A case for parallel-first languages in a post-serial, accelerated world

Paul Sathre
Research Software Engineer
Synergy Lab & NSF Center for Space, High-Performance and Resilient Computing
Virginia Tech

VIRGINIA TECH

SyNeRG
synergy.cs.vt.edu

Disclaimers

- Imperfect tools are better than no tools, and doing serious science with them should be lauded, even as we strive to make better ones.
  - Critical feedback does not diminish the value that prior art has given us
- All thoughts and opinions expressed are my own, and shouldn’t be attributed to my employer, coworkers, or sponsors
- Logos and product names belong to their respective rights holders, and are neither an endorsement of this talk, nor of the referenced products

My Background → portable accelerated computing

- Undergrad:
  - Went in with a high-end desktop for “course work”
    - Quad-core 64-bit AMD, NVIDIA 8800 Ultra GPU, ***Aegia PhysX physics processing unit*** (PPU)
  - Roadrunner just coming online → friends & lab with PlayStation 3s
  - Brief stint doing bioinformatics on Tilera Tile64
  - GPGPU via OpenCL → two summers at LANL (cosmo + neutron transport)

- Grad school and shortly post-MS: CUDA ↔ OpenCL Interop
  - ***MS Thesis***: CUDA to OpenCL translation → (***CU2CL***) on NVIDIA GTX 480
  - Interoperable MPI+{CUDA, OpenCL, OpenMP} for Micro-Air Vehicles → (***MetaMorph***)
    - Running on Nvidia GPU, AMD GPU, Intel MIC, respectively

- Since then: languages and tools for *modern* heterogeneous HPC
  - OpenCL support for FPGA: Linters (***FLOCL***) and autogenerators (***MetaCL***)
  - SYCL for irregular apps: AMD+NVIDIA GPU (via AdaptiveCpp), Intel FPGA (via DPC++)
  - GPU Chapel for irregular apps: perf./prod. tradeoffs → partitionability → portability (soon)

What I really care about:

Closing the gaps between the
parallel hardware we already have,
and ***the people*** who could benefit from it

So how do we ***enable*** them?
(Conversely, what are the ***barriers*** to use?)

Setlist

- Intro [you are here]
- Act I: Parallelism is everywhere, start acting like it
- Act II: The rise of GPUs, up through today
- Act III: Chapel’s role in our GPU future, and our role in Chapel’s
- Outro

ChapelCon #1: Looking back on ten CHIUW keynotes

- GPGPU was already in mind at CHIUW #1
- Python (and later Jupyter) *interactive* dev flows are important
  - *Keep turns within the human thought loop*, whether compilation or analysis
- Need a middle ground between FAANG-scale frameworks which often don’t scale *down* well, and laptop-scale which often don’t scale *up* well
- Analysis, Viz, Packaging, *community-alignment* are all important
- PGAS tends to beat explicitly-distributed when it comes to network-perf and productivity
- Flexibility and performance are *more important than transparency*
  - Start high level, but keep access and provide a smooth ramp to greater complexity

