation
- **challenge**: how do we automate the **how**?

APL x Academics

Hot topics:

- **static rank polymorphism** (Remora @ Northeastern, Futhark @ Copenhagen, Dex @ Google)
- application area of **dependent types** (Idris @ St Andrews, Granule @ Kent)
- functional programming for **tensor computation** (Halide @ MIT / Adobe)
- an approach to simplify **parallel computation** (CUDA & Legate @ Nvidia, Matlab, Julia, Numpy, TensorFlow … etc. machine learning applications)

Software bugs
and where to find them

- main challenge: **recursion**
- at distance (i.e., abstractly) the waterfall makes no sense...
- ... but software will not realize something is wrong, unless we define **constraints** which describe how to build a waterfall
- => the need for **abstract interpretation**

Waterfal (Escher, 1961)

Types remove ambiguity

How problematic recursion can be caught

- types may express, **ownership**, **direction**, **multiplicities**

- linear types express **ownership**
  Finnish “koiran” - “dog’s”

- dependent types express **direction**
  Finnish “koirastani” - “from my dog”

- quantitative types tie **multiplicity** into ownership
  Finnish “koirillanikin” - “(something) that also my dogs have”

Typed waterfall

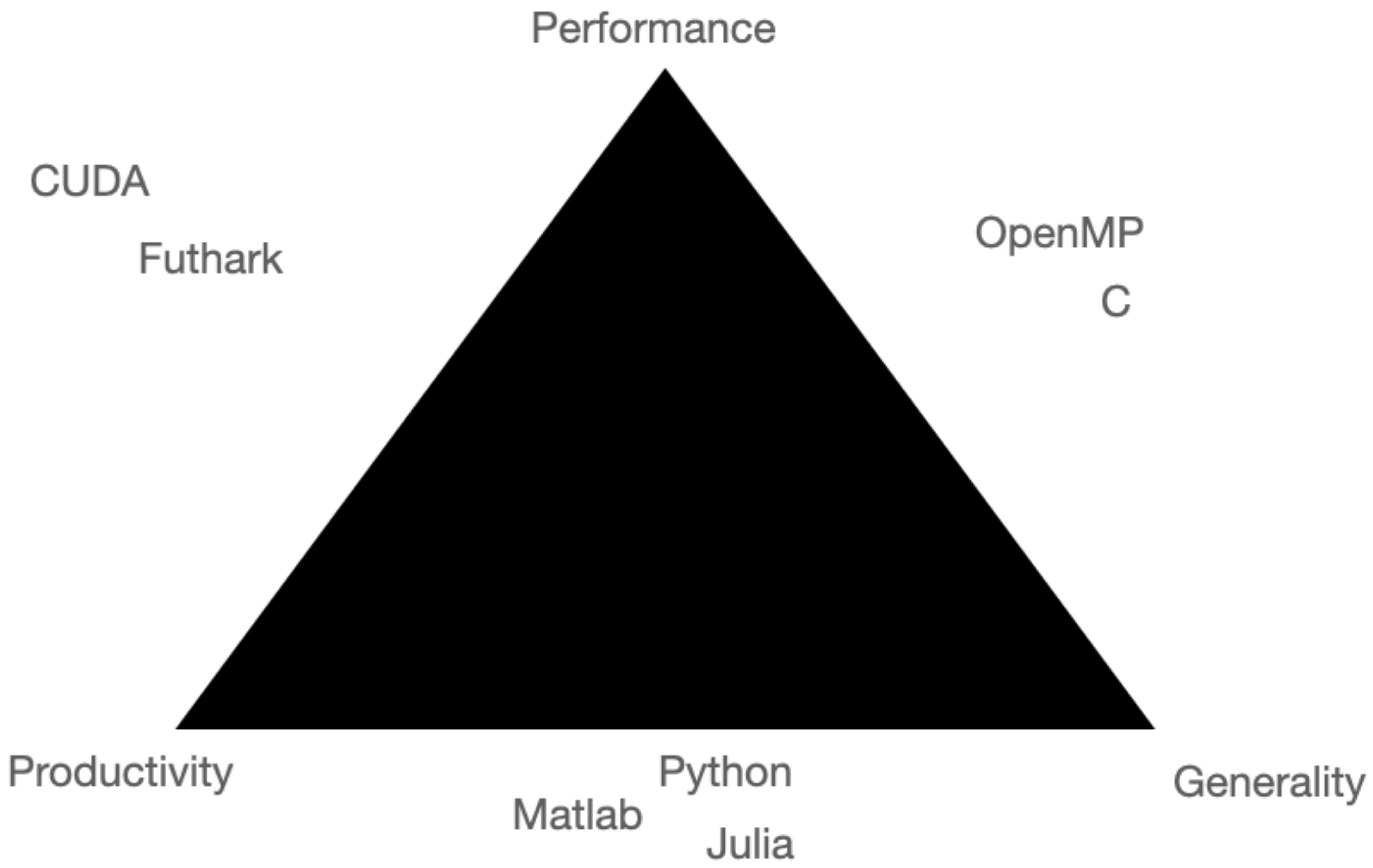
- types may express, **ownership**, **direction**, **multiplicities**

