

# *The Future and what to do about it: Processors, People, and Programming*



Tim Mattson

The Human Learning Group plus an honorary professor at the University of Bristol

Worked at Intel from 1993 to 2023

[tim@timmattson.com](mailto:tim@timmattson.com)

**Predicting the future of hardware is easy**

# Follow the Money

- The five hyperscalars are in control ... they ask from their vendors and they receive (discounts, design changes, etc.).

1. Amazon web services
2. Microsoft Azure
3. Google Cloud
4. IBM Cloud
5. Apple

If you are in the microprocessor business, the fact each of these companies are now designing their own processors is VERY scary.

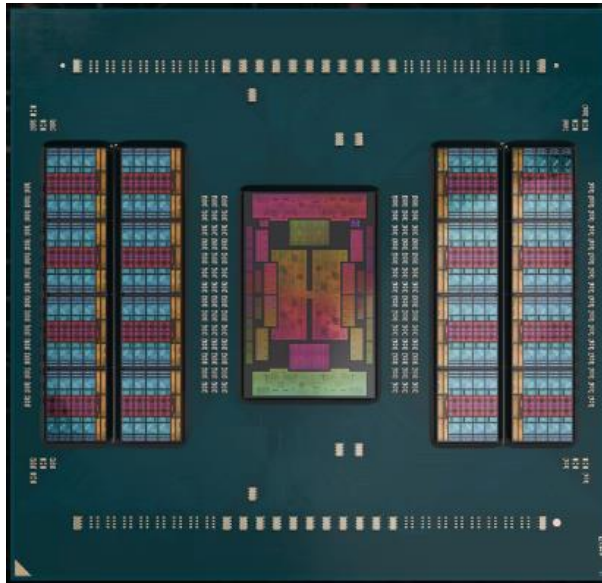
- Nvidia has a temporary “near-monopoly” giving them some power over the hyperscalars, but it is temporary (AMD GPUs are great and Intel will figure it out eventually).
- Chip vendors will do what it takes to maximize profit (i.e., keep hyperscalars happy, accelerate designs, and reduce costs).

Since the early days when Cray had a monopoly, the HPC community has always fed off the table scraps from industry.

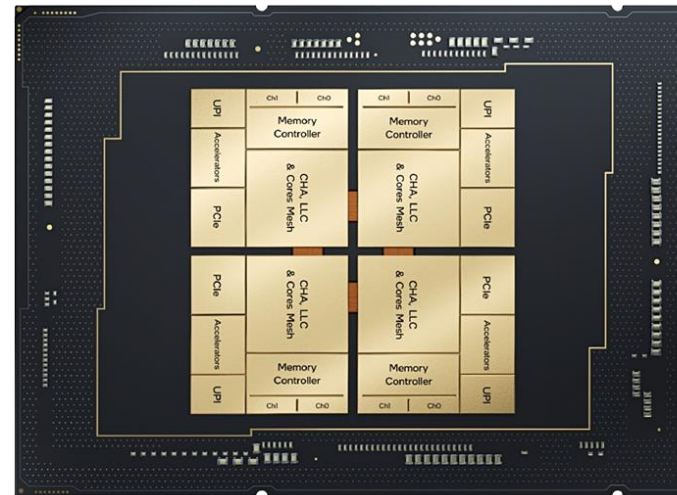
Government spending (e.g. the ASCI program in the 1990's) sometimes offsets this effect, but it never lasts long enough at significant levels to really matter.

**The heart of computing ... microprocessors**

# It used to be so easy ... x86 ruled



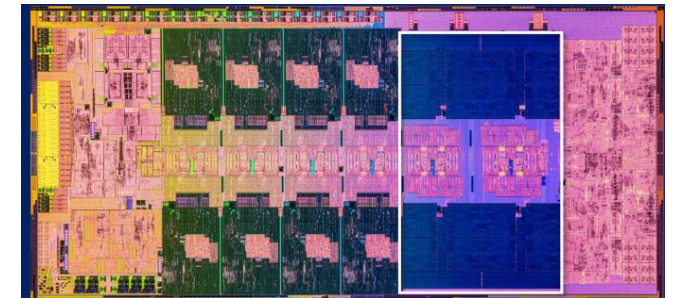
AMD® EPYC™ 9005 Server CPU  
with 128 Zen 5 cores



4<sup>th</sup> Gen Intel® Xeon® CPU with 56  
cores and Novel On-Die  
Accelerators

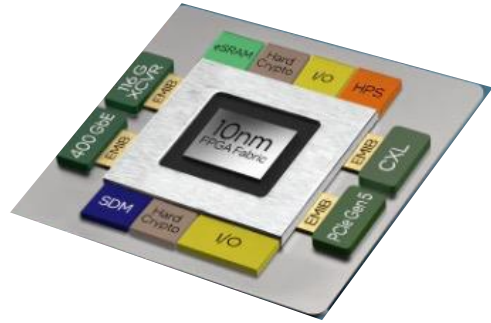


AMD® Ryzen™ 9 5900X  
Desktop CPU with 12 cores

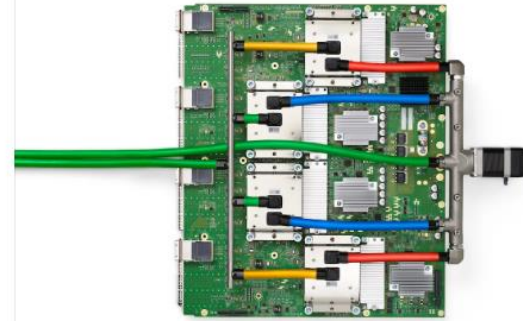


13<sup>th</sup> gen Intel® Core™ CPU  
Hybrid Architecture with 16 efficiency cores and 8  
performance cores + integrated GPU

# Now ... Diversity rules



Intel® Agilex™ FPGAs



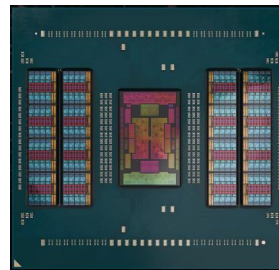
Google® Tensor Processing Unit



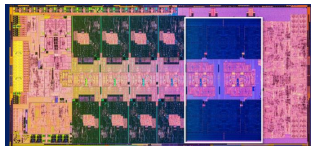
Habana® Gaudi® 2 deep learning accelerator



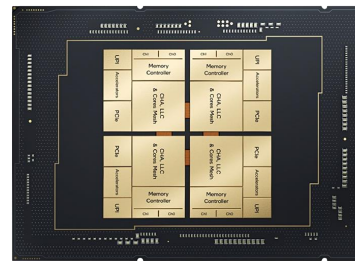
AMD® Ryzen™ 9 5900X Desktop CPU with 12 cores



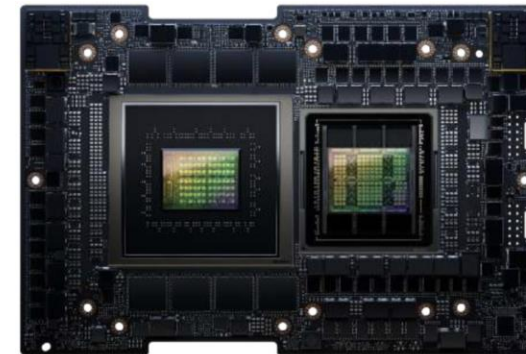
AMD® EPYC™ 9005 Server CPU with 128 Zen 5 cores



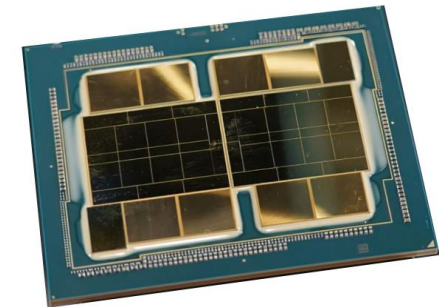
13<sup>th</sup> gen Intel® Core™ CPU Hybrid Architecture with 16 efficiency cores and 8 performance cores + integrated GPU



4<sup>th</sup> Gen Intel® Xeon® CPU with 56 cores and Novel On-Die Accelerators



NVIDIA® Grace Hopper™ superchip 72 Arm Neoverse v2 CPUs and GH200 GPU



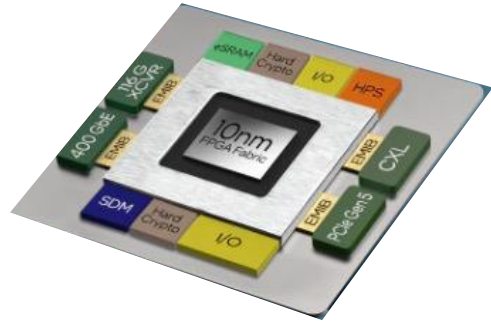
Intel® Xe HPC GPU



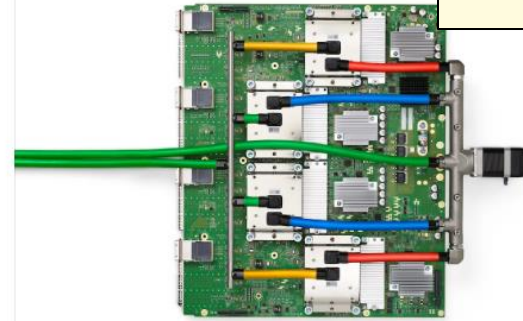
Including RISC architectures we used to ignore



# Now ... Diversity rules



Intel® Agilex™ FPGAs



Google® Tensor Processing Unit

And now that the x86 “monopoly” is over, all my explicitly vectorized code is broken!!

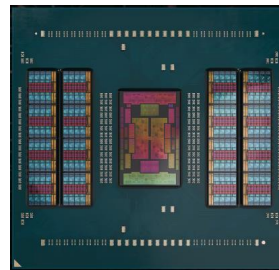
Including RISC architectures we used to ignore



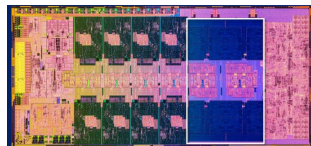
Habana® Gaudi® 2 deep learning accelerator



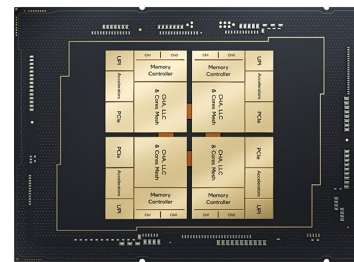
AMD® Ryzen™ 9 5900X Desktop CPU with 12 cores



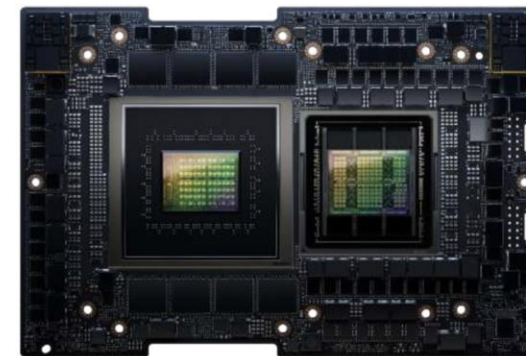
AMD® EPYC™ 9005 Server CPU with 128 Zen 5 cores



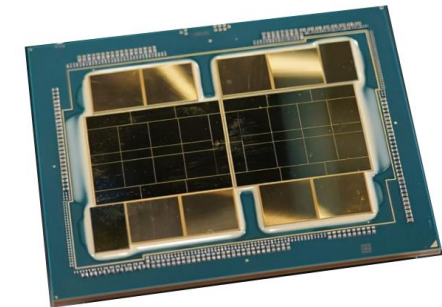
13<sup>th</sup> gen Intel® Core™ CPU Hybrid Architecture with 16 efficiency cores and 8 performance cores + integrated GPU



4<sup>th</sup> Gen Intel® Xeon® CPU with 56 cores and Novel On-Die Accelerators



NVIDIA® Grace Hopper™ superchip 72 Arm Neoverse v2 CPUs and GH200 GPU



Intel® Xe HPC GPU

**... and its about to get MUCH "worse"**



# What does the future hold? Get ready for Chiplets ...

Chiplet: A distinct silicon-device packaged with other chiplets to create the processor you drop in a socket

2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)

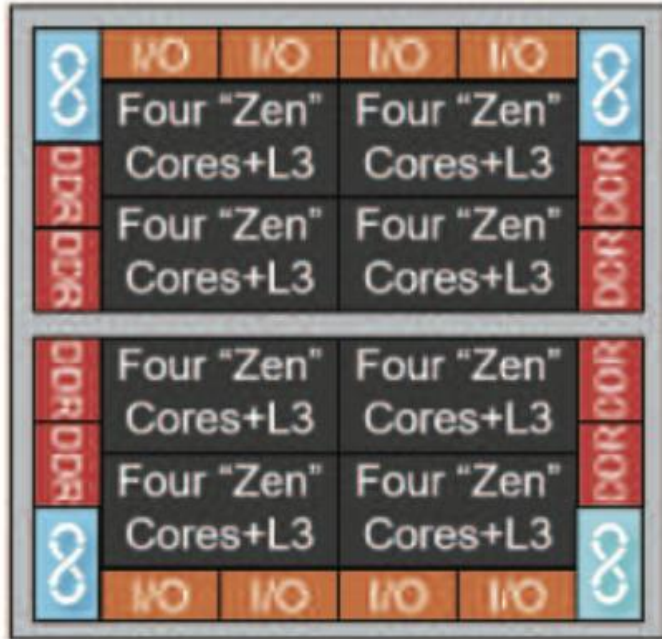
## **Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families**

Industrial Product

Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, Sean White  
Advanced Micro Devices, Inc.

A really nice case study on the value of chiplets

# Chiptlets: Follow the Money



Monolithic 32-core Chip  
777mm<sup>2</sup> total area  
1.0x Cost

A hypothetical Monolithic chip

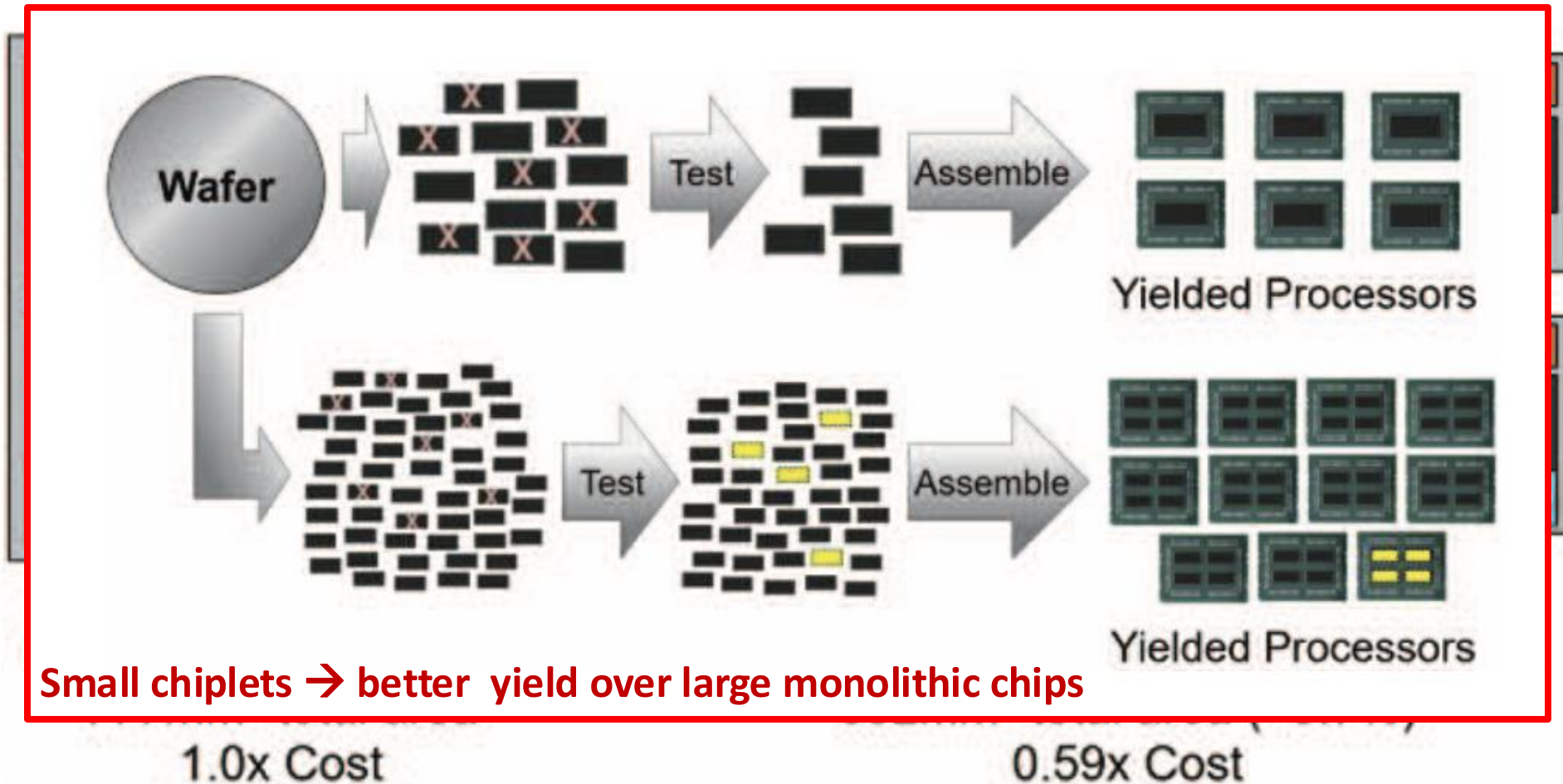


4 x 8-core Chiplet, 213mm<sup>2</sup> per chiplet  
852mm<sup>2</sup> total area (+9.7%)  
0.59x Cost

1<sup>st</sup> generation AMD EPYC™ processor  
built from 4 identical chiplets

 Infinity Fabric™ a coherent interconnect between chiplets

# Chiptlets: Follow the Money



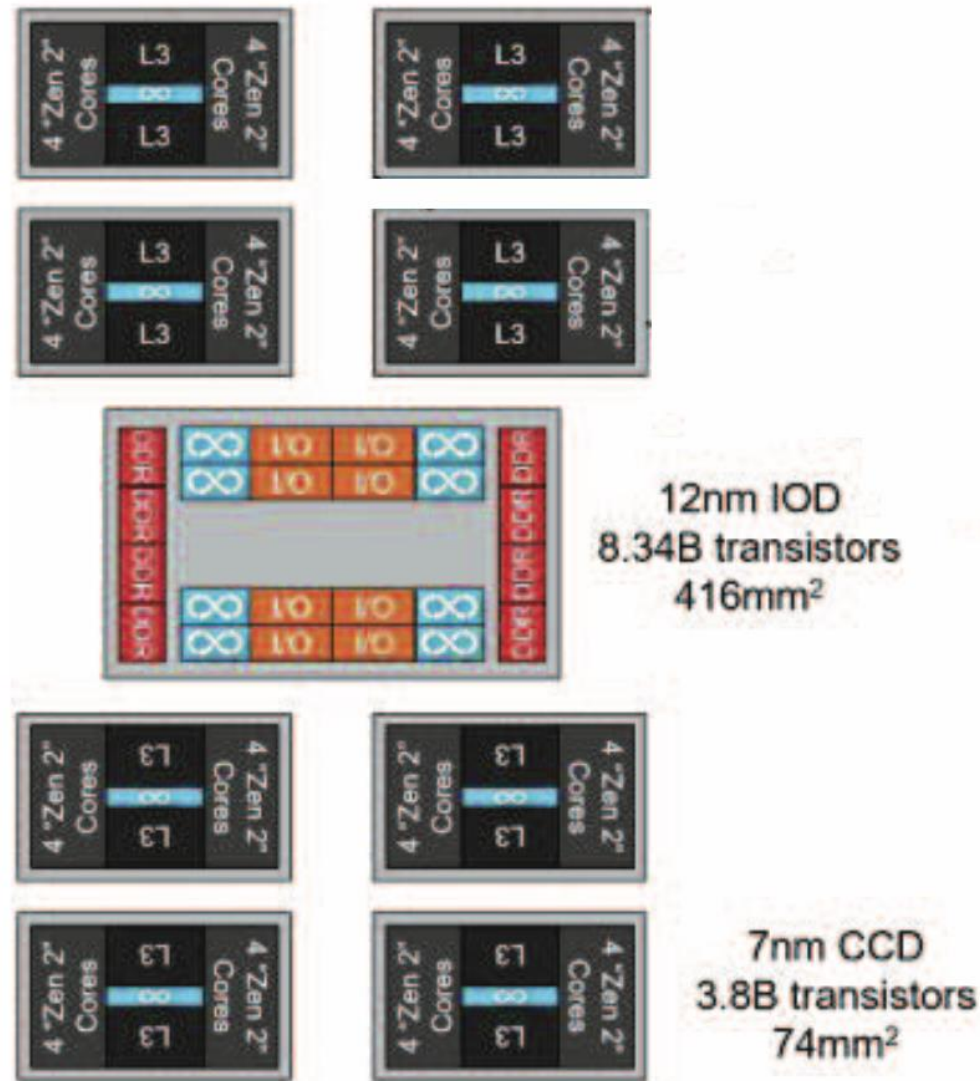
A hypothetical Monolithic chip

1<sup>st</sup> generation AMD EPYC™ processor  
built from 4 identical chiptlets

 Infinity Fabric™ a coherent interconnect between chiptlets

# Chiplets: The value of mixing chip fabrication nodes

16 Zen 2™ cores that benefit from the top-end technology node .... While the I/O Die (IOD) uses less expensive and mature 12 nm technology.