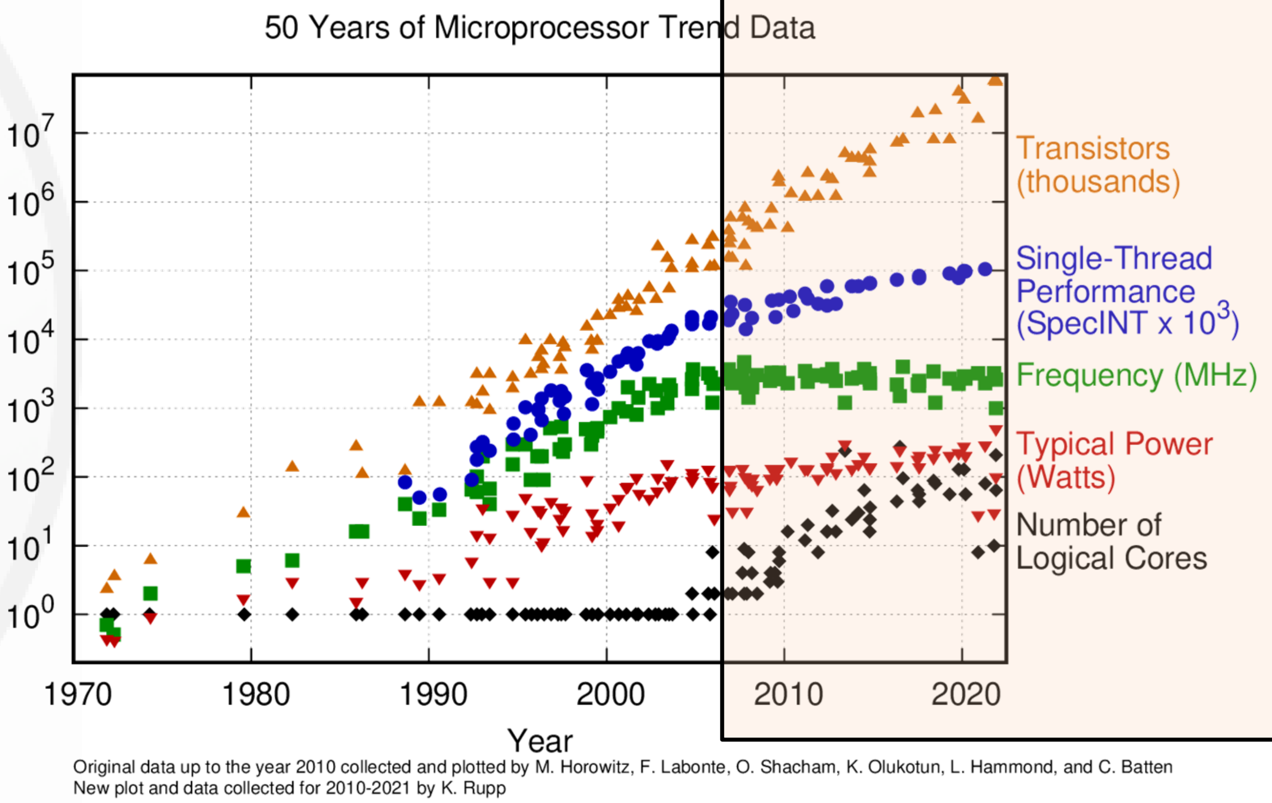
Act I

Parallelism is everywhere, start acting like it

Hardware is parallel, and likely to stay so

- Serial performance has barely improved since I started
- Parallel hardware was already common, now ubiquitous
  - Try to buy a laptop or cellphone without at least dual-core
- HPC has been using parallel and distributed software, but it's still not very ***general***
- Hardware is also ***heterogeneous***, more on that in Act II

My cohort,
and ***all to come***



HPC moved to parallel, distributed,
and heterogeneous long ago

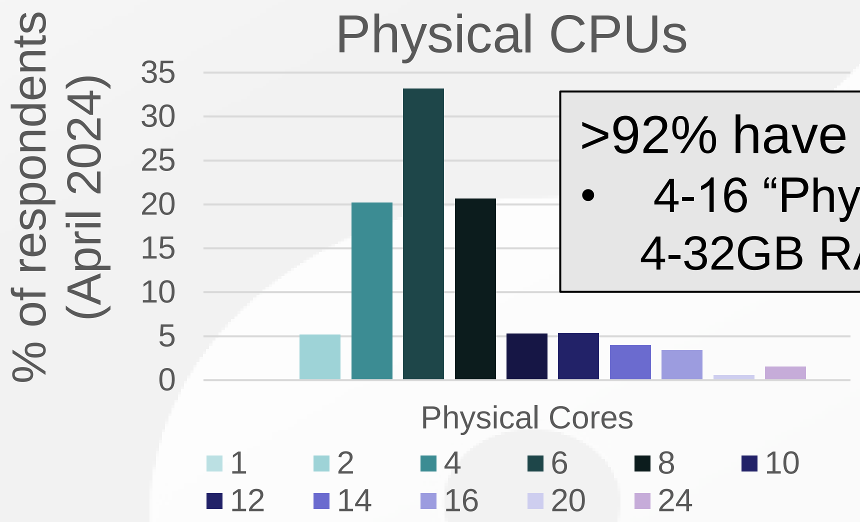
But ***nobody starts programming on an HPC cluster***
They start on a laptop/desktop

So what are *regular users* using?