- what is the direction of the water?
- where does the water come from?
- can the water be re-used?
- answering these questions **constraints** the ways that a waterfall may be built
- constraints are **guides** in an otherwise random search for a solution

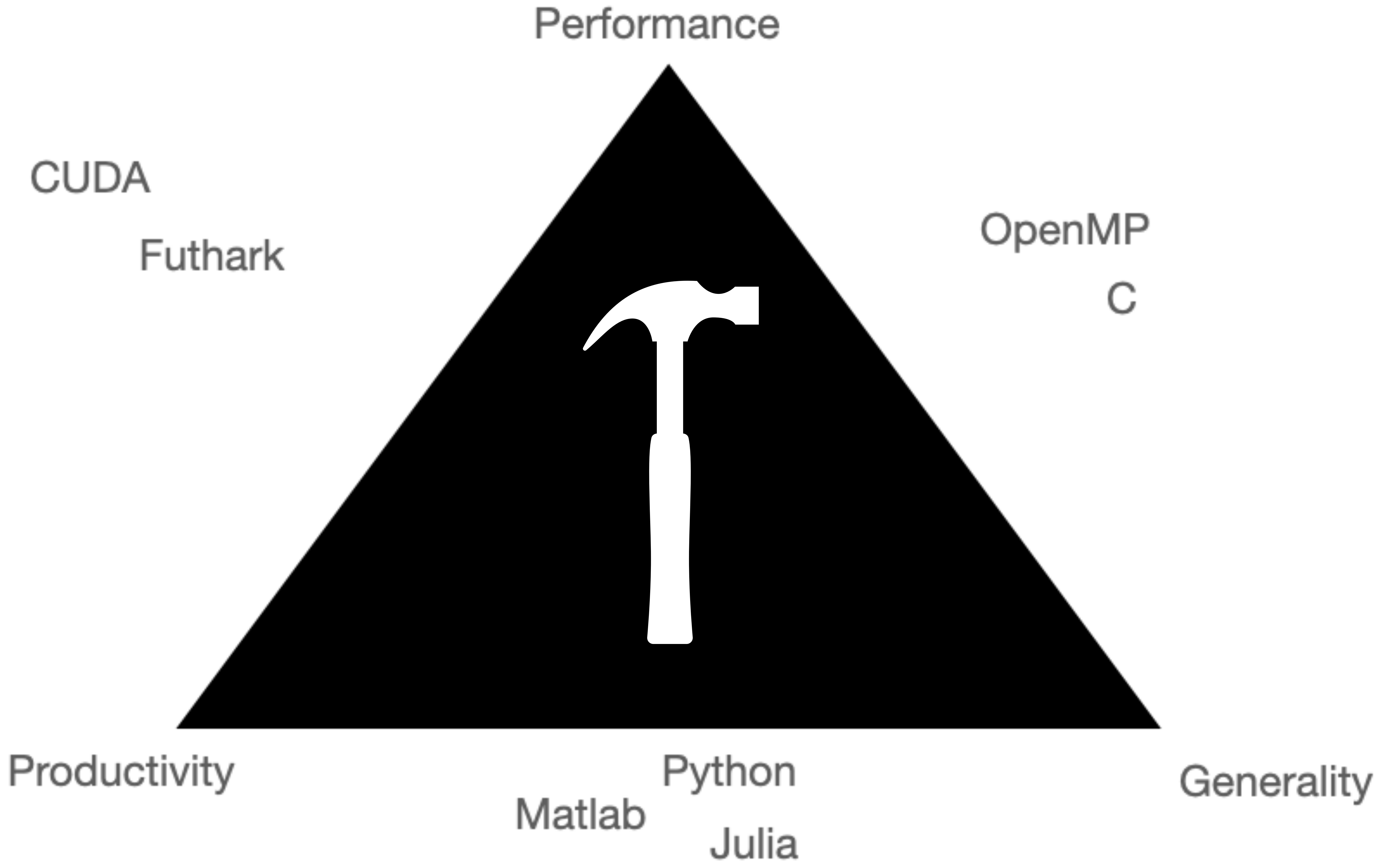
Waterfal (Escher, 1961)