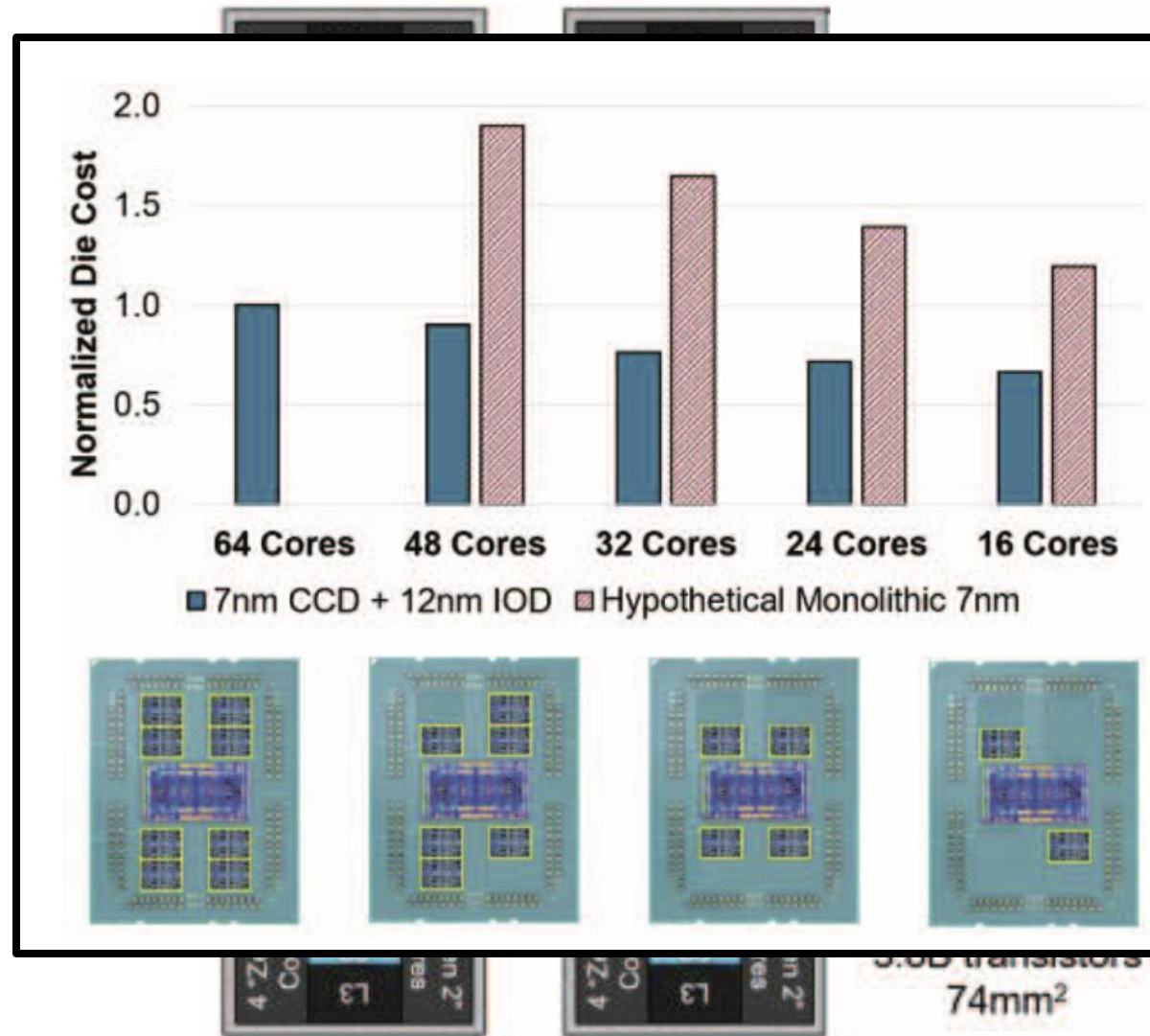


Matching the chiplet to the most effective technology node is a potentially huge cost savings

2<sup>nd</sup> Generation AMD EPYC™ processor

# Chiplets: The value of mixing chip fabrication nodes

16 Zen 2™ cores that benefit from the top-end technology node .... While the I/O Die (IOD) uses less expensive and mature 12 nm technology.



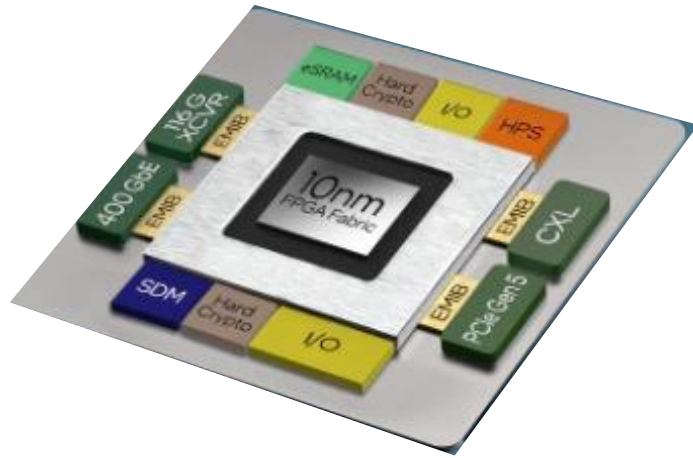
Matching the chiplet to the most effective technology node is a potentially huge cost savings

Support a family of products from two chiplet building blocks at a lower die cost

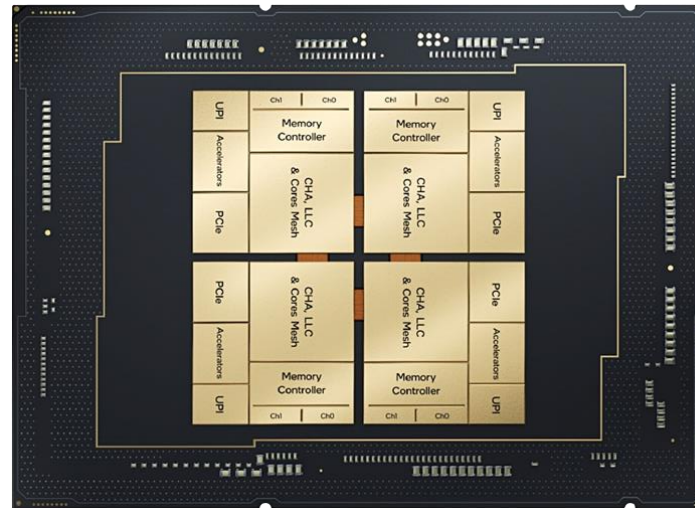
2<sup>nd</sup> Generation AMD EPYC™ processor

# Intel is using chiplet technology as well

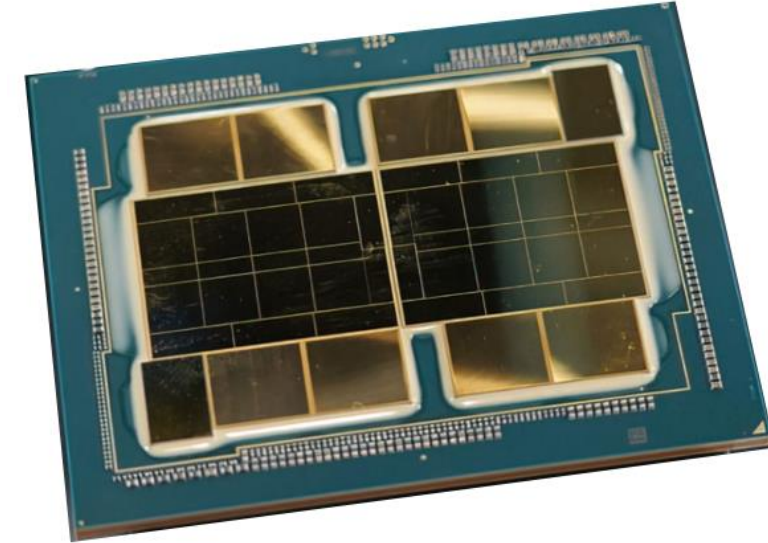
These Intel products use chiplet technology



Intel® Agilex™ FPGAs



4<sup>th</sup> Gen Intel® Xeon® CPU with 56 cores and Novel On-Die Accelerators

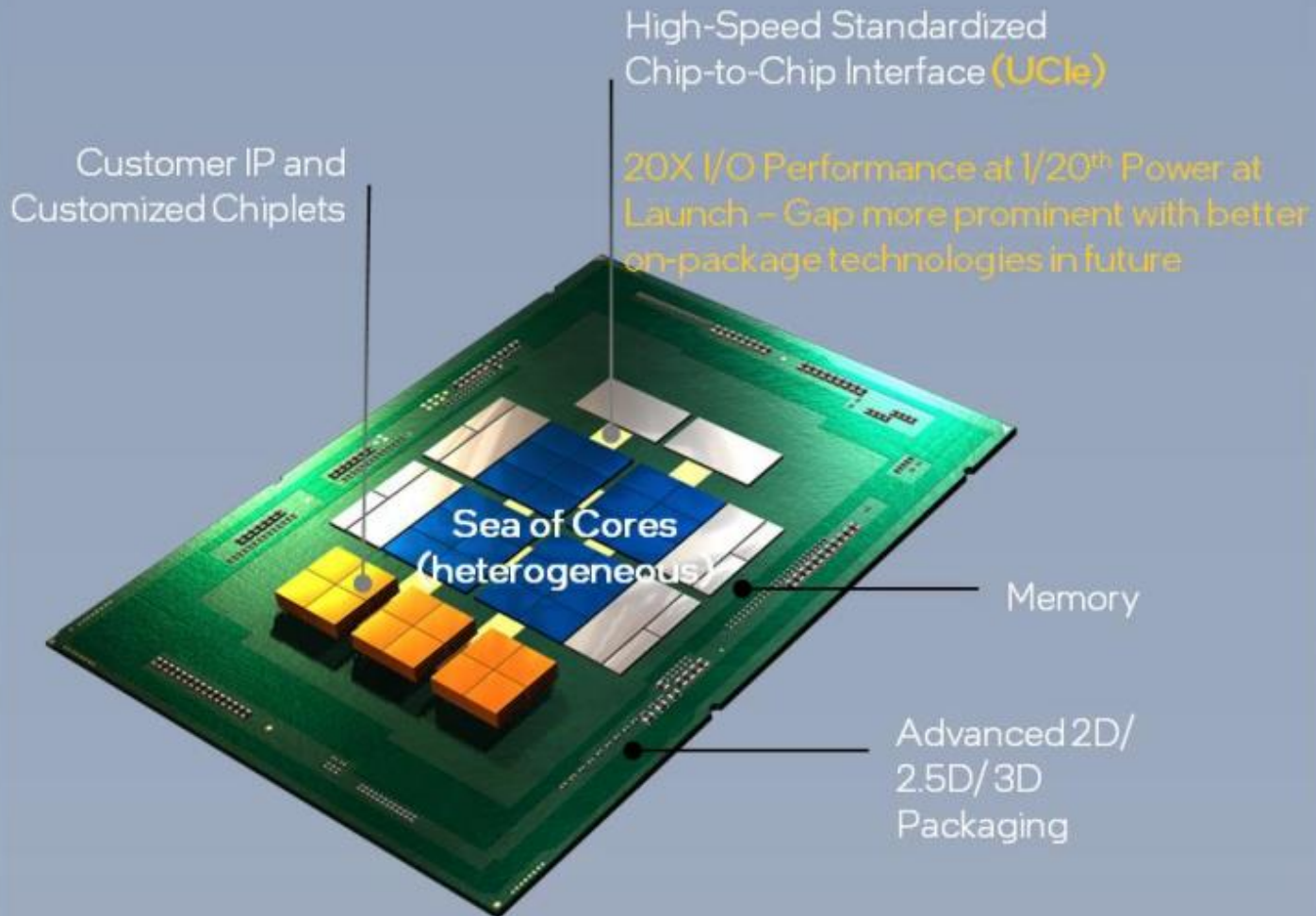


Intel® Xe HPC GPU

# A New Golden Age for Computer Architecture



## Open Chiplet: Platform on a Package



- Chiplet-based architectures ... building a package placed in a socket composed of distinct little chips (the “chiplet”).
- Connected by high speed in package interconnects ... lets chiplets from multiple fabs fit into one package.
- The Universal Chiplet Interconnect Express effort defines a standard for how to connect chiplets.
- The result ... multi-chiplet packages in a socket with heterogeneous devices from multiple vendors.

120+ Member Companies and growing!

# Board Members

Leaders in semiconductors, packaging, IP suppliers, foundries, and cloud service providers are joining together to drive The open chiplet ecosystem.

**JOIN US!**



With a common chiplet standard, we can expect interesting combinations of chiplets across vendors



**With UCle we will have chiplets from multiple vendors using different process technologies in a single socket.**

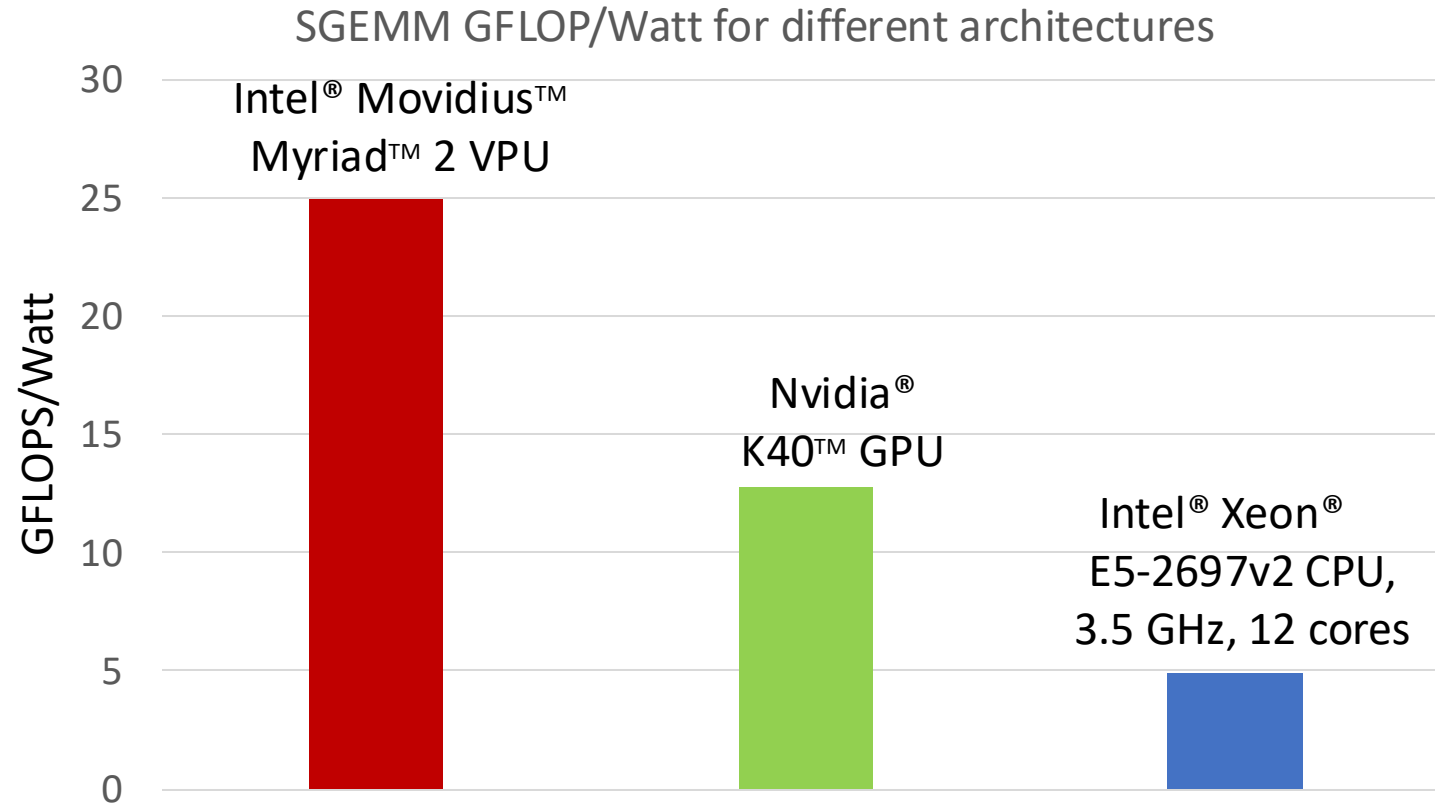
**Heterogenous architectures within a socket will become common**

Developing a code-base for our applications that spans all this heterogeneity will be a real headache.

Can we just ignore it? Just because HW people build stuff, we don't have write code for it, do we?