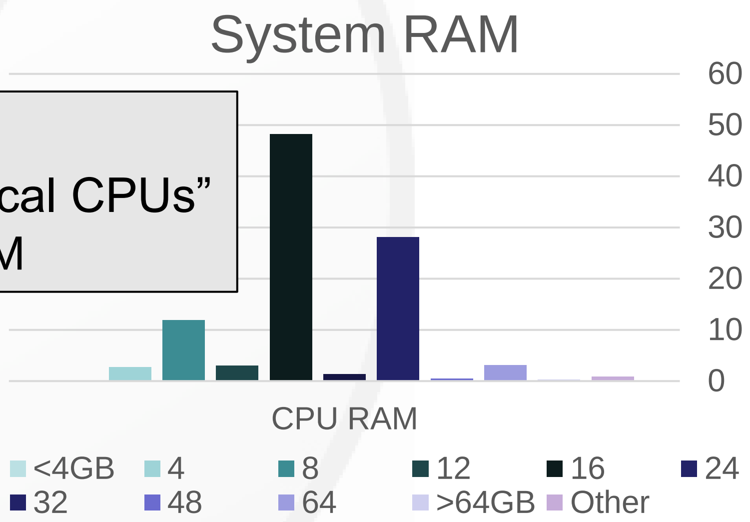
What do desktop and laptop users actually have?

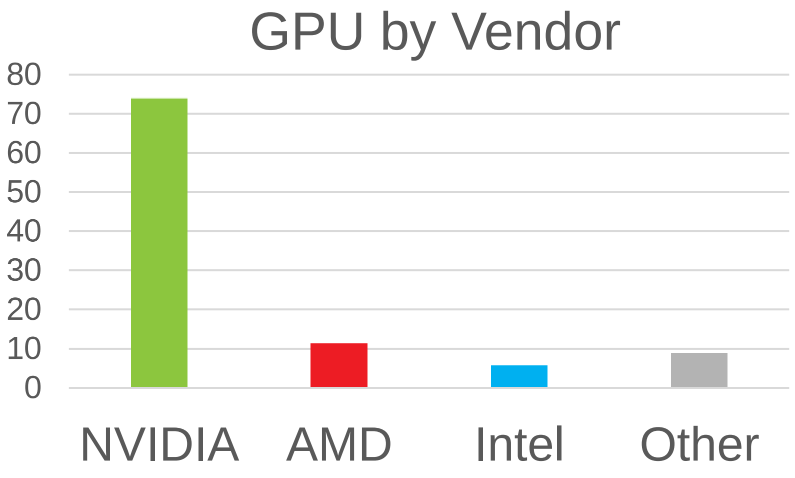
Steam hardware survey → rough proxy for general-purpose users

– In reach of {hobbyists, tinkerers, undergrads} → ***future HPC buyers***



>92% have
- 4-16 “Physical CPUs”
  4-32GB RAM





More in Act II

At least 68.17% ***CUDA-capable*** GPUs according to: https://developer.nvidia.com/cuda-gpus

At least 0.38% ***ROCm-capable*** GPUs according to: https://rocm.docs.amd.com/projects/install-on-linux/en/docs-6.1.1

Data collected 5/28/24 from https://store.steampowered.com/charts/

HPC is changing, *broadening*

- Users and developers are more mobile/remote
  - Less cubicle high-power workstation, more laptop / BYOD / tablet / cell
  - Are centrally-installed dev environments still the norm? Or more individual?
- Multicore + GPU at home in gaming / streaming / editing rigs
  - Everybody's a Twitch, YouTube, insert-platform-here star
- It's not just the privately-owned datacenter anymore → lots of IaaS
  - Renting cycles as needed vs. surpassable Big Iron cap-ex
  - Anyone with a credit card can buy GPU cycles for AI, crypto, etc.
- Need a ***unified*** approach to programming ***all of it***
  - CPUs, ***GPUs,*** NICs, and *whatever comes next*

So regular users have (or can get) ***parallel hardware***

But isn't ***parallelism*** hard to understand?

The natural order of the world is *massively parallel!*

- Humans have innate *experiences* and *understanding of parallel processes*
  - Beehives → Scatter/Gather
  - School of fish, flock of sheep → single instruction (sheepdog), multiple thread (sheep)
  - Check-out lines → task parallelism and work stealing
  - Road networks → numerous parallel and sync constructs

Pipelined Parallelism
(SIMD *LANES*)

Memory Fence
(toll *STORE*)

Semaphore
(controlled intersection)