Types are changing
… but so are computers

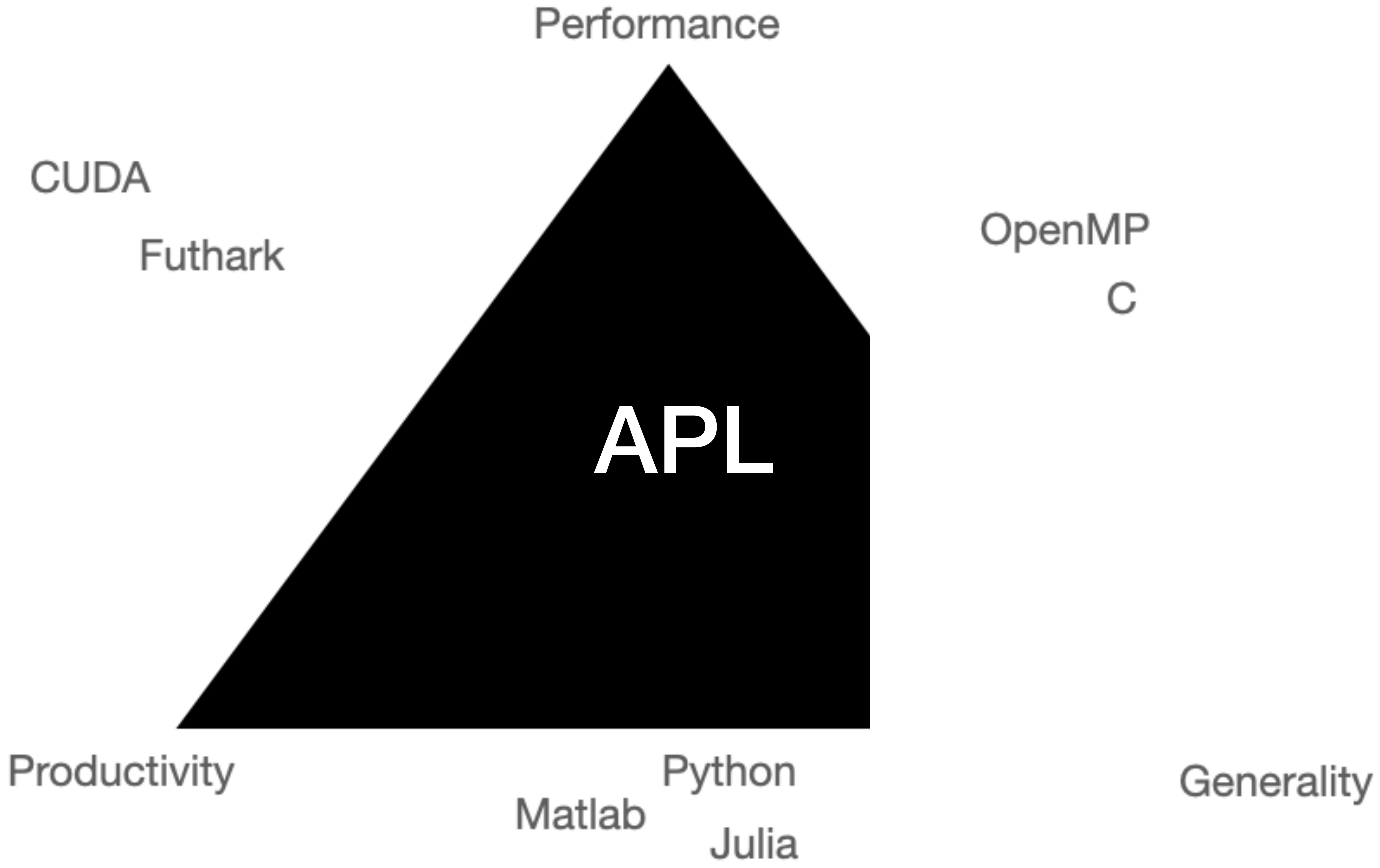
The Language Trilemma