# If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.



Hence, future systems will be increasingly heterogeneous ... GPUs, CPUs, FPGAs, and a wide range of accelerators

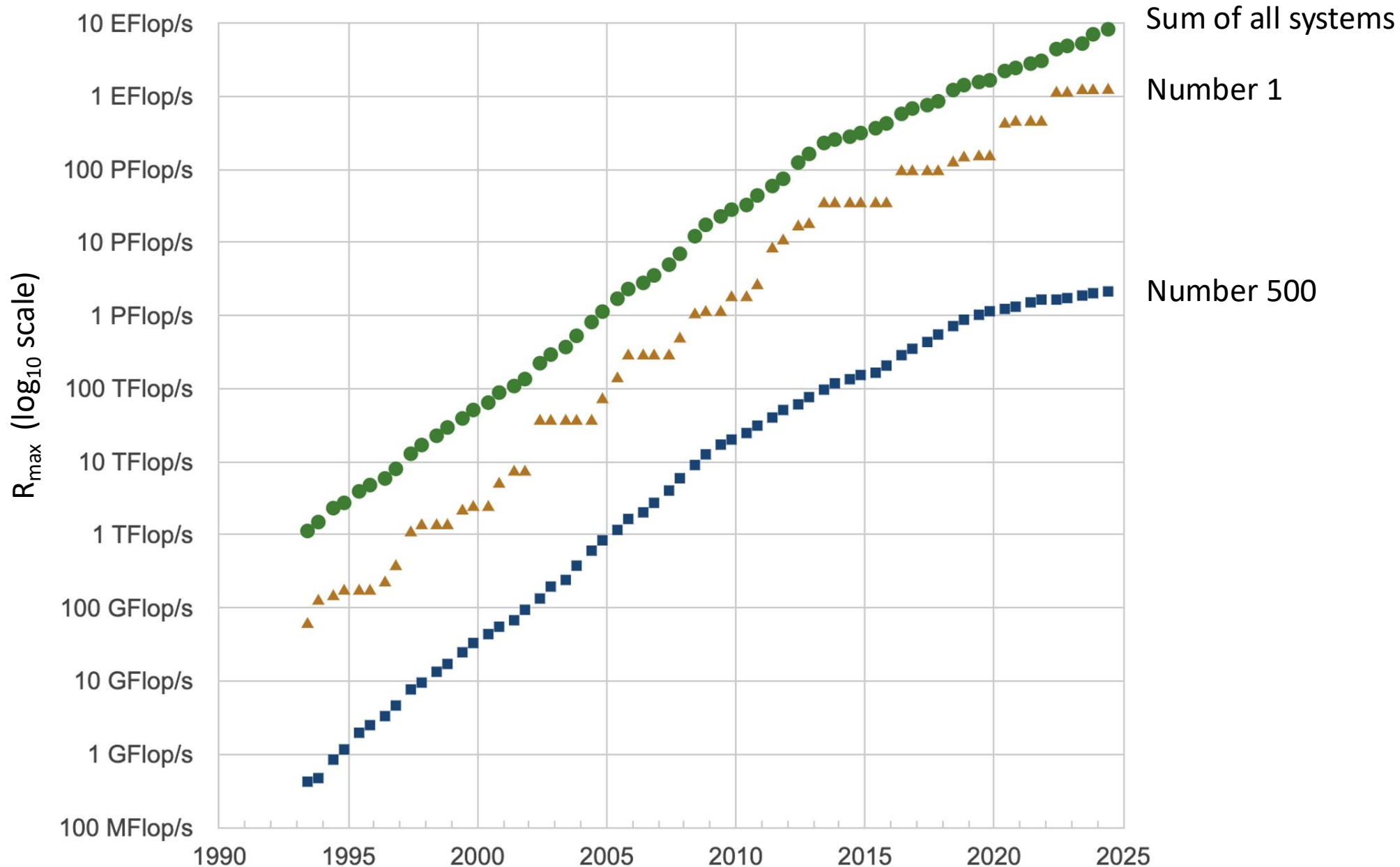
This is all at the level of a socket.

Consider the large scale systems at the heart  
of HPC.

# We all ❤️ the top-500 List

The 500 fastest computers in the world as measured by their ability to solve a dense linear algebra problem.

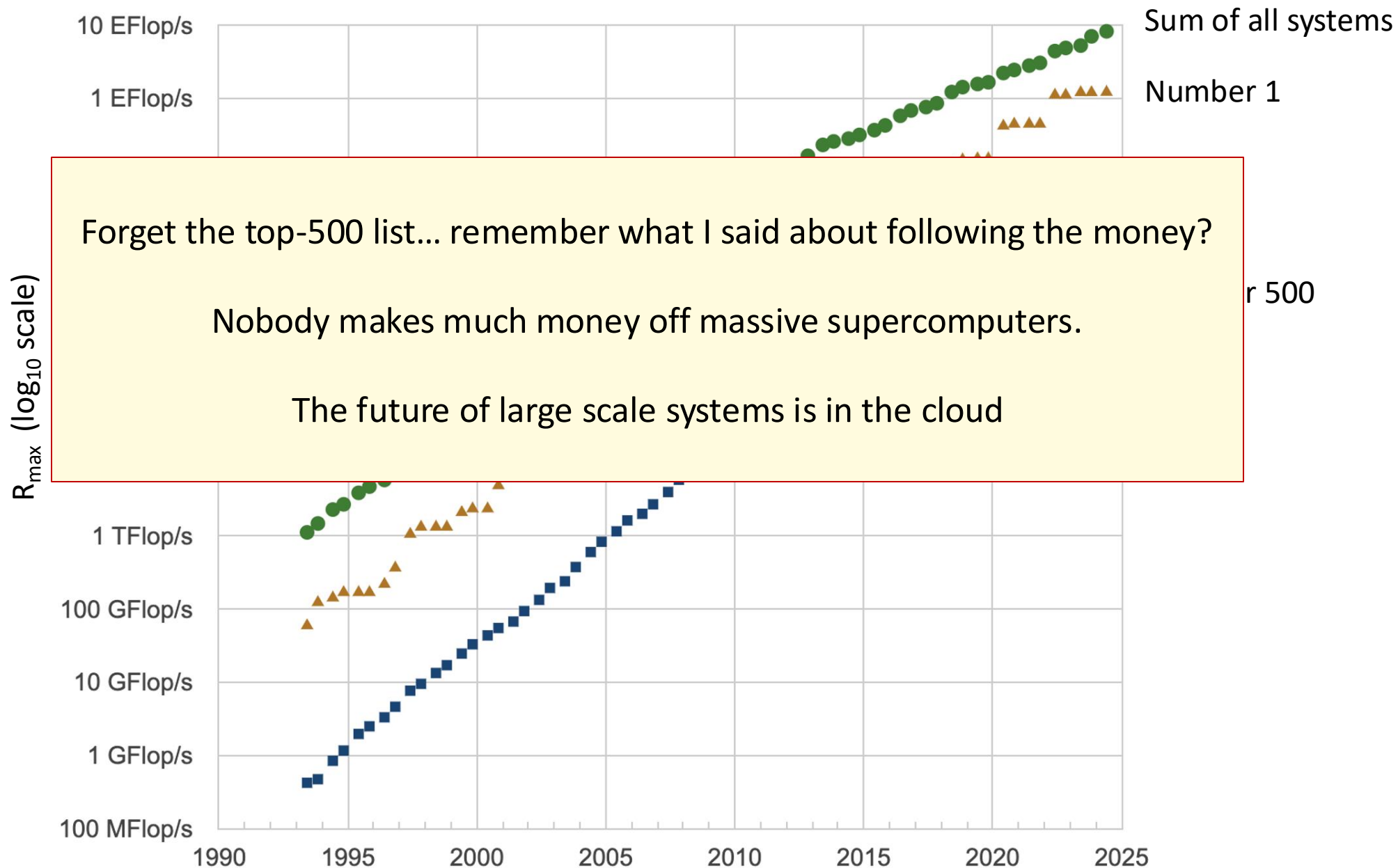
[www.top500.org](http://www.top500.org)



# We all ❤️ the top-500 List

The 500 fastest computers in the world as measured by their ability to solve a dense linear algebra problem.

[www.top500.org](http://www.top500.org)



... to really understand the cloud, and take full advantage of what it has to offer, you need to understand traditional distributed computing.

# The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 ... item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is **low and fixed**
3. Bandwidth is **high and fixed**
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is **negligible**
8. The network is homogeneous

# The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 ... item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

## Cloud

- ~~X.~~ ~~The network is reliable~~
- ~~X.~~ ~~Latency is **low and fixed**~~
- ~~X.~~ ~~Bandwidth is **high and fixed**~~
- ~~X.~~ ~~The network is secure~~
- ~~X.~~ ~~Topology doesn't change~~
- ~~X.~~ ~~There is one administrator~~
- ~~X.~~ ~~Transport cost is **negligible**~~
- ~~X.~~ ~~The network is homogeneous~~


## HPC Cluster

- ✓1. The network is reliable
- ✓2. Latency is **low and fixed**
- ✓3. Bandwidth is **high and fixed**
- ✓4. The network is secure
- ✓5. Topology doesn't change
- ✓6. There is one administrator
- ~~X.~~ ~~Transport cost is **negligible**~~
- ✓8. The network is homogeneous

HPC == tightly coupled parallel workloads



# The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster		Cloud
Execution Agent	Threads	Processes		Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes		Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD		Event driven tasks, FaaS, and Actors

Laptop/server and cluster models work well together.

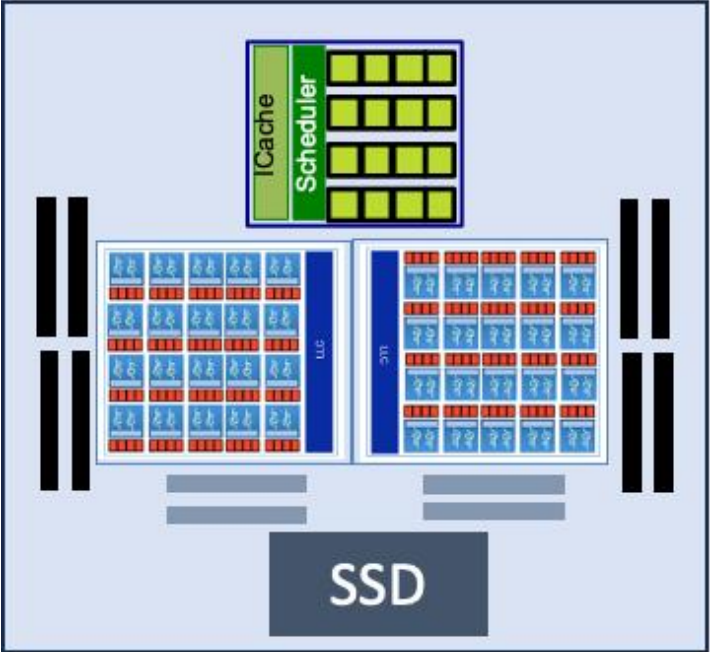
An impenetrable wall separates them from the cloud-native world

OK ... 4 domains. I need to add a column for the GPU

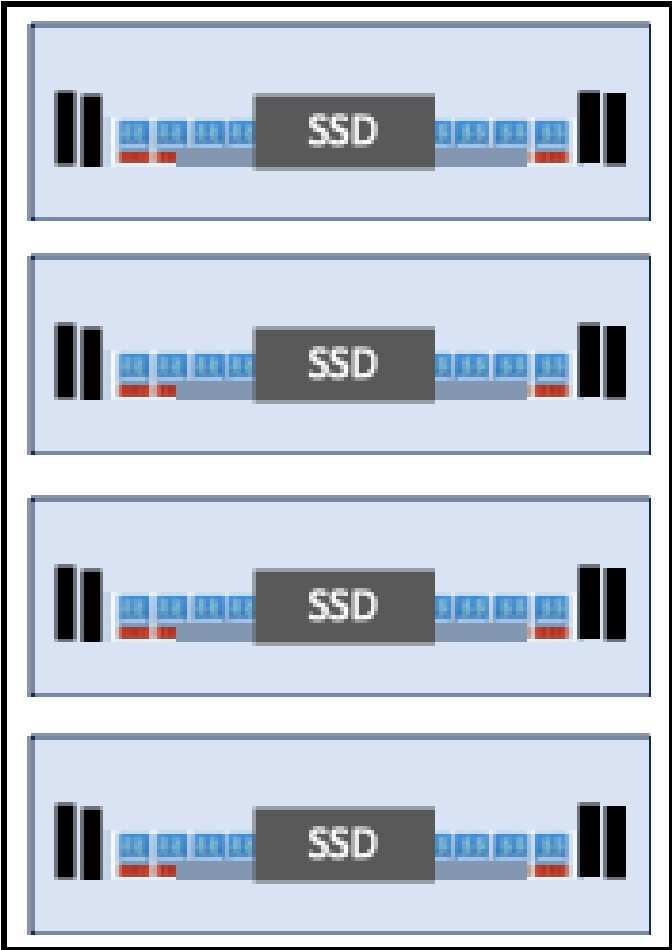
With that general background out of the way,  
consider the hardware inside a cloud data  
center.

# What do the racks in a cloud data center look like?

Take the most commonly allocated unit ... a dual processor server ... maybe with a GPU or even an FPGA

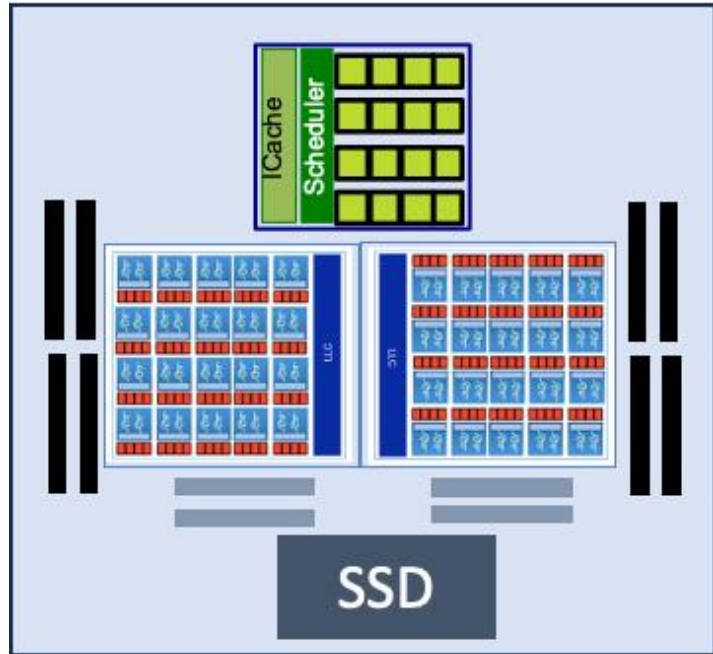


Pack them into racks and fill you data center with the racks

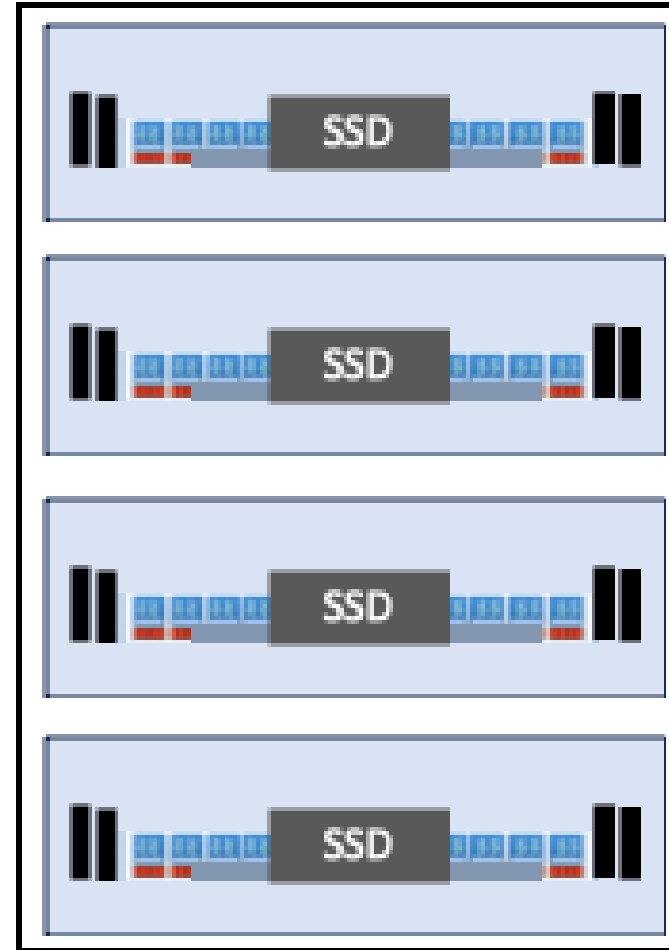


# What do the racks in a cloud data center look like?

Take the most commonly allocated unit ... a dual processor server ... maybe with a GPU or even an FPGA



Pack them into racks, add high speed networking, and fill your data center with the racks



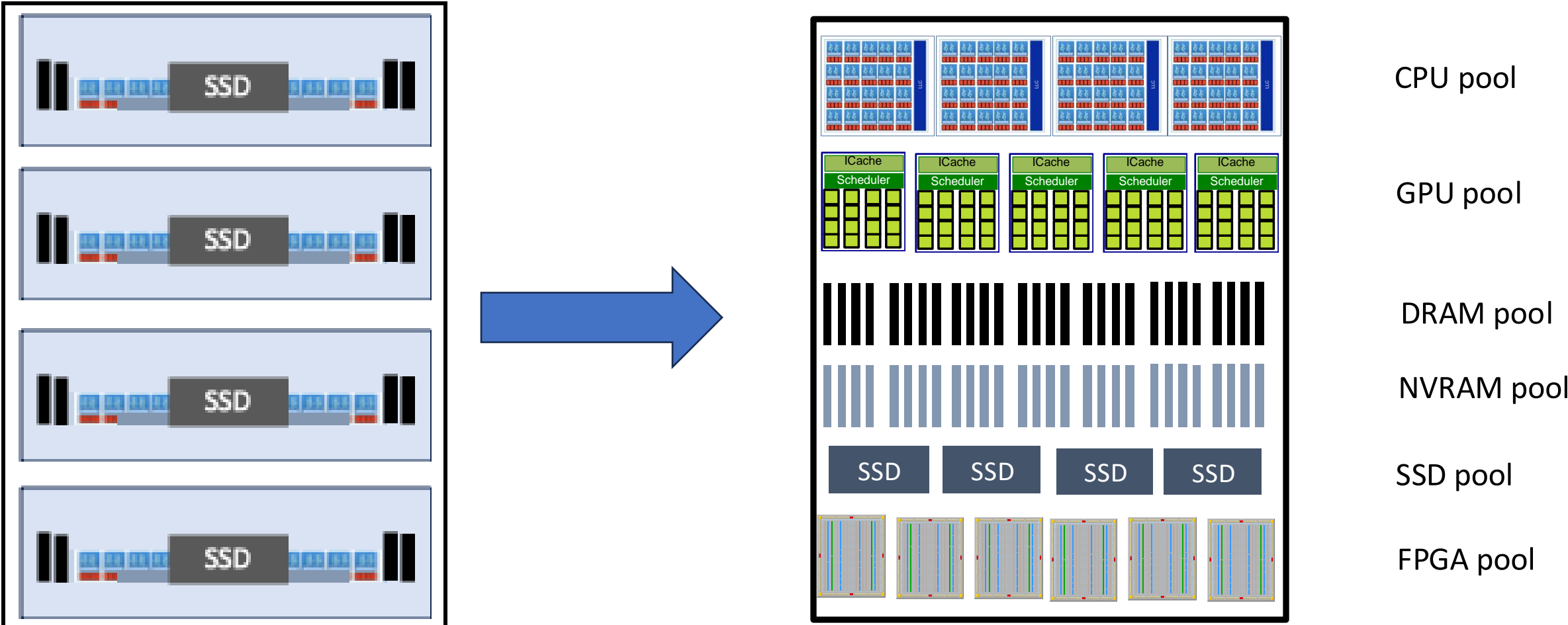
Resources are tied to the concept of node.