We have broad access to ***parallel hardware***

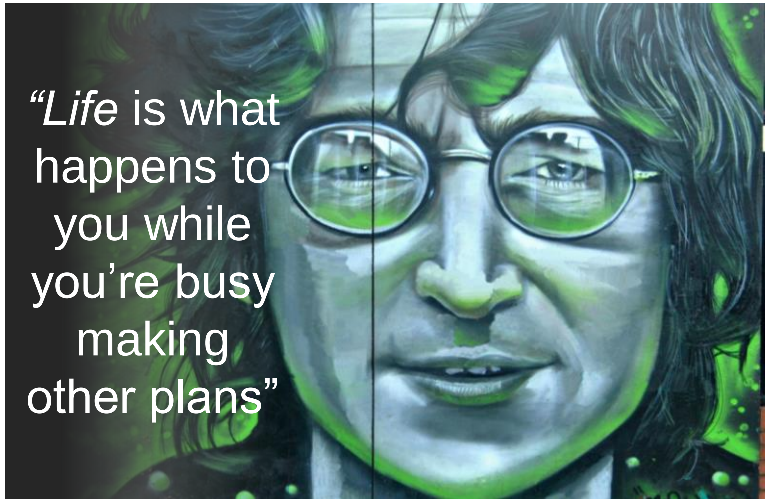
Humans experience ***parallel phenomena***
in their day-to-day lives

Why does it still seem ***so hard***?

Because we teach ***serial first***, through the
lens of old, ***post-serial*** norms!

Reality and hardware are parallel, so teach that

- Even "serial" threads ***don't run in isolation***
  - OS time-slicing, async IO, ISP/power outages
- Teach how to be safe and effective in a ***parallel world***
  - New driver → ***defensive driving***
  - New programmer → ***thread-safety*** and ***fault-tolerance***



- Should be teaching async, concurrent, and parallel ***to all, and earlier!***
  - ***After*** C/systems sequence, as an ***elective*** is ***too late*** → serial habits already anchored
  - To move earlier, need a language that is easy, ***parallel-first,*** and ***general***
- PGAS is more approachable for learning ***distributed***
  - Just some "further" cores/mem with more latency and failure modes
  - Don't snail-mail a co-located co-worker! Use a whiteboard, post-its, Kanban, *lunch meeting*

Post-serial or "serial with sprinkles"

- Dominant programming models are still ***post-serial***
  - "Sprinkles": ***optional*** libraries, pragmas, language extensions

- Chapel presents a different option: ***parallel-first***
  - A non-separable part of the *keywords*, *data abstractions*, and *semantics* of the language (i.e. **promotion**)

- Why ***parallel-first*** matters?
  - Philosophical: ***realign*** languages to hardware, ***demystify*** parallelism
  - Technical: better ground-up ***parallel safety*** built into its fabric
  - Technical: ***no conflicts*** between base and parallel sprinkle, ***they grow together***

- ISO/IEC 14882:2011, *Information T*

This OpenMP API specification refe
of the OpenMP specification are exp
may result in unspecified behavior.

OpenMP 5.0 spec. (2018)

Code Example: CPU vector add

Serial C

```c
void vecAdd(float *A, float *B,
 float *C, int n) {
  for (int i=0; i<n; i++) {
    C[i] = A[i]+B[i];
  }
}
```

OpenMP

```c
#include <omp.h>
void vecAdd(float *A, float *B,
 float *C, int n) {
  #pragma omp parallel for
 shared(A, B, C) private(i)
  for (int i=0; i<n; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Chapel (without promotion)

```chapel
proc vecAdd(A: [] real(32),
  B: [] real(32), C: [] real(32))
  {
  forall i in C.domain {
    C[i] = A[i] + B[i];
  }
}
```

A vector is just a *linear collection of things*.

Should we express our code according to an *individual element's experience*?

Or do we actually care about the *collective*?

This is closer to the mental model

Chapel