The Language Trilemma



The Language Trilemma



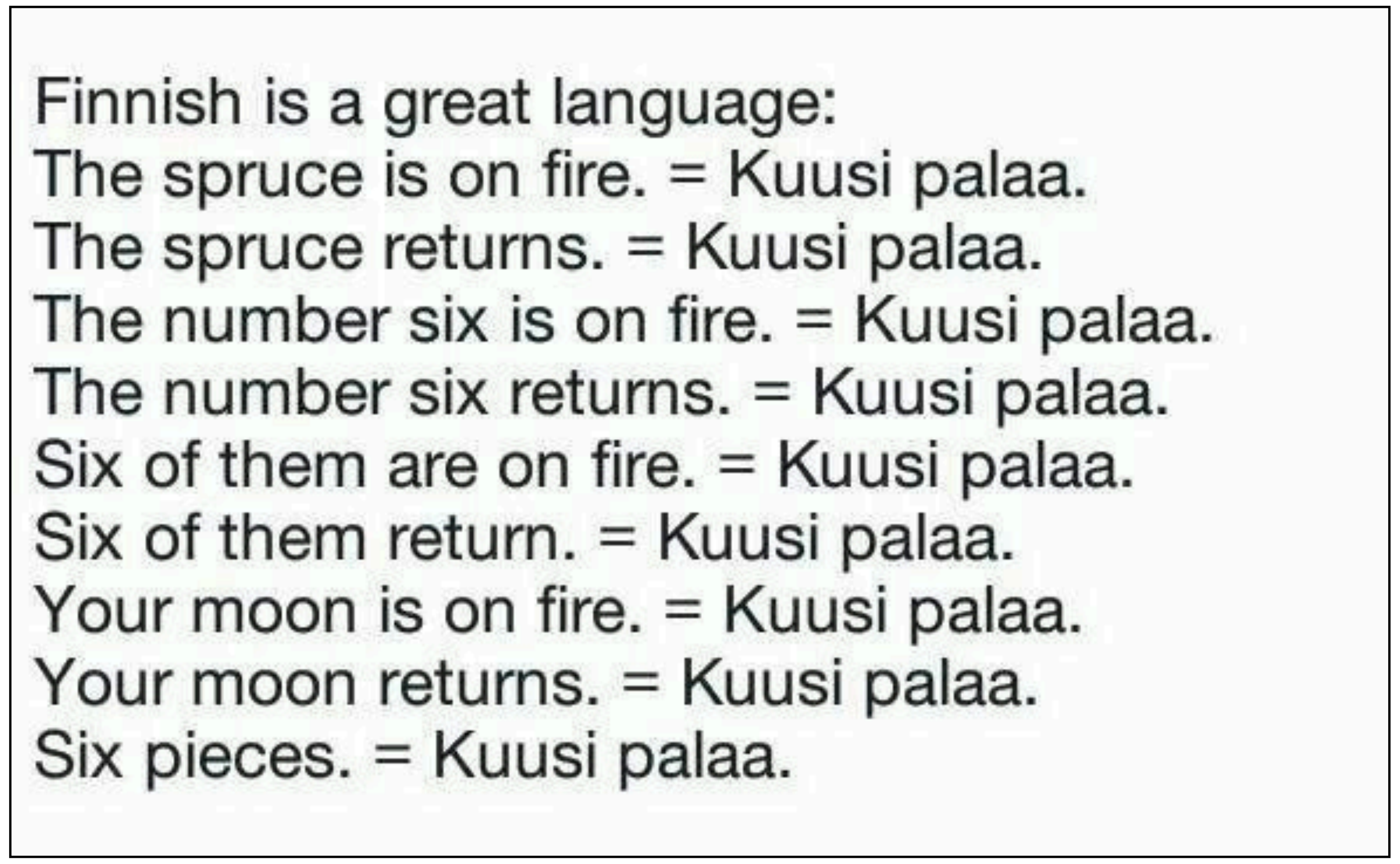
Q: What can we gain by losing generality?