Memory ... for example ... is tied to CPUs so depending on CPU allocation you may have vast amounts of unused memory.

This applies to all resources, not just memory

# The birth of Disaggregated Computing

Replace a rack of nodes with pools of resources

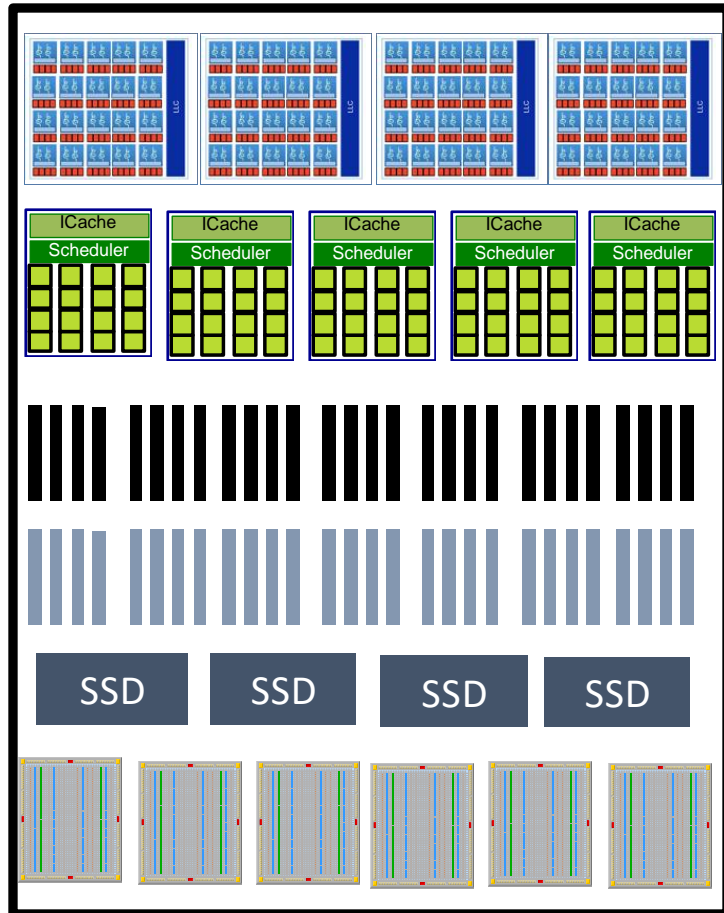


Based on "The five Epochs of distributed computing" talk by Amin Vahdat of Google:

# Disaggregated Computing for SW Defined Servers (SDS)

Consider a Rack composed of multiple pools

Dynamically compose across pools to match a software defined server to the workload



CPU pool

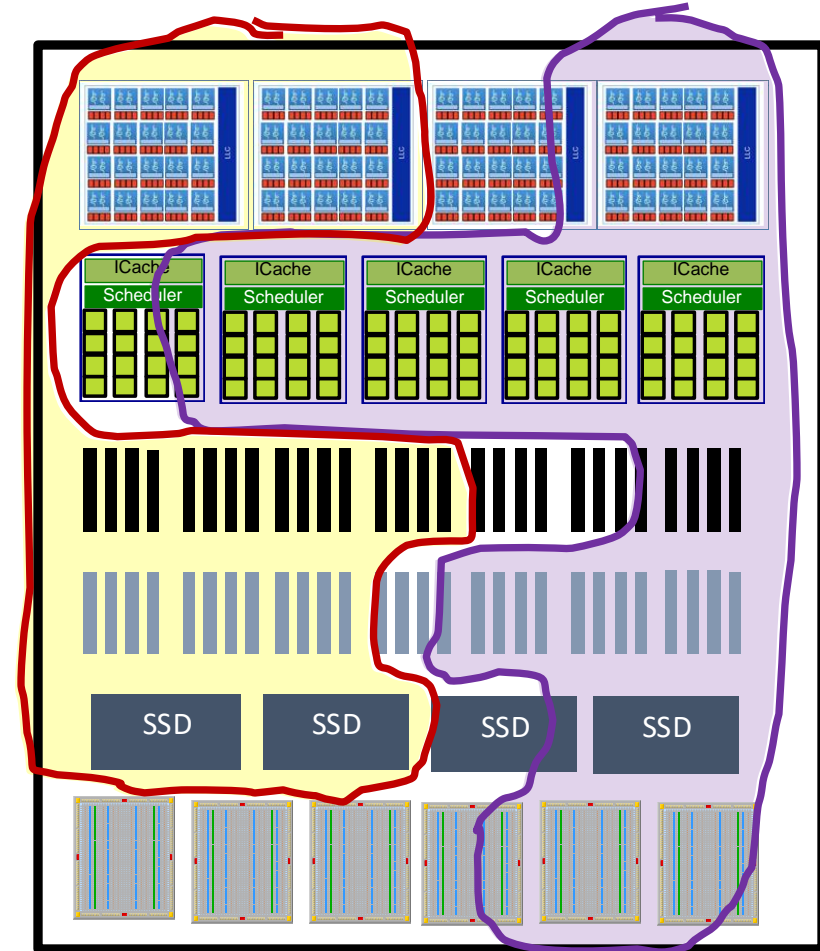
GPU pool

DRAM pool

NVRAM pool

SSD pool

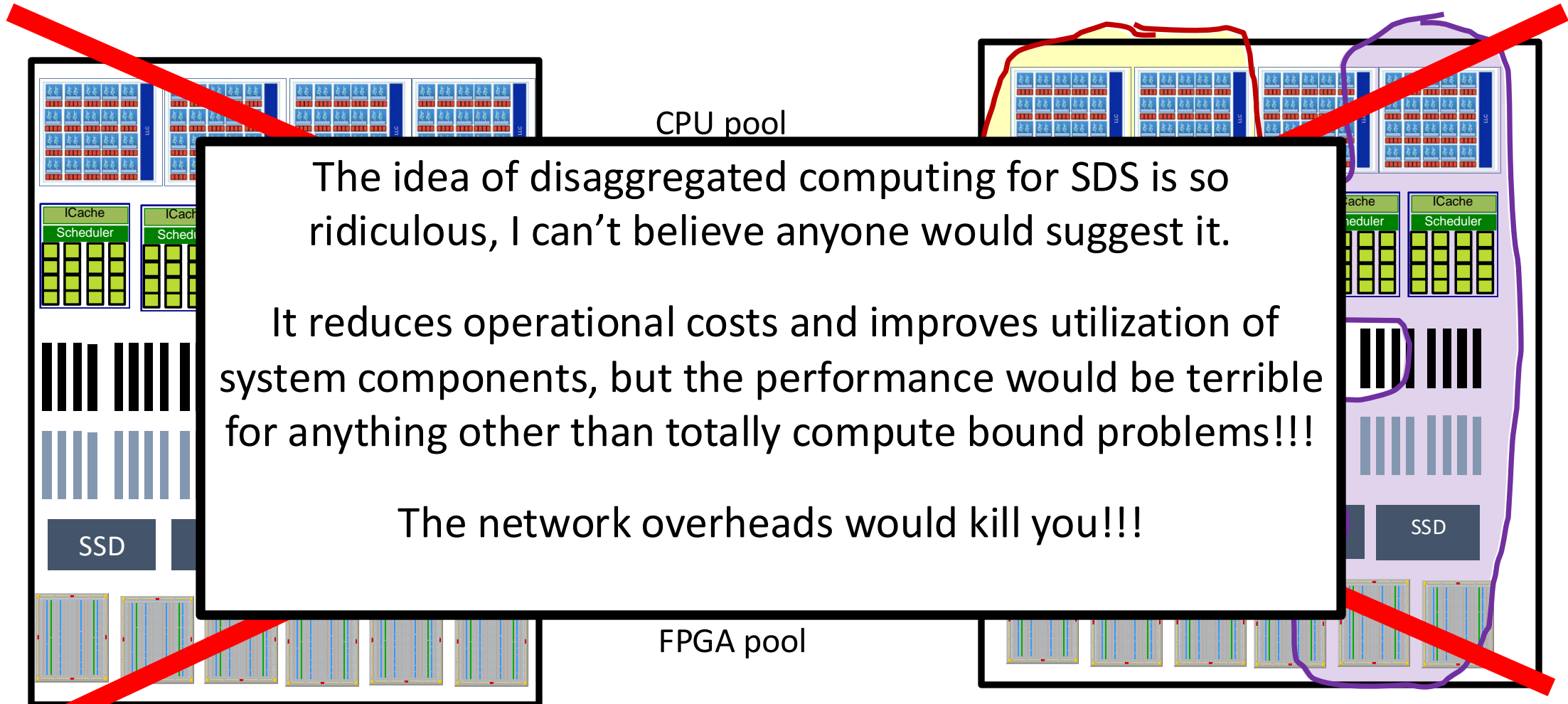
FPGA pool



# Disaggregated Computing for SW Defined Servers (SDS)

Consider a Rack composed of multiple pools

Dynamically compose across pools to match a software defined server to the workload



# TeraPHY High-Density Electronic-Photonic Chiplet

The TeraPHY™ optical I/O chiplet is a small-footprint, low-power, high-throughput alternative to copper backplane and pluggable optics communications. The TeraPHY chiplet's modular multiport design can carry eight light channels (the equivalent of an x8 PCIe Gen5 link). This industry-first optical I/O chiplet combines silicon photonics with standard CMOS manufacturing processes. It fits into existing system-in-package architectures and does not require SoC customization.

---

## Key Features

### Bandwidth

**4 Tbps**

bi-directional

### Latency

**5 ns**

per chiplet + TOF

### Power Efficiency

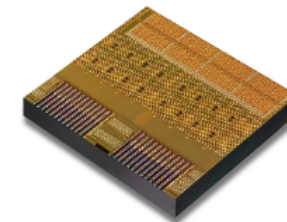
**Less than 5 pJ/b**

(10 Watts)

### Reach

**mm to km**

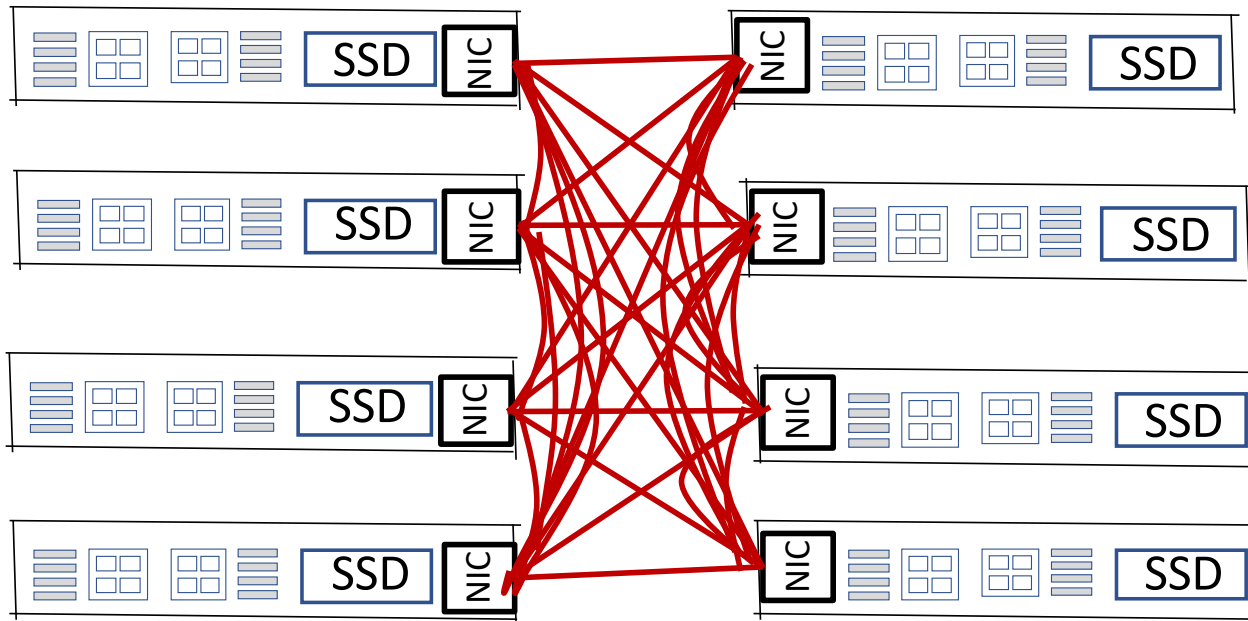
package-to-package connections



I know about this work through the Intel Group that worked on the HIVE program ... a scalable architecture optimized for workloads dominated by graph algorithms



# Networking technology is evolving quickly ... Optical networks plus new topologies



A clique: A graph where every vertex is connected to every other vertex

Optical technologies inside cabinets  
down to the chip level

New topologies ... Clique: a network  
of diameter one with  
 $O(\frac{1}{4}N^2)$  bisection bandwidth

Low latency: Five nanosecond  
latencies on each end plus time of  
flight over fiberoptic cable

# Latencies every engineer should know ...

L1 cache reference 1.5 ns

L2 cache reference 5 ns

Branch misprediction 6 ns

Uncontended mutex lock/unlock 20 ns

L3 cache reference 25 ns

Main memory reference 100 ns

“Far memory”/Fast NVM reference 1,000 ns (1 us)

Read 1 MB sequentially from memory 12,000 ns (12 us)

SSD Random Read 100,000 ns (100 us)

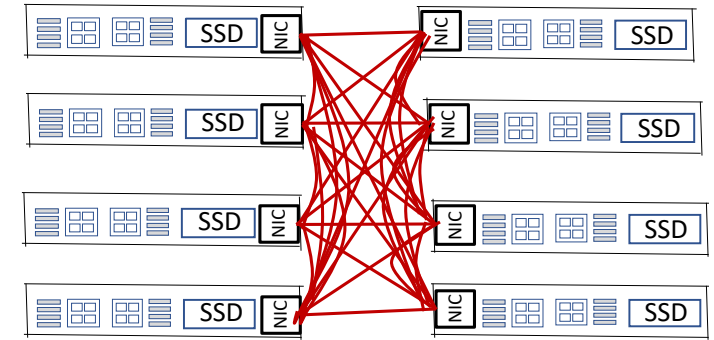
Read 1 MB bytes sequentially from SSD 500,000 ns (500 us)

Read 1 MB sequentially from 10Gbps network 1,000,000 ns (1 ms)

Read 1 MB sequentially from disk 10,000,000 ns (10 ms)

Disk seek 10,000,000 ns (10 ms)

Send packet California→Netherlands→California (150 ms)



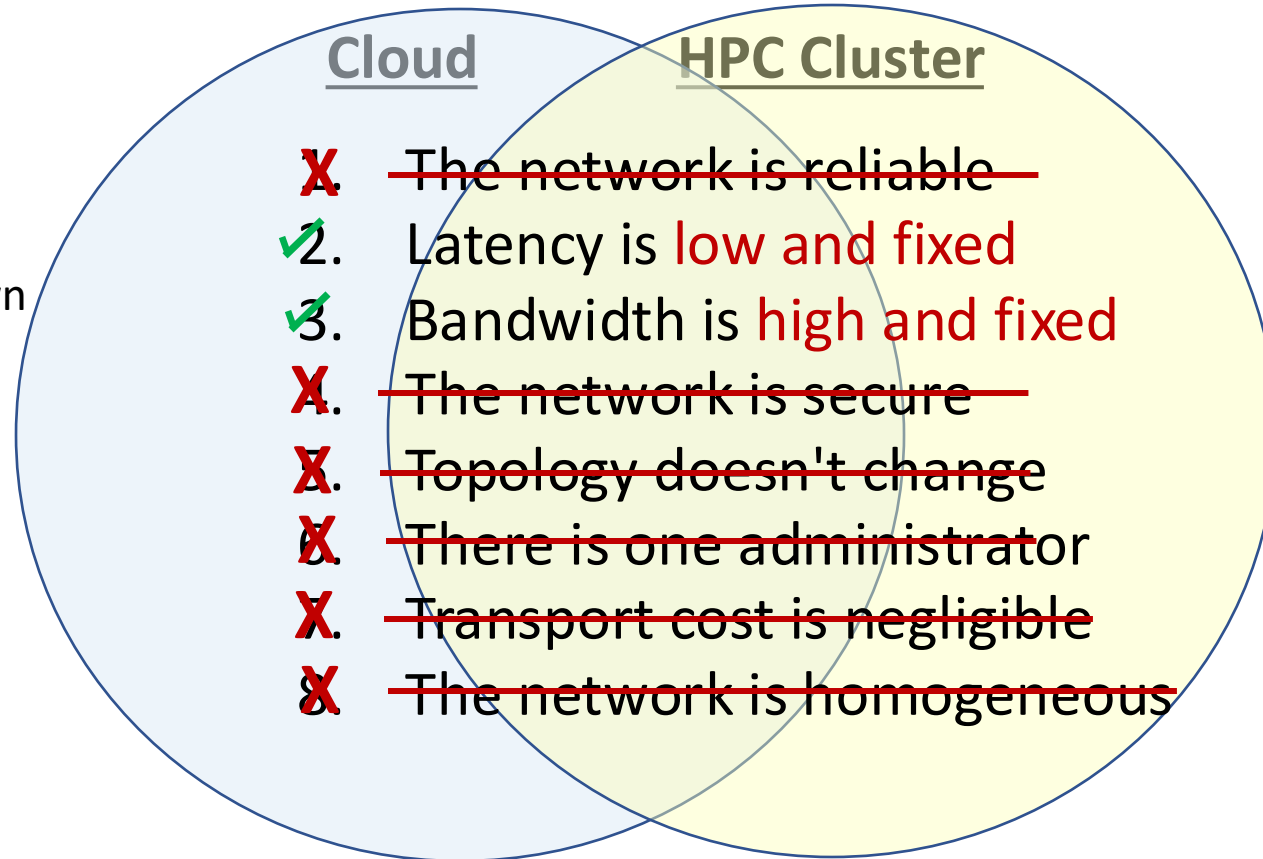
A cluster of nodes with a Clique network topology and low latency optical network...

Yields one hop network latencies on par with DRAM access latencies.

Source: **The Datacenter as a Computer: Designing Warehouse-Scale Machines**, Luiz Andre Barroso, Urs Holzle, Parthasarathy Ranganathan, 3<sup>rd</sup> edition, Morgan & Claypool, 2019.

# With next generation optical interconnects in the data center, cloud and cluster overlap

Chip-to-chip optical networks push latency down and bandwidth up



Data Streaming Accelerator reduces tail latency.