```chapel
proc vecAdd(A: [] real(32),
  B: [] real(32), C: [] real(32))
  {
  C = A + B; //promoted
}
```

Intermezzo

- The world and modern hardware are parallel, ***let's start acting like it***
  - Use parallel-first languages, and ***teach them to new users***
  - Move our mental model from "The Hero [thread]'s Journey", to "shepherds of threads"

- But we also have to think about ***heterogeneous parallelism...***

Setlist

- Intro
- Act I: Parallelism is everywhere, start acting like it
- [you are here]
- Act II: The rise of GPUs, up through today
- Act III: Chapel’s role in our GPU future, and our role in Chapel’s
- Outro

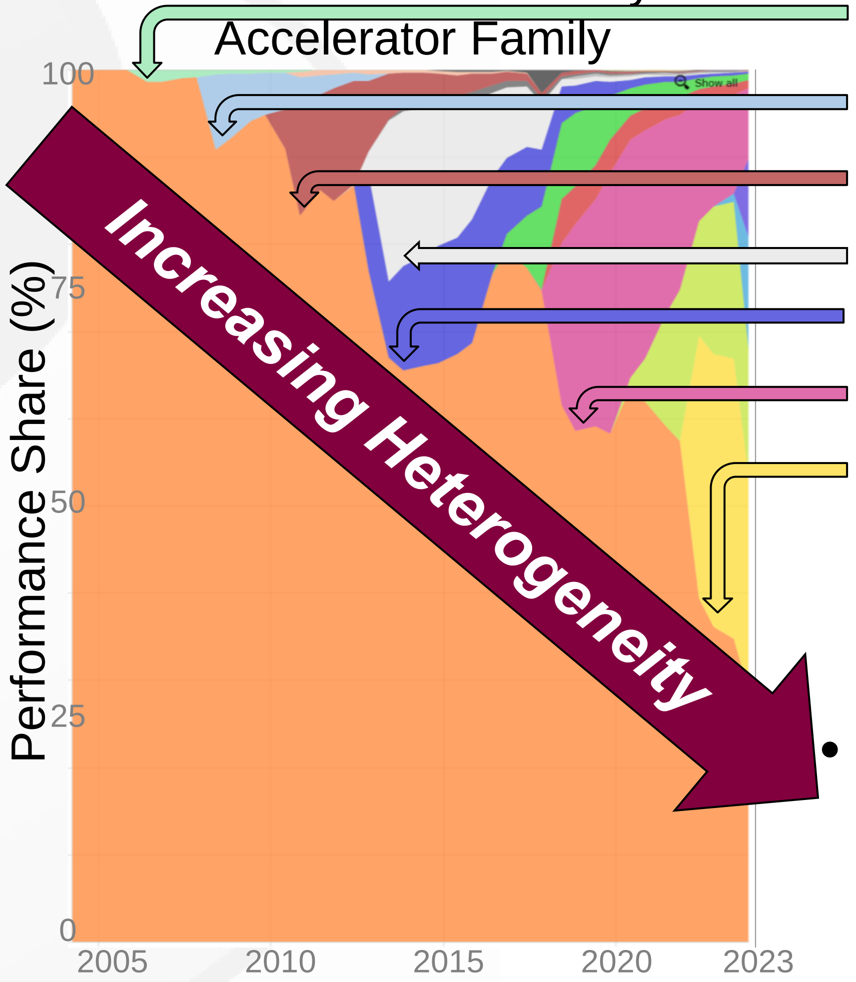
Act II

The rise of GPUs, up through today

Hardware is ***heterogeneous***

- *Heterogeneous* hardware is ubiquitous (and *has been*)
  - {floating-point-, graphics-, physics-, signal-, crypto-, tensor-, data-, vision-, reconfigurable-, ...} Processing Units
  - Some get married to the CPU as a SoC, some don't
    - NVIDIA Grace Hopper, AMD MI300A APU, AMD Versal FPGA, Altera Agilex FPGA, etc.

Performance Share by Accelerator Family



(Source: Top500.org, June 2004 – Nov 2023)

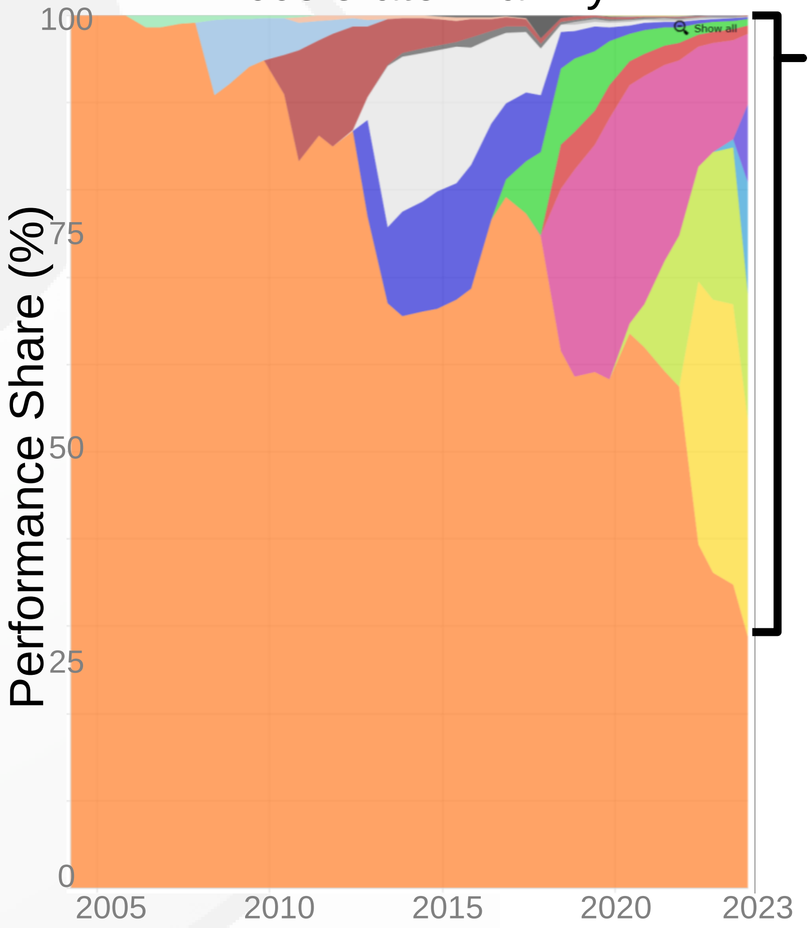
June 2006: Clearspeed (1.4%)
June 2008: IBM Cell (8.6%)
Nov. 2010: NVIDIA Fermi (12.7%)
Nov. 2013: Intel Xeon Phi (17.7%)
+ NVIDIA Kepler (12.2%)
Nov. 2018: NVIDIA Volta (23.7%)