A: Static rank polymorphism

- static semantics
- - "abstract interpretation"
- - what can we know before we start a program

- rank polymorphism
- - adds value-based "context" to the language interpretation

Gödel's incompleteness theorem: it is hard to interpret even simple programs



Finnish is a great language:
The spruce is on fire. = Kuusi palaa.
The spruce returns. = Kuusi palaa.
The number six is on fire. = Kuusi palaa.
The number six returns. = Kuusi palaa.
Six of them are on fire. = Kuusi palaa.
Six of them return. = Kuusi palaa.
Your moon is on fire. = Kuusi palaa.
Your moon returns. = Kuusi palaa.
Six pieces. = Kuusi palaa.

A certain kind of rank polymorphism in action

Q: Why is static rank polymorphism useful?

A: It simplifies *Parallel Programming*

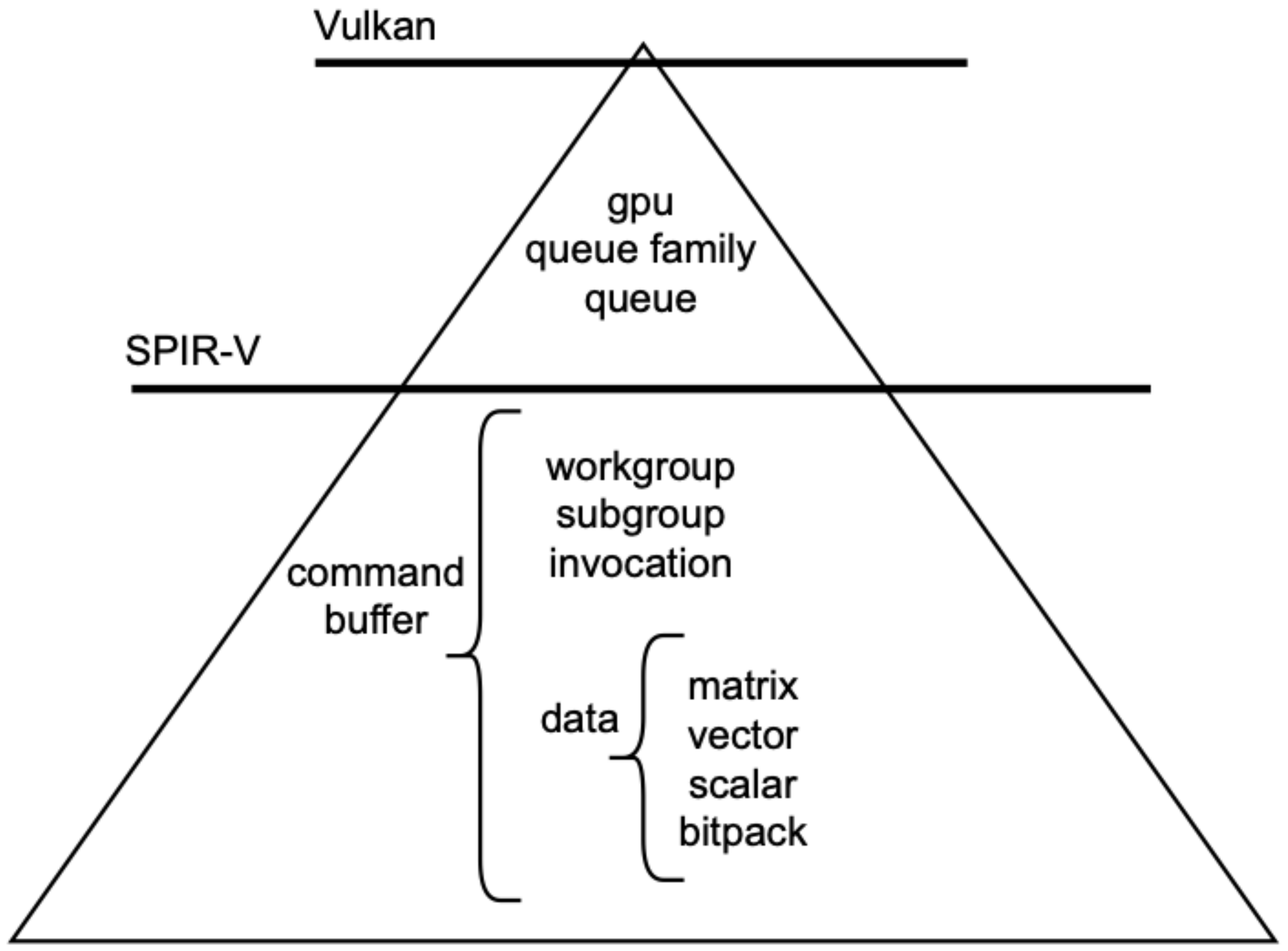
- Accumulators 👎
  divide-and-conquer 👍
- However, dividing and conquering is hard when you don't know the amount of "troops"
- Remark: array programming languages abstract away the execution strategy