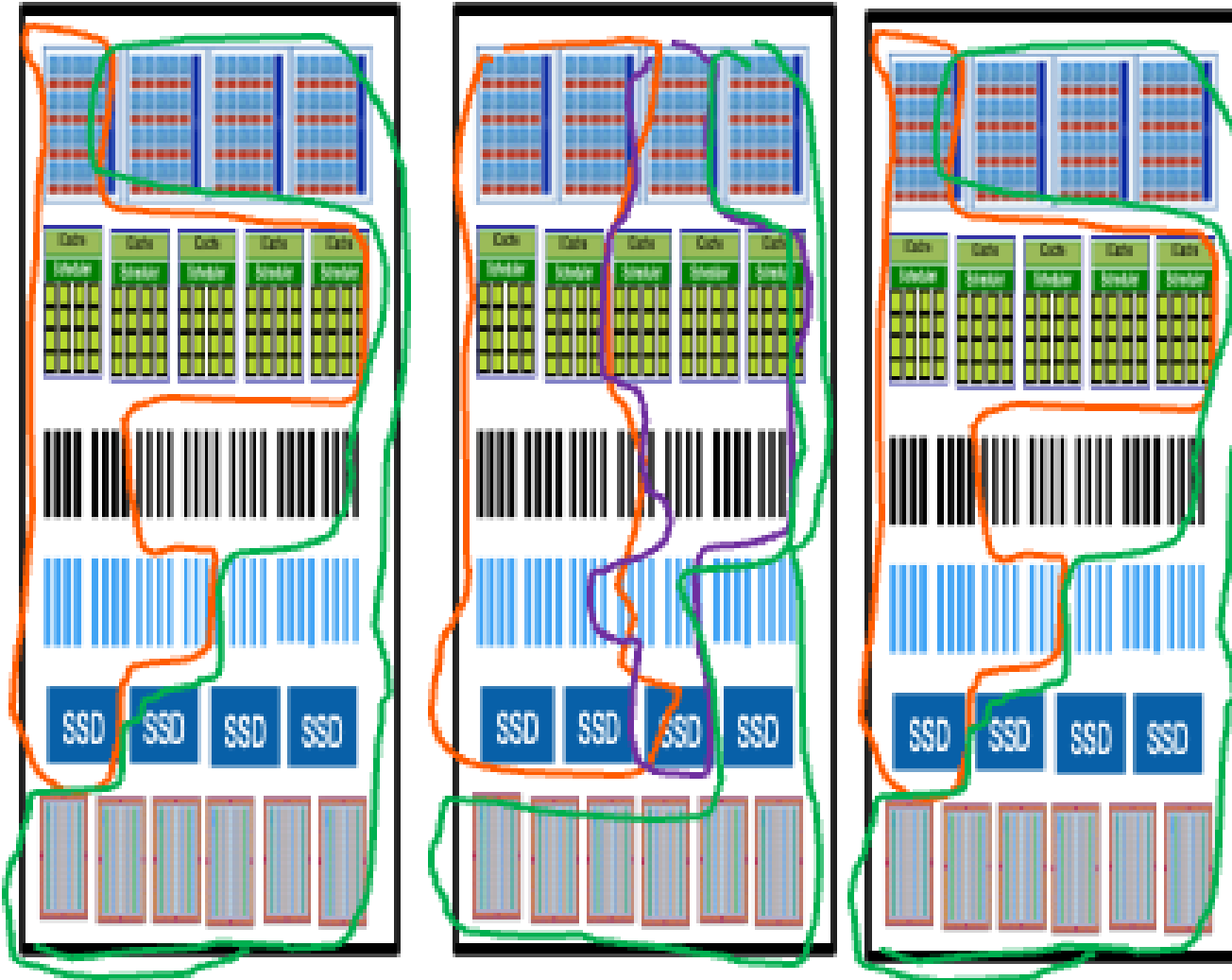
Programmable Infrastructure Processing Units drive down latency and reduces jitter

With Low Latencies, high bandwidths and stable performance, we can do loosely synchronous and synchronous applications in the cloud. The economics of the cloud vs dedicated HPC clusters means the cloud will dominate HPC

HPC applications will need to change to deal with reliability and network inhomogeneities.

# SW Defined clusters of SW defined Servers

Low latency, high bandwidth network between cliques



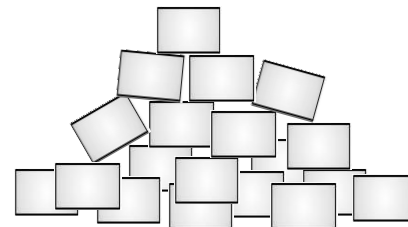
• • •

- Dynamic ... changing from one job to the next.
- SW defined servers composed of heterogeneous components
- Dynamically composed into a cluster
- Integrated over a 5G network to devices (and people) at the edge

# The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors

Advances in networking technology plus low-overhead software stacks optimized to reduce tail-latency will shatter this wall



This is a compelling future.

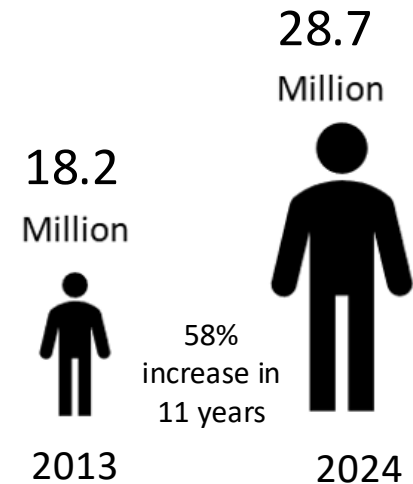
Who is going to write all the software for these systems?

# Evolution of software developers from 2013 to 2024

The number of Software developers worldwide is growing rapidly ...

<https://www.computersciencezone.org/developers>

<https://www.statista.com/statistics/627312/worldwide-developer-population/>



But look what the U.S. Bureau of Labor Statistics says ...

How can both of these trends be correct?

Developers increasingly come from application domains, not computer science!

## Quick Facts: Computer Programmers

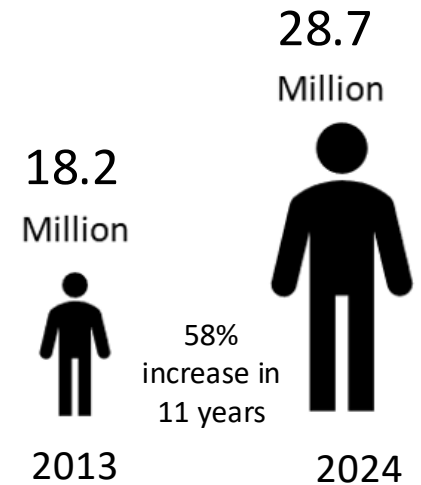
<a href="#">2019 Median Pay</a>	\$86,550 per year
<a href="#">2021 Median Pay</a>	\$99,700 per year
<a href="#">Typical Entry-Level Education</a>	Bachelor's degree
<a href="#">Number of Jobs, 2019</a>	213,900
<a href="#">Number of Jobs, 2021</a>	139,400
<a href="#">Job Outlook, 2021-31</a>	-10% (Decline)
<a href="#">Employment Change, 2023-33</a>	-13,400

# Evolution of software developers from 2013 to 2024

The number of Software developers worldwide is growing rapidly ...

<https://www.computersciencezone.org/developers>

<https://www.statista.com/statistics/627312/worldwide-developer-population/>



## Quick Facts: Computer Programmers

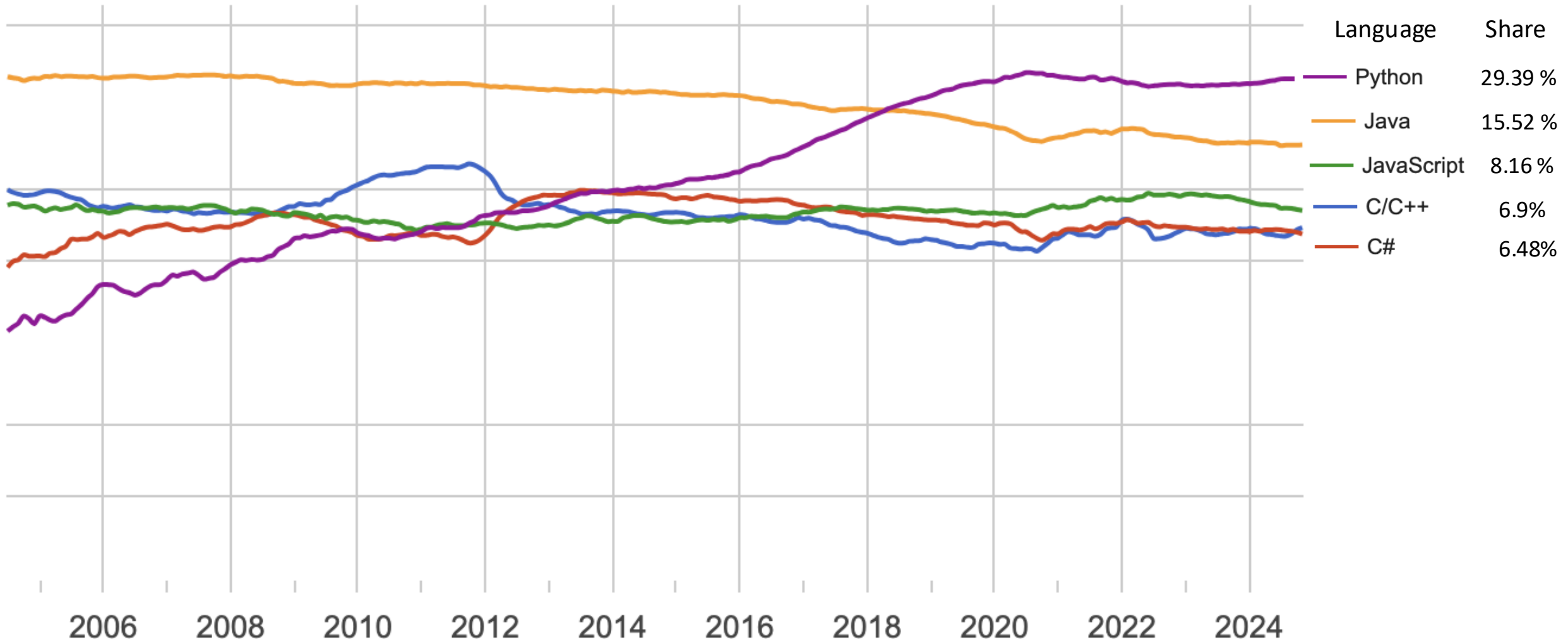
<a href="#">2019 Median Pay</a>	\$86,550 per year
<a href="#">2021 Median Pay</a>	\$99,700 per year
<a href="#">Typical Entry-Level Education</a>	Bachelor's degree
<a href="#">Number of Jobs, 2019</a>	213,900
<a href="#">Number of Jobs, 2021</a>	139,400
<a href="#">Job Outlook, 2021-31</a>	-10% (Decline)
<a href="#">Employment Change, 2023-33</a>	-13,400

It used to be that professional programmers wrote the applications needed by different communities.

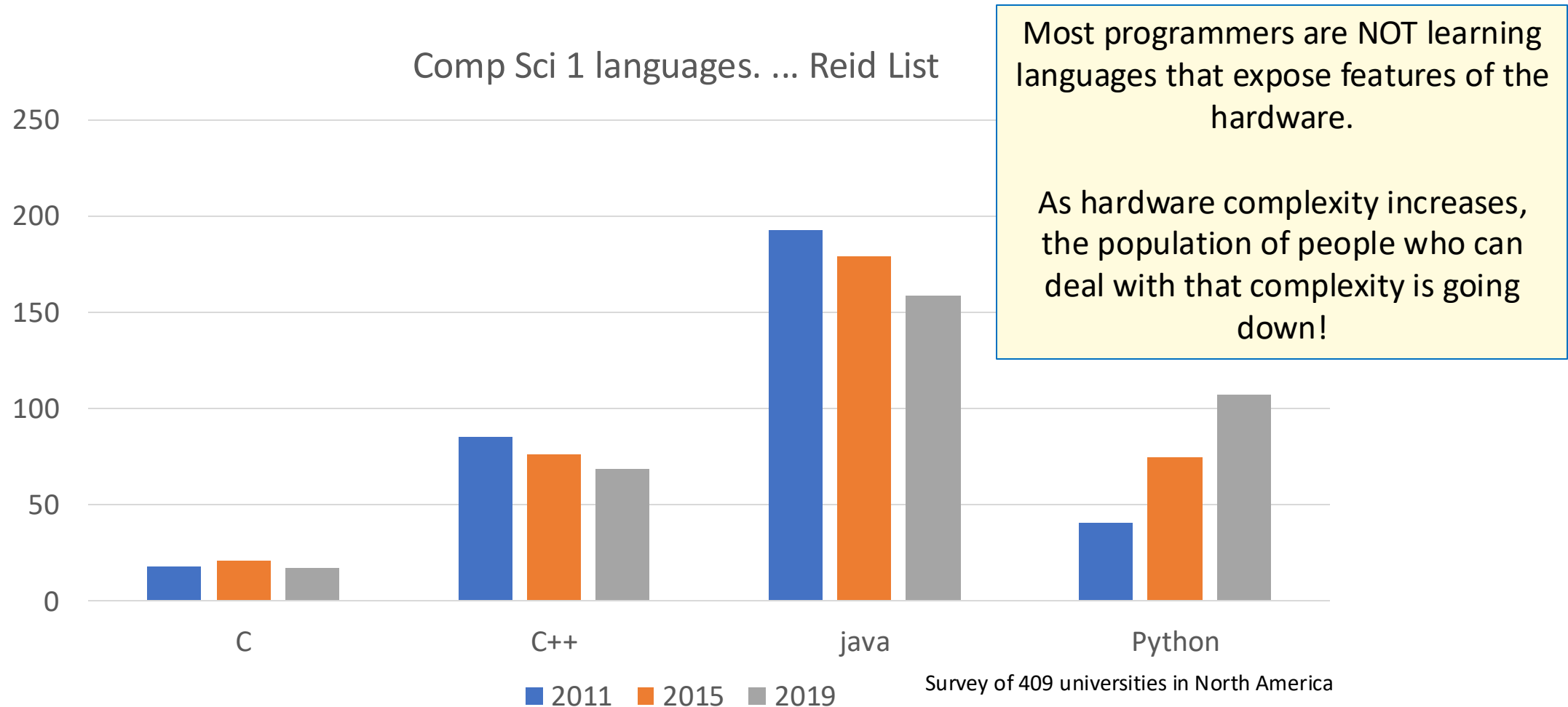
Now those *different communities* write their own applications.



# Popularity of Programming Languages (PyPI)



# Primary Language used in first year, Computer Science Courses



The Reid List tracks a large sample of North American Universities and the languages they use in teaching.

The Reid List was started by Richard Reid in the 1990s. He has retired but others are carrying on the tradition. The above data comes from Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning, Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, Jason P. Siegfried, The 16th International Conference on Computer Science & Education (ICCSE 2021) August 18-20, 2021.

# Solution: Bring HPC programming to Python

## PyOMP: mapping OpenMP into Python

```
from numba import njit
```

```
from numba.openmp import openmp_context as openmp
```

```
@njit
```

```
def piFunc(NumSteps):
```

```
    step = 1.0/NumSteps
```

```
    pisum = 0.0
```

```
    with openmp ("parallel for private(x) reduction(+:pisum)"):
```

```
        for i in range(NumSteps):
```

```
            x = (i+0.5)*step
```

```
            pisum += 4.0/(1.0 + x*x)
```

```
    pi = step*pisum
```

```
    return pi
```

```
pi = piFunc(100000000)
```

OpenMP managed through the *with* context manager.

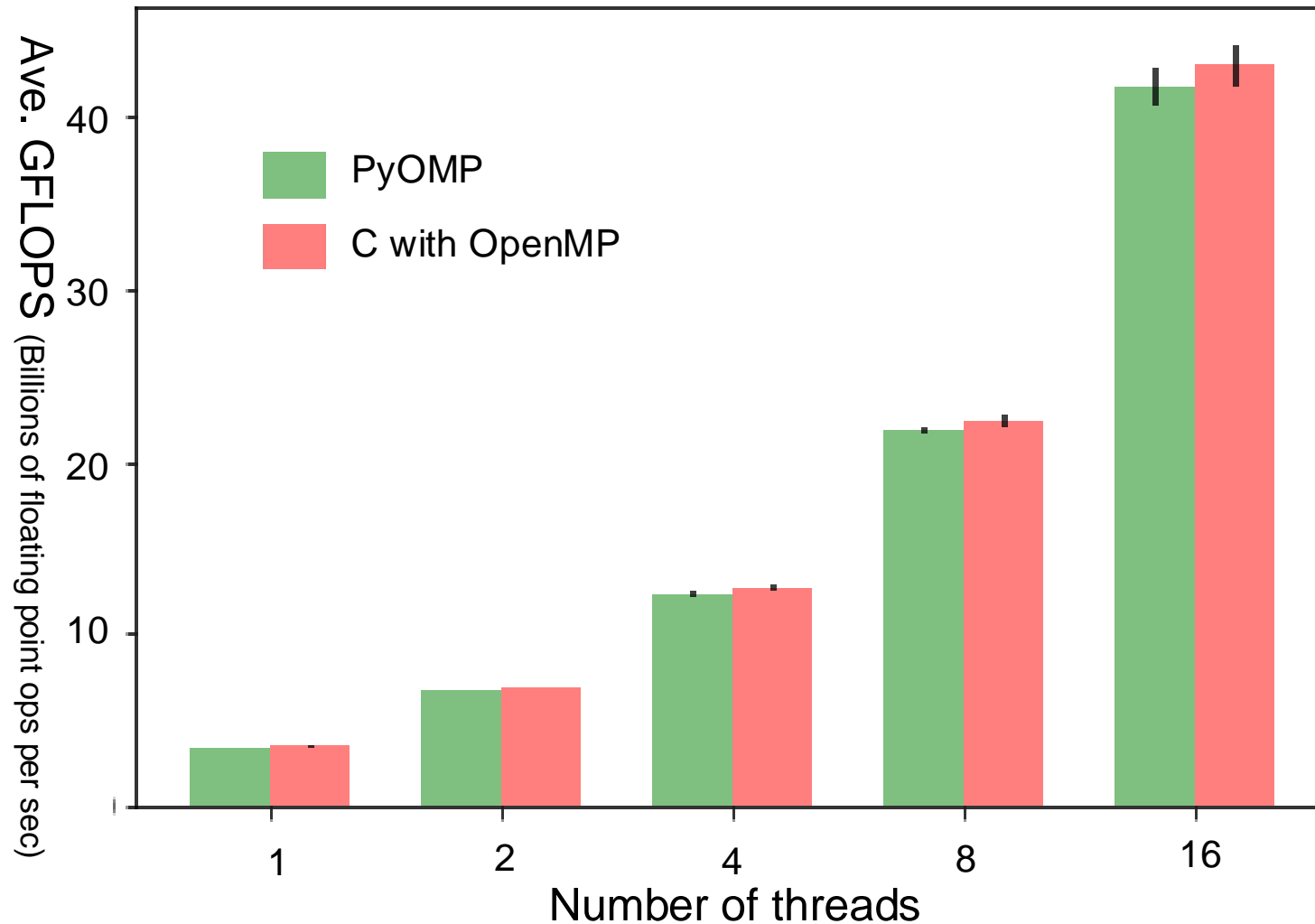
Numba Just In Time (JIT) compiler compiles the Python code into LLVM thereby bypassing the GIL. Compiled code cached for later use.

Pass the OpenMP directive into the OpenMP context manager as a string

- **parallel**: creates a team of threads
- **for**: maps loop iterations onto threads.
- **private(x)**: each threads gets its own x
- Loop control index of a parallel for (**i**) is private to each thread.
- **reduction(+:sum)**: combine sum from each thread using +

# DGEMM PyOMP vs C-OpenMP

Matrix Multiplication, double precision, order = 1000, with error bars (std dev)



250 runs for order 1000 matrices

PyOMP times **DO NOT** include the one-time JIT cost of ~2 seconds.