June 2022: AMD MI250 (30.2%)

- Accelerator *du jour* keeps shifting vendor & language

Hardware is ***heterogeneous***

- Accelerator *du jour* keeps shifting vendor & language
- ***Portable*** languages necessary to reach everything and ***reduce rework!***

Performance Share by Accelerator Family



(Source: Top500.org, June 2004 – Nov 2023)

November 2023:
70.1% GPU

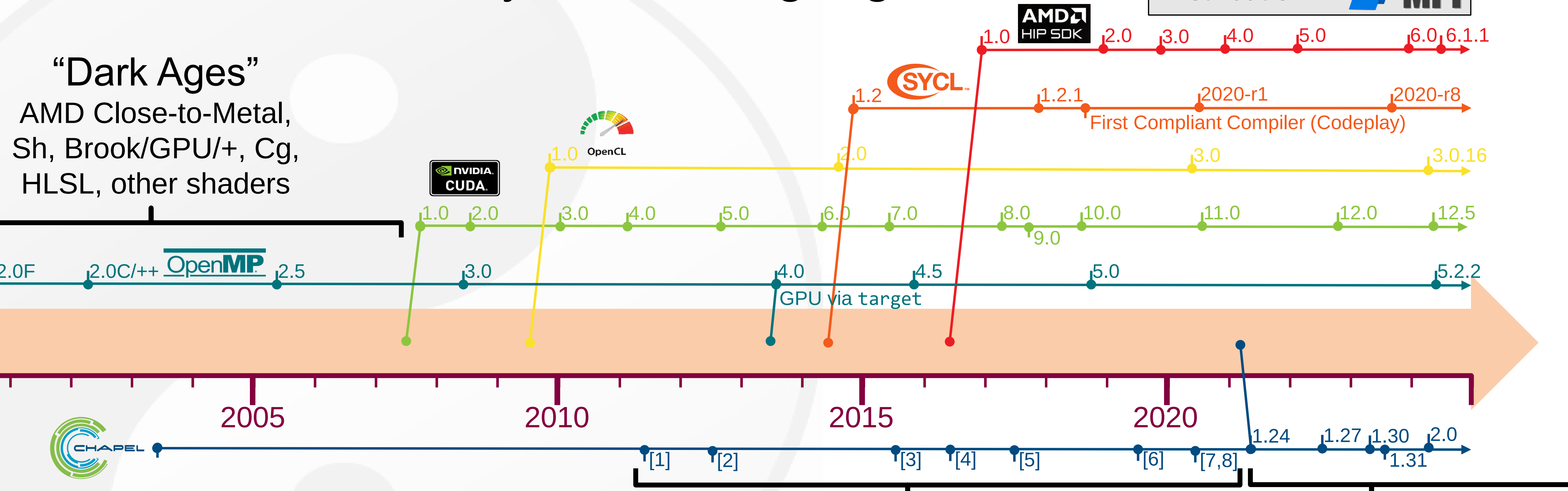
By vendor:

- NVIDIA: 36.3%
  - Ampere: 14.3%
  - Hopper: 12.9%
  - Volta: 8.2%
- AMD: 24.9%
  - MI250: 24.9%
- Intel: 9.0%
  - Max Data Center: 8.9%
- No Coprocessor: 28.8%

Timeline of today's GPU languages*

Distribution → MPI

“Dark Ages”
AMD Close-to-Metal, Sh, Brook/GPU/+, Cg, HLSL, other shaders



Heroic Efforts

Official GPU Support
1.24 Low-level NVIDIA
1.27 Multi-locale NVIDIA
1.30 AMD via ROCm 4.X+
1.31 Multi-locale AMD

[1] Sidelnik, A., et al. "Using the high productivity language chapel to target GPGPU architectures." Tech Report. 2011.
[2] Sidelnik, A., et al. "Performance portability with the chapel language." IPDPS. 2012.
[3] Breternitz, M., et al. “A Progress Report on COHX: Chapel on HSA+XTQ.” CHIUW. 2015
[4] Pan, A. & Chu, M. “Chapel-on-HSA: Towards seamless acceleration of Chapel programs using HSA.” CHIUW. 2016
[5] Chu, M., et al. "GPGPU support in Chapel with the Radeon Open Compute platform." CHIUW. 2017.