GPU	Subgroup size	Queue families	Queue lengths
Intel	8/16/32	?	?
AMD	64	3	16/8/1
Nvidia	32	3	16/8/1
Adreno	64	?	?
Mali	16	?	?
Apple M1	32	4	1/1/1/1

Varying "troops" on GPUs => need for dynamic scheduling

Putting it together

- Challenge: understanding what **shapes** data may have
- Needed for: **constraints** which build the rest of the pyramid for us
- => can be achieved with shape analysis by employing new type systems
- “the APL way” remains — data-driven, no unnecessary software ceremonies



Shape analysis

To always know how many “troops” we have!

- practicality
- - typing for GPUs: strong typing facilitates work splitting — efficiency
- - background: nice to have a single program work for any GPU

- theoretical
- - advanced type system applications: what can we know beforehand
- - array programming: how can we generalize, adapt the information

Parallelism is much easier with abstract interpretation (“knowing your troops”)

Scheduling Array Operations

Juuso Haavisto (University of Oxford), October 2022

Static Semantics

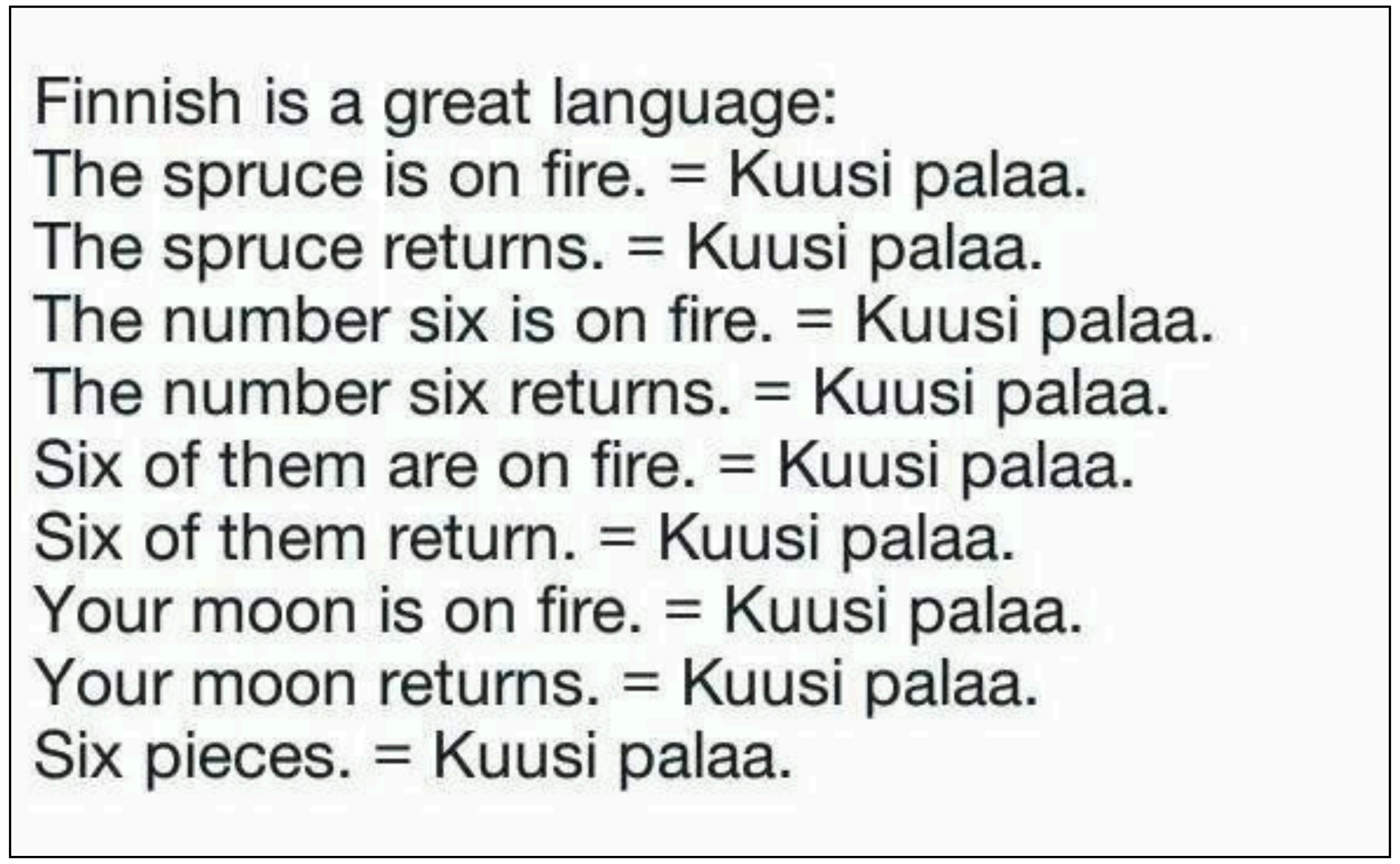
Ability to detect correctness errors before execution

Statically absurd situation:
Waterfal (Escher, 1961)

Rank Polymorphism

Language property where semantically same thing means multiple things depending on the context