... but remember, the JIT'ed code can be cached for future use. It's straightforward to hide the JIT cost.

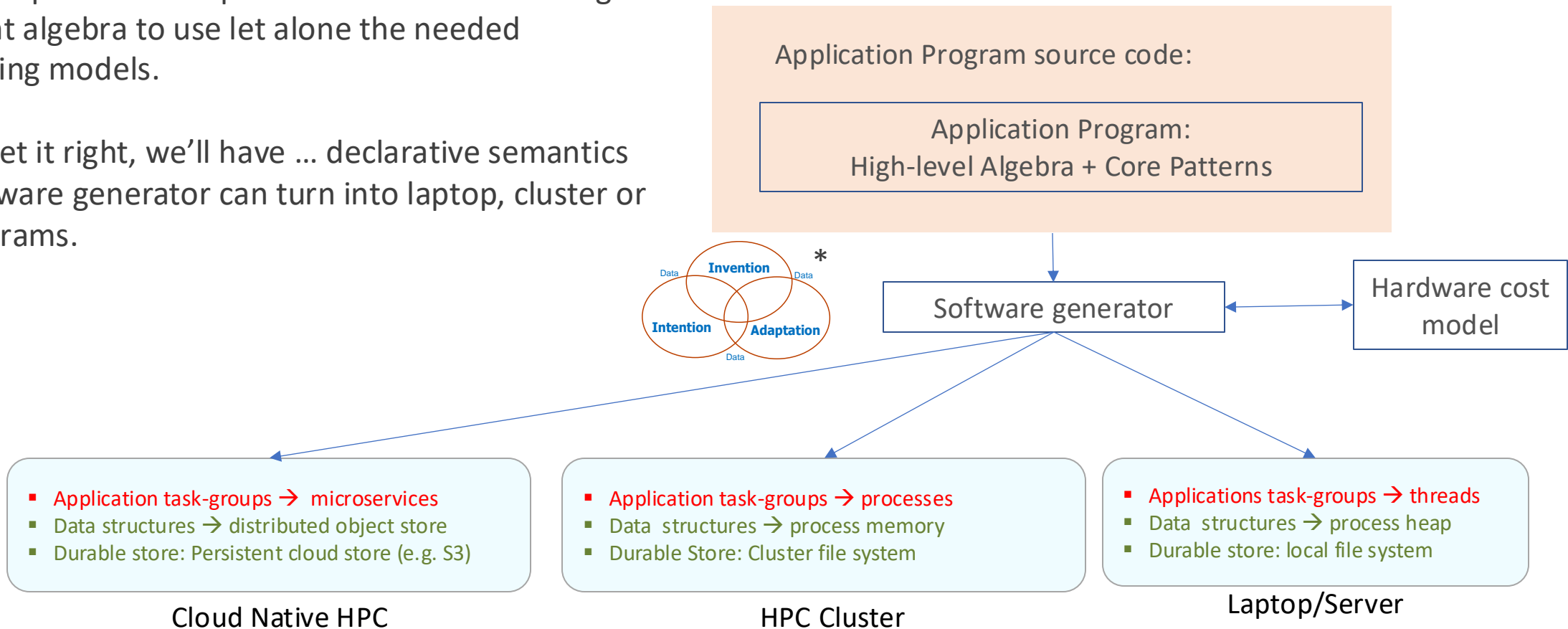
Intel® Xeon® E5-2699 v3 CPU, 18 cores, 2.30 GHz, threads mapped to a single CPU, one thread/per core, first 16 physical cores.  
Intel® icc compiler ver 19.1.3.304 (icc -std=c11 -pthread -O3 xHOST -qopenmp)

**Traditional parallel programming models mapped into Python is valuable, however, it doesn't solve the problem of how we map code onto diverse hardware.**

**Our only hope is to rethink how we engineer software**

# One codebase → many systems

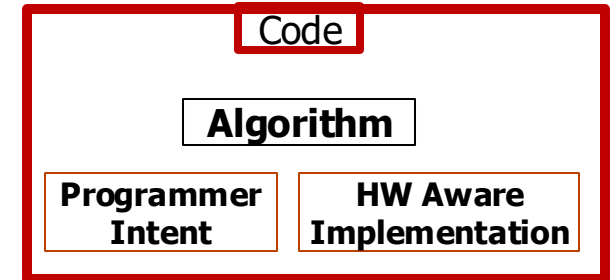
- Performance, Productivity AND Portability ... the database people “did it” with relational algebras and SQL.
- Can HPC people solve this problem? We can't even agree on the right algebra to use let alone the needed programming models.
- But if we get it right, we'll have ... declarative semantics that a software generator can turn into laptop, cluster or cloud programs.



\*This is the logo of the machine programming research program I helped lead inside Intel Labs

# Traditional programming

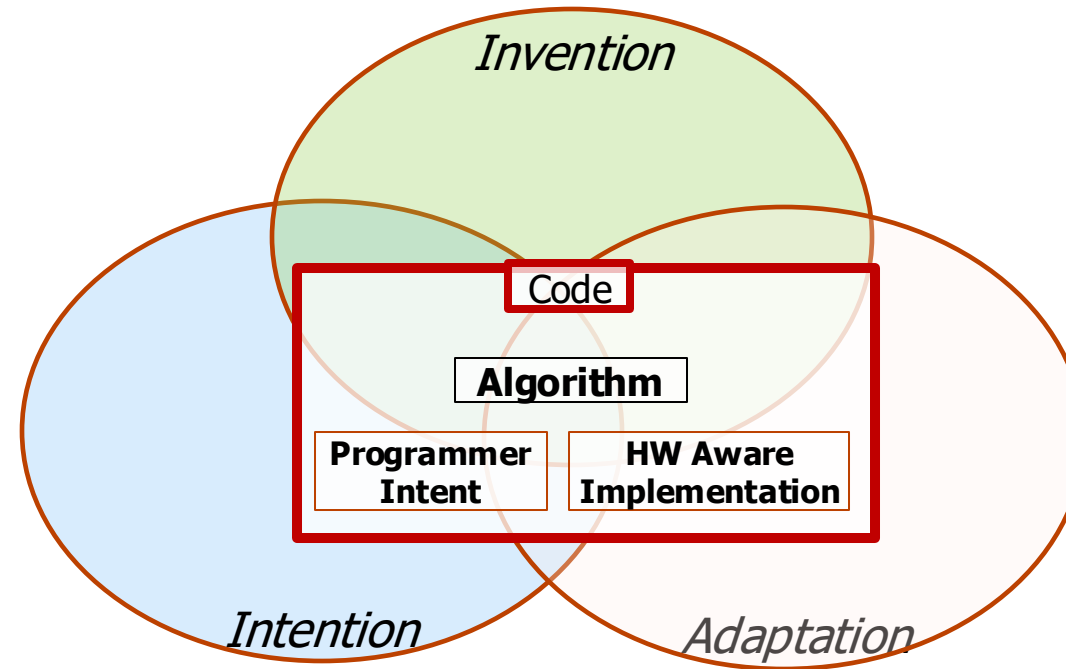
- Three fundamental aspects of software development:
  - Express the intent of the program
  - Invent algorithms/data-structures
  - Adapt the software to the details of the hardware for high performance
- Programmers do all this together when they write code.



Past attempts to automatically generate code have failed since they tried to "do it all" together (just as a human would).

# Separation of concerns

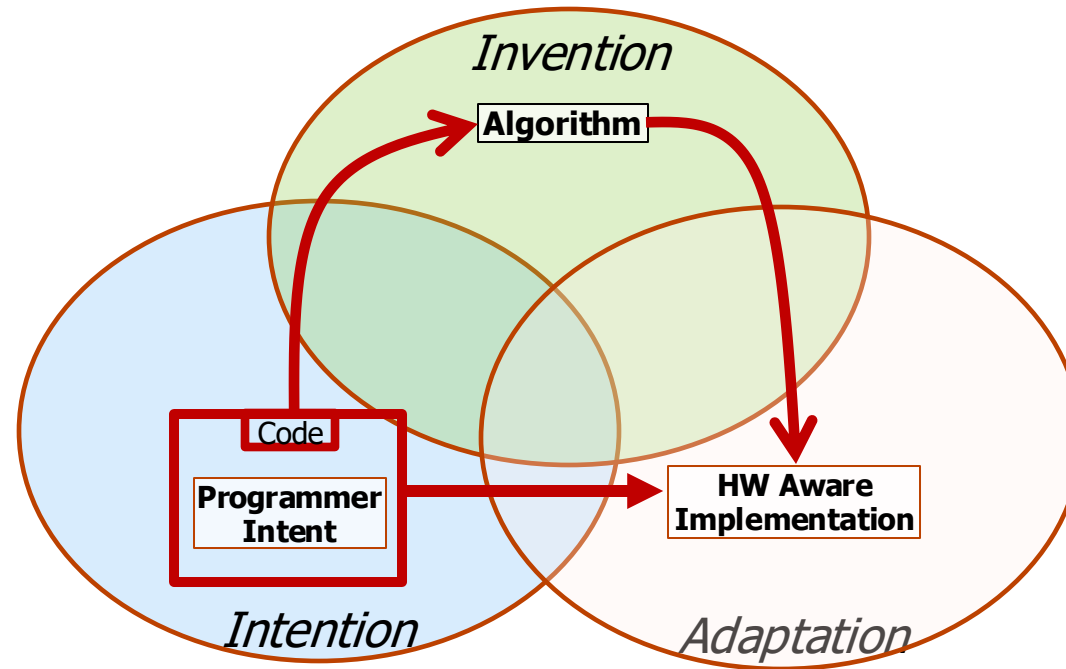
- Let's break up the software development process and consider each aspect Separately





# Separation of concerns

- Let's break up the software development process and consider each aspect Separately



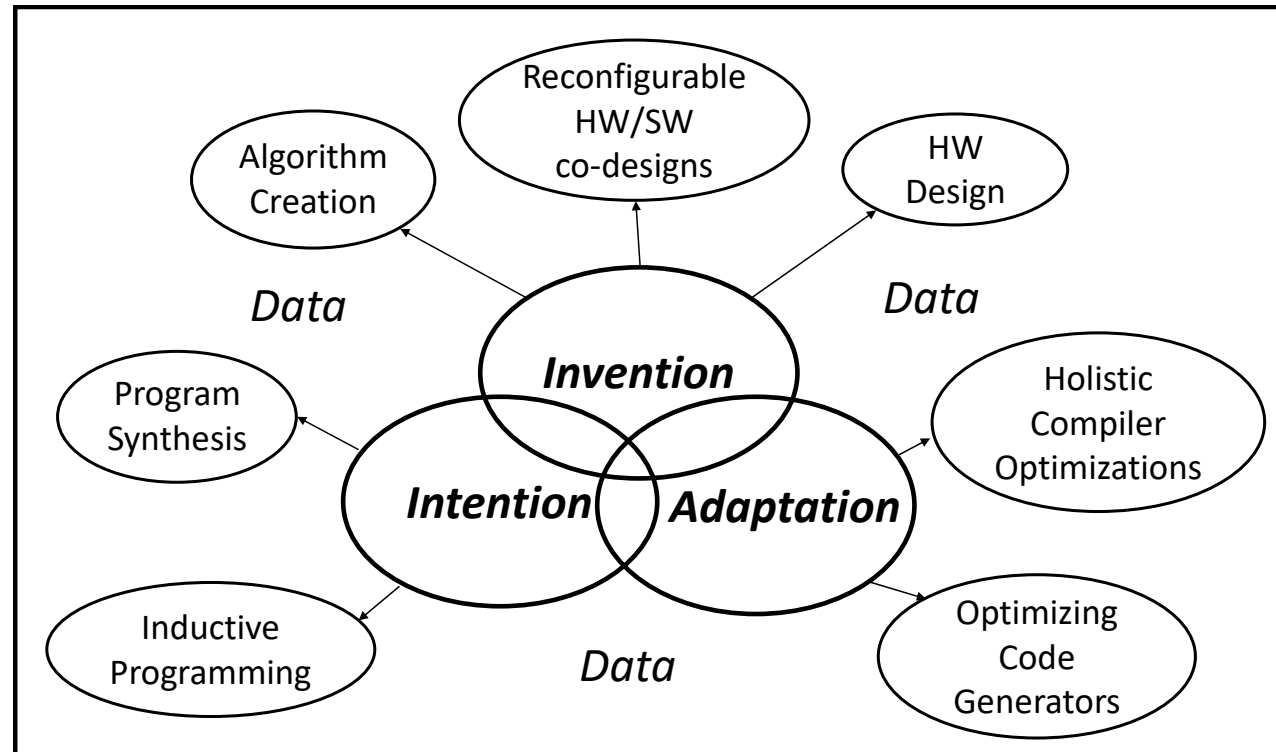
Programmers should just worry about expressing their intent. We will automate the Invention and Adaptation work

# The Three Pillars of Machine Programming

## MAPL/PLDI'18



Justin Gottschlich, ~~Intel~~  
Armando Solar-Lezama, MIT  
Nesime Tatbul, Intel  
Michael Carbin, MIT  
Martin, Rinard, MIT  
Regina Barzilay, MIT  
Saman Amarasinghe, MIT  
Joshua B Tenebaum, MIT  
Tim Mattson, ~~Intel~~  
University of Bristol

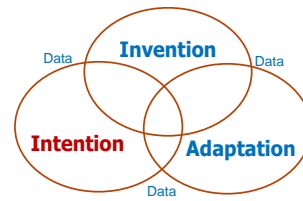


A position paper laying out our vision for how to solve the machine programming problem. The three Pillars:

- **Intention:** Discover the intent of a programmer
- **Invention:** Create new algorithms and data structures
- **Adaption:** Evolve in a changing hardware/software world

# Halide: Focusing on programmer intent

A domain specific language for image processing



Halide  
separates the

## Algorithm

from the

## Schedule

```
Func blur_3x3(Func input) {  
  Func blur_x, blur_y;  
  Var x, y, xi, yi;
```

```
// The algorithm - no storage or order
```

```
blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;  
blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
```

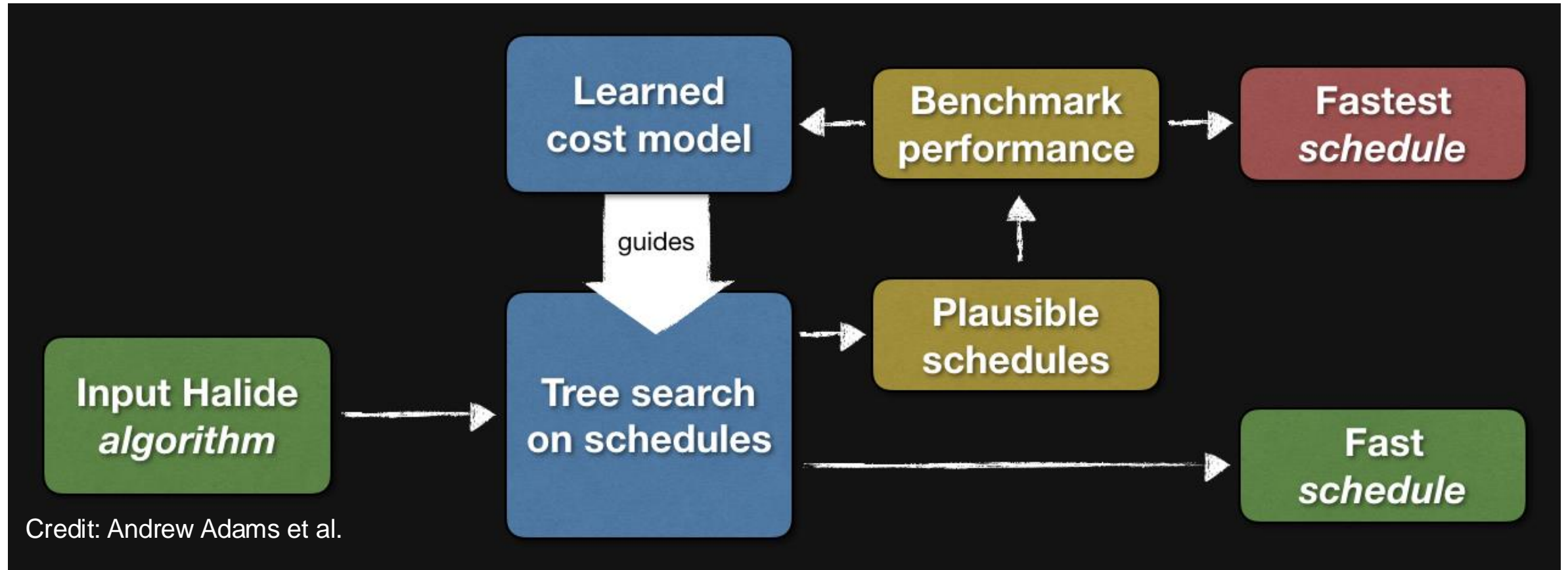
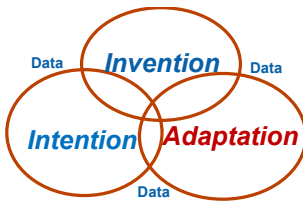
```
// The schedule - defines order, locality; implies storage
```

```
blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);  
blur_x.compute_at(blur_y, x).vectorize(x, 8);
```

```
  return blur_y;  
}
```

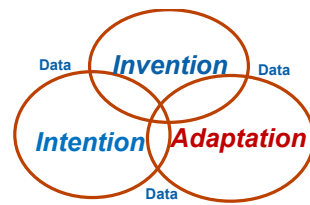
- Algorithm:
  - What the program does,
  - Written by a domain specialist
- Schedule:
  - How the program runs
  - Written by SW/HW expert

# Halide Learned Schedules

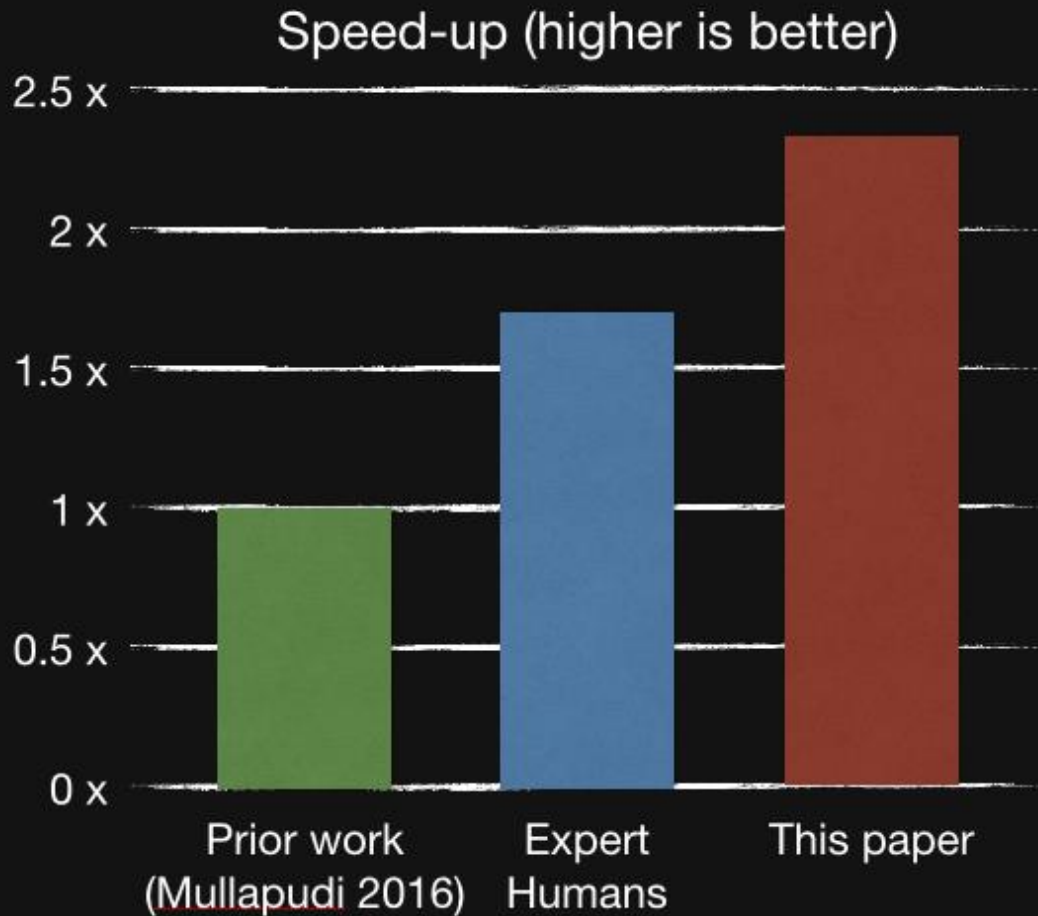


Credit: Andrew Adams et al.