[6] Hayashi, A., et al. "GPUIterator: Bridging the gap between Chapel and GPU platforms." CHIUW. 2019.
[7] Ghangas, R. & Milthorpe, J. "Chapel on accelerators." CHIUW. 2020
[8] Hayashi, A., et al. "Exploring a multi-resolution GPU programming model for Chapel." CHIUW. 2020.

*scale approximate

What can we learn from CUDA?

NVIDIA CUDA

Consistency is king

- Pick a model, stick to it, and iterate on it when needed
- Build ***and keep*** knowledge: docs, forums, code examples
  - Dead links are dead ends, insights lost to time

Reach people ***before they learn something else***

- University programs – free training and GPUs – were a smashing success

Give away your tools

- Easy to install and use ***everywhere***

Welcome the small fish, they are many and ***some will grow bigger***

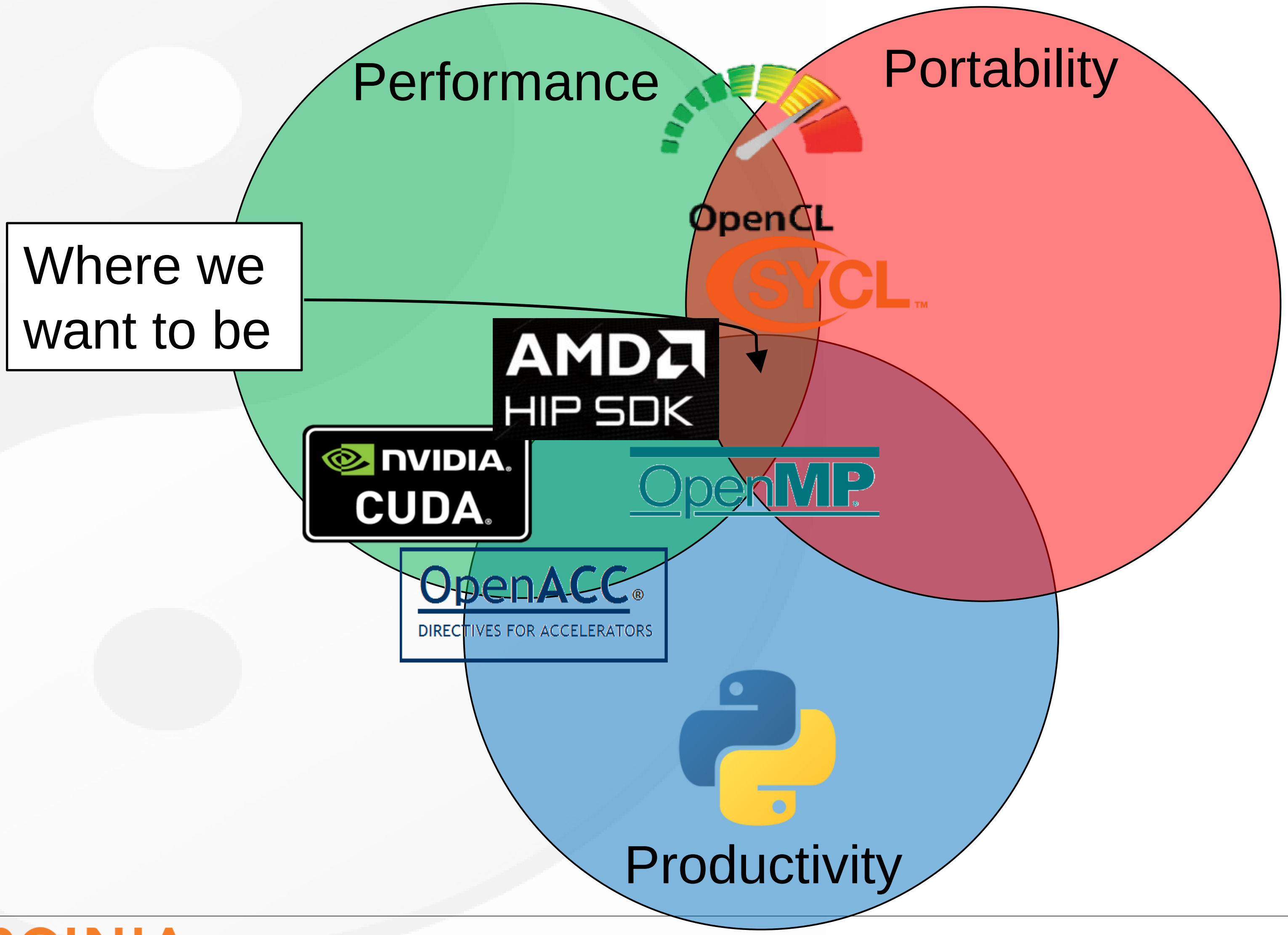
- New codes, users, buyers ***start on laptops*** before growing to datacenter

Recall Act I:
At least 68% of the NVIDIA “gaming” GPUs in Steam’s April 2024 survey support CUDA!

Drive the conversation

- Know where you excel, and *show it to people*

GPU languages have a pick 2 of 3 problem



But where do we fit?

CHAPEL

I haven't gotten to do much portability work yet, just a few spot tests on AMD RDNA3 GPU.

Hope you saw the talks in Session #2!

GPU Productivity Example: Vector Add

- Either: ***concise but implicit*** or ***explicit but verbose***

OpenMP 4.0+