Challenge: building connection between static semantics

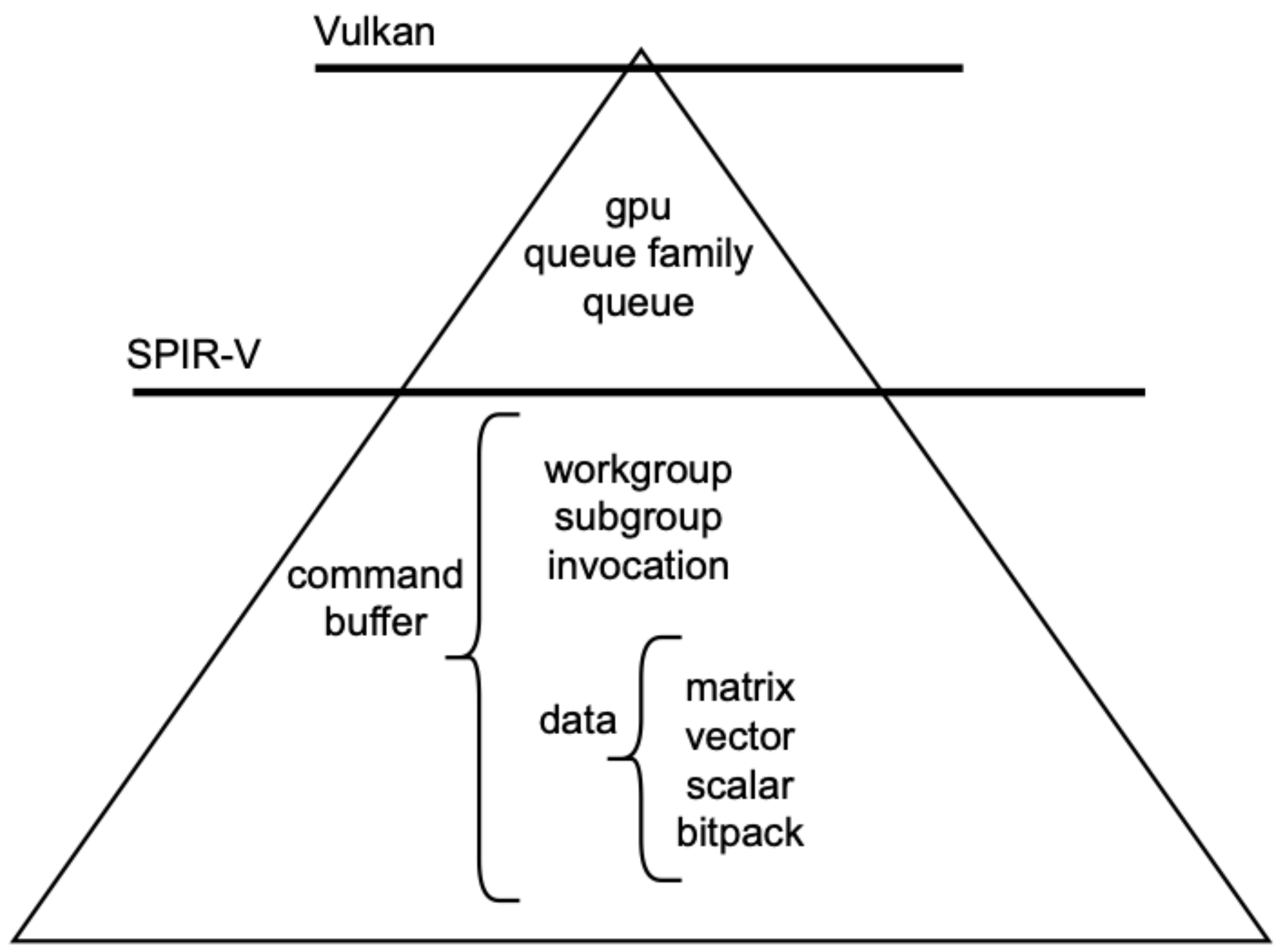


Finnish is a great language:
The spruce is on fire. = Kuusi palaa.
The spruce returns. = Kuusi palaa.
The number six is on fire. = Kuusi palaa.
The number six returns. = Kuusi palaa.
Six of them are on fire. = Kuusi palaa.
Six of them return. = Kuusi palaa.
Your moon is on fire. = Kuusi palaa.
Your moon returns. = Kuusi palaa.
Six pieces. = Kuusi palaa.

A certain kind of rank polymorphism in action

Scheduling

With static rank polymorphism, we can give GPU a data structure that it understands, and from which to derive scheduling schemes for multi-core use



Hierarchy of GPU

Takeaway

- APLers already think in the way that new hardware wants us to …
- … however, the languages must "see" things like APLers do
- => types can help the computer to constrain, search its way out to "see"
- => this way, the types build **on top** of the APL arrays

- **let the types work for us, not the other way around**
- => performance optimizations on multi-core systems, such as GPUs (SIMD use)
- => automatic distribution (divide and conquer strategies)

Questions?