# Superhuman Performance



## A new automatic scheduling algorithm for Halide



### Larger search space

- includes more Halide scheduling features
- extensible

### Hybrid cost model

- Mix of machine learning and hand-designed terms
- Can model complex architectures

But now days ... all people seem to care about  
is large language models (LLM) that generate  
code

# LLMs don't do well with parallel code

## Example

If it is possible, use OpenMP to accelerate the following code:

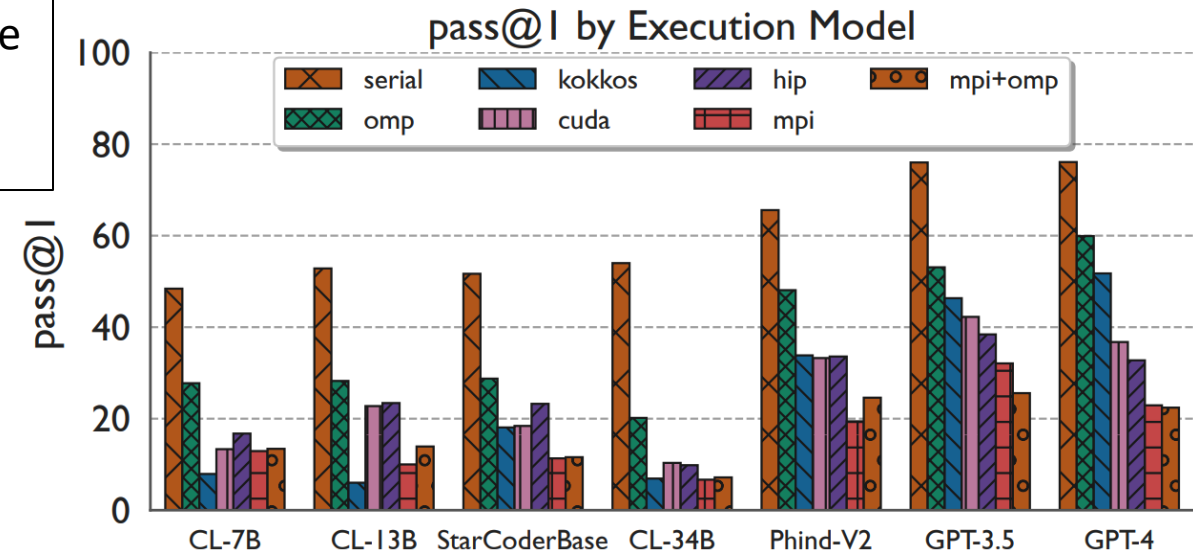
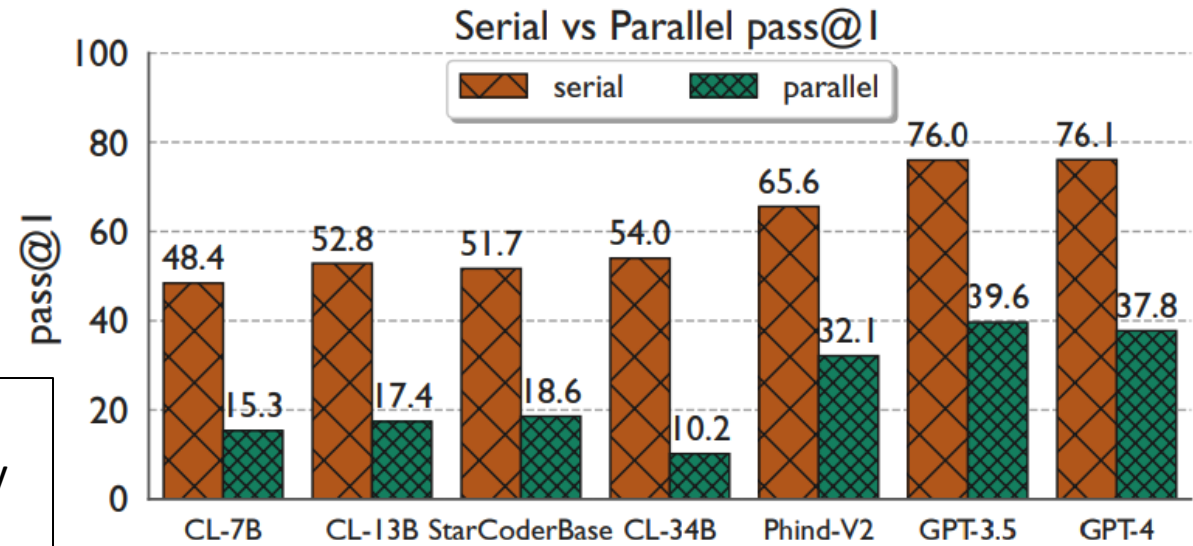
```
int main()
{
    int N = 10000;
    double *arr = (double*)malloc(N * sizeof(double));

    for(int i = 0; i < N-1; i++)
        arr[i] = arr[i] + arr[i+1]
}
```



```
// Parallel region
#pragma omp parallel for
for(int i = 0; i < N-1; i++) {
    arr[i] = arr[i] + arr[i+1];
}
```

pass@1  
probability  
correct  
parallel code  
on first  
attempt



Source: Can Large Language Models Write Parallel Code? (2024), <https://arxiv.org/abs/2401.12554>

# Can we do better with LLMs specialized to HPC?

- Kadosh, Tal, Niranjana Hasabnis, Vy A. Vo, Nadav Schneider, Neva Krien, Mihai Capota,, Abdul Wasay , Guy Tamir, Ted Willke, Nesreen Ahmed, Yuval Pinter, Timothy Mattson, and Gal Oren . "**MonoCoder: Domain-Specific Code Language Model for HPC Codes and Tasks.**" *arXiv preprint <https://arxiv.org/html/2312.13322v2>*, (2024). **HPEC'24**
- Kadosh, Tal, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. "**Quantifying OpenMP: Statistical Insights into Usage and Adoption [HPCorpus].**" *arXiv preprint arXiv:2308.08002* (2023). **HPEC'23**
- Schneider, Nadav, Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. "**MPI-rical: Data-Driven MPI Distributed Parallelism Assistance with Transformers.**" *arXiv preprint arXiv:2305.09438* (2023). **AI4DEV @ SC'23**
- Kadosh, Tal, Nadav Schneider, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. "**Advising OpenMP Parallelization via a Graph-Based Approach with Transformers.**" *arXiv preprint arXiv:2305.11999* (2023). **IWOMP'23**
- Harel, Re'em, Yuval Pinter, and Gal Oren. "**Learning to parallelize in a shared-memory environment with transformers.**" *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023. **PPoPP'23**
- Re'em Harel, Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren "**PragFormer: Data-driven Parallel Source Code Classification with Transformers**", 29 August 2023, *preprint* (V1) Research Square [<https://doi.org/10.21203/rs.3.rs-3254961/v1>]

This collaborative work is led by **Gal Oren** (Technion) and **Yuval Pinter** (Ben Gurion Univ) and their students:

- Tal Kadosh
- Nadav Schneider
- Neva Krien
- Re'em Harel



# You need data: HPCorpus



- We searched C, C++ and Fortran codes in GitHub from 2012-mid 2023 for parallel code

	Repos	Size(GB)	Files (#)	Functions (#)
C	144,522	46.23	4,552,736	87,817,591
C++	150,481	26.16	4,735,196	68,233,984
Fortran	3,683	0.68	138,552	359,272

All data and associated scripts are available at:  
<https://github.com/Scientific-Computing-Lab-NRCN/HPCorpus>

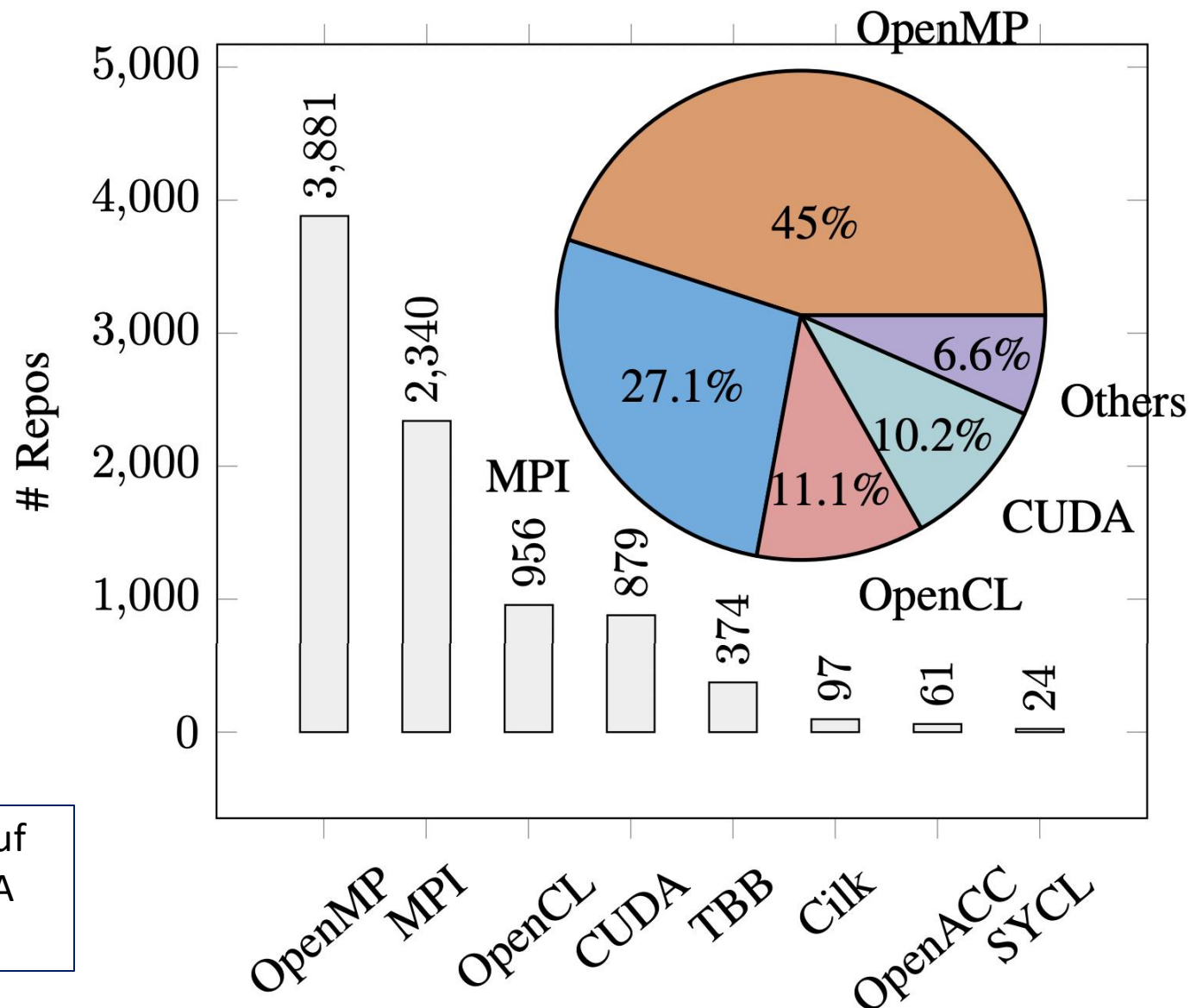
```
1 WITH selected_repos as (  
2   SELECT f.id, f.repo_name as repo_name, f.ref as  
3     ↪ ref, f.path as path  
4 FROM `bigquery-public-data.github_repos.files` as  
5   ↪ f  
6 JOIN `bigquery-public-data.github_repos.licenses`  
7   ↪ as l on l.repo_name = f.repo_name  
8 )  
9 deduped_files as (  
10  SELECT f.id, MIN(f.repo_name) as repo_name,  
11    ↪ MIN(f.ref) as ref, MIN(f.path) as path  
12 FROM selected_repos as f  
13 GROUP BY f.id  
14 )  
15 SELECT  
16   f.repo_name, f.ref, f.path, c.copies, c.content,  
17 FROM deduped_files as f  
18 JOIN `bigquery-public-data.github_repos.contents`  
19   ↪ as c on f.id = c.id  
20 WHERE  
21   NOT c.binary  
22   AND (f.path like '%.c' OR f.path like '%.cpp' OR  
23     ↪ f.path like '%.f' OR f.path like '%.f90' OR  
24     ↪ f.path like '%.f95')
```

# Programming models usage 2013 to 2023

OpenMP is clearly the most popular parallel programming model in use today.

This surprised us ... we thought MPI would be the most popular model.

However, while MPI dominates at the major supercomputing research center, GitHub includes laptop, single server, and a much wider range of applications.



Note: since we did not collect files with .cu or .cuf suffices, we unintentionally undercounted CUDA usage in HPCorpus.

# You need to tokenize your data

- LLMs work on sequences of tokens from a pre-set vocabulary.
- GPT and related technologies use a standard tokenizer derived from research on human language.
- Can we do better with a tokenizer specialized to HPC languages: C, C++ and Fortran?

# Current tokenizer technology applied to HPCorpus-Fortran

	GPT2 (BPE)	NLTK
Code sample	Tokenized code	AST (xSBT)
<pre>function calculate_pi (max, seed) result(pi) implicit none integer, intent(in) :: max, seed real(8) :: pi real(8) :: area, x, y integer :: i external :: drand48 integer :: pi_count  pi_count = 0 call seed48(seed)  do i = 1, max x = drand48() * 2 - 1 y = drand48() * 2 - 1 ...</pre>	<pre>['function', 'calculate', '_', 'pi', '(', 'max', ',', 'seed', ')', 'result', '(', 'pi', ')', 'impl', 'icit', 'none', 'integer', ',', 'intent', '(', 'in', ')', '::', 'max', ',', 'seed', 'real', '(', '8', ')', '::', 'pi', 'real', '(', '8', ')', '::', 'area', ',', 'x', ',', 'y', 'integer', '::', 'i', 'external', '::', 'dr', 'and', '48', 'integer', '::', 'pi', '_', 'count', 'pi', '_', 'count', '=', '0', 'call', 'seed', '48', '(', 'seed', ')', 'do', 'i', '=', '1', ',', 'max', 'x', '=', 'dr', 'and', '48', '()', '*', '2', '-', '1', 'y', '=', 'dr', 'and', '48', '()', '*', '2', '-', '1', 'if', '(', 'x', '*', 'x', '+', 'y', '*', 'y', '&lt;', '1', ')', 'then', 'pi', '_', 'count', '=', 'pi', '_', 'count', '+', '1', 'end', 'if', 'area', '=', '4', '.', '0', '*', 'real', '(', 'pi', '_', 'count', ')', '/', 'real', '(', 'i', ')', 'end', 'do', 'pi', '=', '4', '.', '0', *', 'real', '(', 'pi', '_', 'count', ')', '/', 'real', '(', 'max', ')', 'end', 'function']</pre>	<pre>['declaration__', 'parameter_declaration', 'parameter_declaration', '__declaration', 'declaration', 'declaration__', 'parameter_declaration', '__declaration', 'call_expression', 'declaration', 'declaration__', 'parameter_declaration', '__declaration', 'assignment_expression', 'binary_expression__', 'binary_expression__', 'call_expression', '__binary_expression', '__binary_expression', 'binary_expression__', 'binary_expression__', 'call_expression', '__binary_expression', 'call_expression__', 'binary_expression__', 'binary_expression__', 'binary_expression', 'binary_expression', '__binary_expression', .....</pre>
<b>Avg. # tokens</b>	753.04	<b>179.55</b>
<b>Total # tokens</b>	50K	

BPE: Byte Pair Encoding, a core tokenizer used in many leading LLMs

NLTK: Natural Language Toolkit tokenizer

AST(xSBT: abstract syntax tree built by the “Simple Build Tool”

# Our "Tokompiler" applied to HPCorpus-Fortran

## Generate Variable-Anonymized Code:

Create a version of the original code with anonymized variable names, numbers, and strings

## AST Parsing:

Parse the anonymized code using tree-sitter or any suitable parser to generate an AST (Abstract syntax tree)

## Recreate AST Changes:

Update the AST to reflect changes made during anonymization. Keep a dictionary of all changes done per file/function to facilitate restoring semantics back later

## AST to Code-Tokenize:

Transform the updated AST back into code, eliminating any comments, new lines, and READMEs that may have been introduced during anonymization. This code-tokenized version will have a much smaller number of tokens

## Token Splitting:

Split multi-part tokens (e.g., "var\_1" to ["var", "1"]) to ensure that the model comprehends variable names as a combination of type and a unique identifier

## Random names for recurrent tokens:

For recurrent tokens (e.g., "var\_1" or "num\_2"), use random numbers during each tokenization. The attached numbers are randomly chosen without any relation to the type or order of the replaced tokens or the file/function length

Tokenized code	Tokenized AST
<pre>[ 'function', 'func', '48', '(', 'arg', '128', ',', 'arg', '807',   ')', 'result', '(', 'func', '180', ')', 'implicit', 'none',   'integer', ',', 'intent', '(', 'in', ')', '::', 'arg', '128', ',',   'arg', '807', 'real', '(', 'num', '5', ')', '::', 'func', '180',   'real', '(', 'num', '5', ')', '::', 'var', '377', ',', 'var', '84',   ',', 'var', '967', 'integer', '::', 'var', '821', 'external', '::',   'func', '123', 'integer', '::', 'var', '63', 'var', '63', '=',   'num', '156', 'call', 'var', '719', '(', 'arg', '807', ')', 'do',   'var', '821', '=', 'num', '315', ',', 'arg', '128', 'var', '84',   '=', 'func', '123', '(', ')', '*', 'num', '357', '-', 'num',   '315', 'var', '967', '=', 'func', '123', '(', ')', '*', 'num',   '357', '-', 'num', '315', 'if', '(', 'var', '84', '*', 'var', '84',   '+, 'var', '967', '*', 'var', '967', '&lt;', 'num', '315', ')',   'then', 'var', '63', '=', 'var', '63', '+, 'num', '315',   'end', 'if', 'var', '377', '=', 'num', '539', '*', 'func',   '937', '(', 'var', '63', ')', '/', 'func', '937', '(', 'var', '821',   ')', 'end', 'do', 'func', '180', '=', 'num', '539', '*',   'func', '937', '(', 'var', '63', ')', '/', 'func', '937', '(',   'arg', '128', ')', 'end', 'function' ]</pre>	<pre>'translation_unit', '(', 'declaration', '(', 'function',   'function_declarator', '(', 'func', '48', 'parameter_list', '(', '(',   'parameter_declaration', '(', 'arg', '128', ')', ',',   'parameter_declaration', '(', 'arg', '807', ')', ')', ')', ')',   'declaration', '(', 'macro_type_specifier', '(', 'result', '(', 'type',   'descriptor', '(', 'func', '180', ')', ')', 'implicit', ')',   'declaration', '(', 'none', 'integer', ',', 'function_declarator', '(',   'intent', 'parameter_list', '(', '(', 'parameter_declaration', '(',   'in', ')', ')', ')', ')', '(', '::', 'arg', '128', ')', ',', '(', 'arg', '807',   'function_declarator', '(', 'real', 'parameter_list', '(', '(',   'parameter_declaration', '(', 'num', '5', ')', ')', ')', '::',   'func', '180', 'function_declarator', '(', 'real', 'parameter_list',   '(', '(', 'parameter_declaration', '(', 'num', '5', ')', ')', ')', '::',   ')', 'var', '377', ',', 'var', '84', ',', '(', 'var', '967', 'integer', '::',   'var', '821', 'external', '::', 'func', '123', 'integer', '::', 'var',   '63', ')', 'init_declarator', '(', 'var', '63', '=', 'num', '156', ')', ')',   'declaration', '(', 'call', 'function_declarator', '(', 'var', '719',   'parameter_list', '(', '(',   ...</pre>
<b>687.72</b>	1099.1
177 primitives + 1000 locals = <b>1177</b>	

From 50K tokens to under 2K. That is a huge improvement

# An established LLM for generating code

PolyCoder is an open source transformer model based on GPT-2 dedicated to generating code

<b># Parameters</b>	2.7B
<b>Model Size</b>	11.7GB
<b>Transformer Blocks</b>	32
<b>Attention Heads</b>	32
<b>Hidden Dimension</b>	2560

Technical details of PolyCoder model.

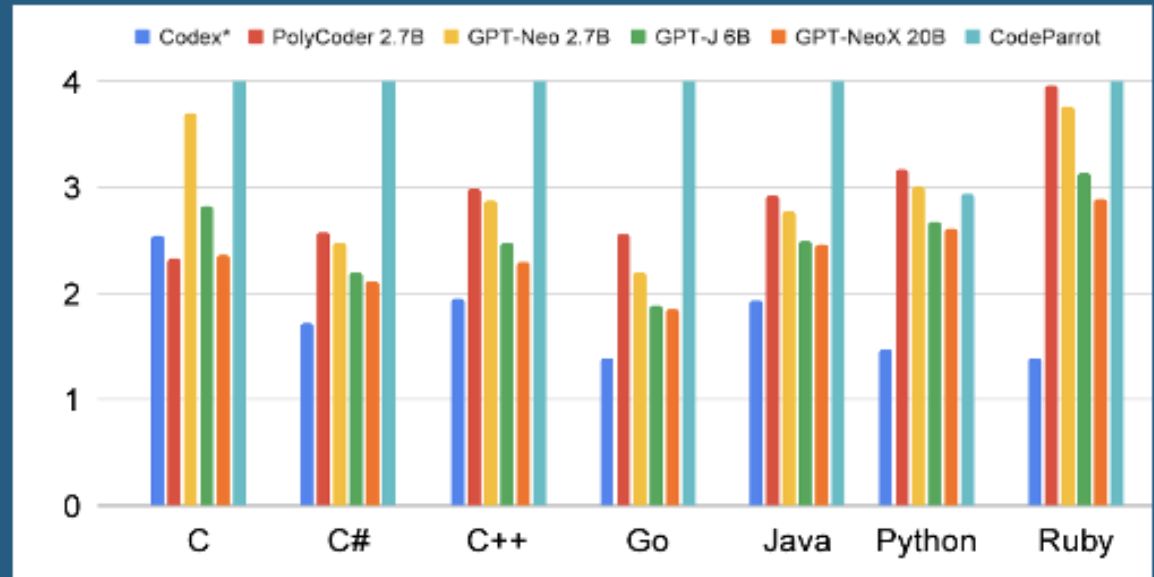
Source: A Systematic Evaluation of Large Language Models of Code (2022), <https://arxiv.org/abs/2202.13169>

Frank Xu, Uri Alon, Graham Neubig, and Vincent Hellendoorn: <https://github.com/VHellendoorn/Code-LMs>

## PolyCoder Corpus Statistics



Python JavaScript Java C C# C++ Go Others



Perplexity comparison.

Perplexity is a measure of model quality: Lower is better

# Tokompiler and Polycoder

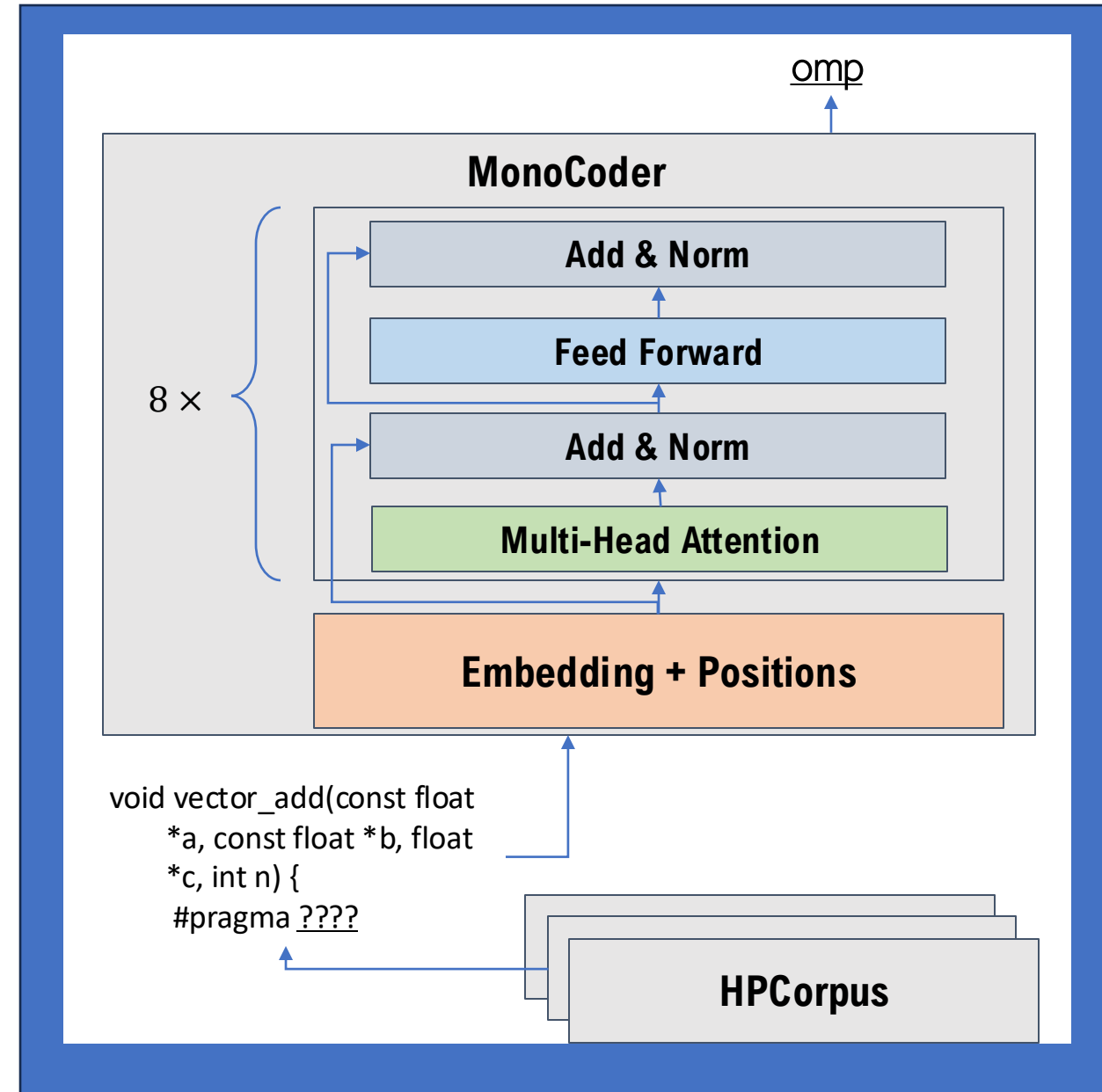
	PolyCoder		Model matched to number of tokens
	BPE	Tokompiler	Tokompiler-small
Model Size	2.8B	<b>2.5B</b>	<b>638M</b>
Time to train (min)	8300	<b>8125</b>	<b>6386</b>

- BPE: the tokenizer used in GPT2 and LLMs for programming tasks (PolyCoder)
- Integrated Tokompiler with PolyCoder
- The number of tokens maps onto the number of parameters in the model, so we were able to build a small model to match the reduced set of tokens
- Training done on 4 A40 GPUs with 48 Gbytes.

# MonoCoder: A model trained on C/C++/Fortran and using our Tokompiler

MonoCoder draws inspiration from the PolyCoder model but reduces the number of transformer blocks by four ... which should be OK since Tokompiler reduces the number of tokens

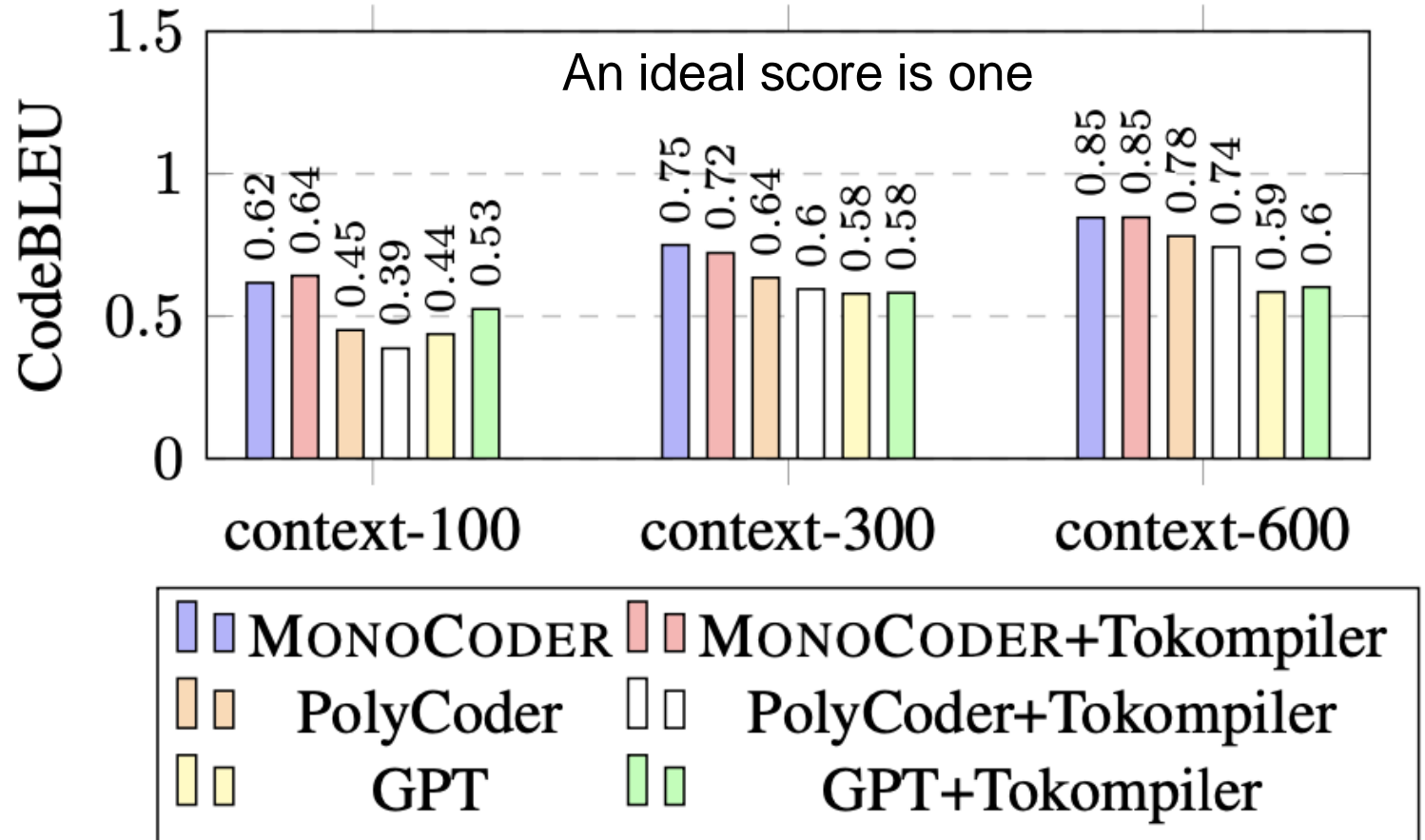
<b># Parameters</b>	0.9B
<b>Model Size</b>	3.6GB
<b>Transformer Blocks</b>	8
<b>Attention Heads</b>	32
<b>Hidden Dimension</b>	2560





# MonoCoder Results: code completion

CodeBLEU is a standard metric of quality in code generation systems ... given a number of tokens for context, how does the system perform with a code completion task ... shown here for 100, 300 and 600 tokens in a 1200 token program.

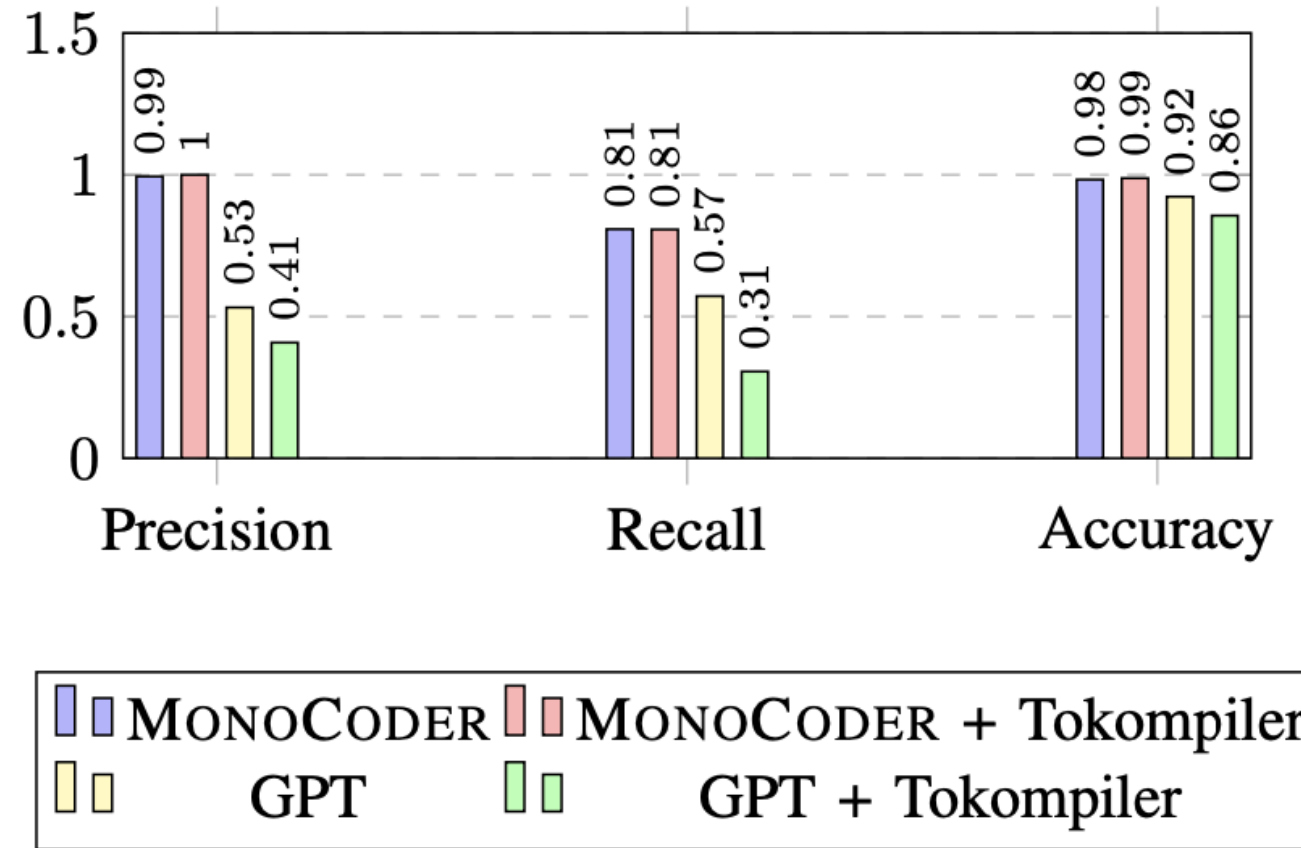


For a general code dataset

# MonoCoder Results: Inserting OpenMP reductions

Trained models on OpenMP code then test them with a data set of OpenMP programs with pragmas removed.

- **Precision:** ratio of correct cases to the sum of correct and incorrect cases.
- **Recall:** ratio of correct cases to sum of correct and missed cases
- **Accuracy:** overall correctness of cases



On a dataset generated from OpenMP codes with pragmas removed, find correct locations and the right form for reduction clauses

# LLMs for parallelism have a long way to go, but we are making progress

- We have shown the value of “less is more” ... training models with languages you do not need degrades the quality of the system.
- We have shown our Tokenizer is smaller and delivers better results than tokenizers designed for natural language processing.
- We believe we can greatly improve our models by working on their ability to use abstract syntax trees and dataflow graph representations of the code.
- Our models work for code completion tasks so we anticipate eventually creating a code recommendation plug-in to aid programmers in developing OpenMP code.
- We have preliminary work showing these approaches work with MPI as well ... though this work was not discussed here.

## ... I see trouble ahead

- LLM methods require vast amounts of data.
- We can find plenty of parallel code from which we can extract data to train our models that add parallelism constructs ... but inserting parallel constructs into code is the easy part of the parallel programming process.
- The hard part ... and this is where software developers need the most help ... is the changes in algorithms, data structures, and program structure to support parallelism.
- We can't find much data on that part of the problem.

No data → No model == ultimate failure

- To restructure code in preparation for parallelism, we need to think like a compiler. Symbolic reasoning, code as a graph that we can cluster and reorganize, and more.

# In closing ...

- The future: Market forces will drive aggressive growth in hardware complexity
- What should we do about this?
  - **Processors:** Embrace hierarchical heterogeneity as you design application software. Chiplets and optical networking makes this an inevitable trend.
  - **People:** Stop whining ... developers are moving to high level languages that hide the hardware. Stop worrying and learn to love Python.
  - **Programming:** We must invest in programming technologies that let people do what they do best (imagine new applications and the math/algorithms needed to support them) and automate what we can in the backend (mapping algorithms and data structures onto the hardware).