```
#include <omp.h>
void vecAdd(float *A, float *B, float *C, int n)
{
  #pragma omp target teams distribute parallel for simd map(to: A[0:n], B[0:n]) map(from: C[0:n])
  for (int i=0; i<n; i++)
  {
    C[i] = A[i]+B[i];
  }
}
```

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

OpenCL via MetaCL

```
__kernel void vecAddKernel(__global float *A, __global float *B, __global float *C, int nelem) {
  size_t tid = get_global_id(0);
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  meta_set_acc(-1, metaModePreferOpenCL);
  cl_device_id dev;
  cl_platform_id plat;
  cl_context ctx;
  cl_command_queue q;
  meta_get_state_OpenCL(&plat, &dev, &ctx, &q);
  cl_mem A, B, C;
  A = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  B = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  C = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  clEnqueueWriteBuffer(q, A, CL_FALSE, 0, sizeof(float) * nelem, A_h, 0, NULL, NULL);
  clEnqueueWriteBuffer(q, B, CL_TRUE, 0, sizeof(float) * nelem, B_h, 0, NULL, NULL);
  size_t local[3] = {256, 1, 1};
  size_t global[3] = {((nelem / local[0]) + (nelem % local[0] ? 1 : 0)) * local[0], 1, 1};
  metacl_vecAdd_vecAddKernel(q, &global, &local, NULL, false, NULL, &A, &B, &C, nelem);
  //Copy buffers
  clEnqueueReadBuffer(q, C, CL_TRUE, 0, sizeof(float) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```

Kernel boxed in green

Chapel GPU Productivity Example: Vector Add

- Chapel allows you to be **both** *concise* and *explicit*
  - Concise → ***Productivity*** Explicit → ***Maintainability***

Chapel

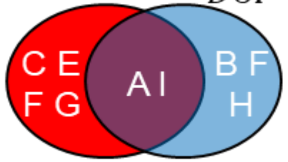
```
use GPU;
proc vecAdd(A_h: [] real(32), B_h: [] real(32), C_h: [] real(32)) {
  on here.gpus[0] {                           //enter 0-th device scope
    var A: [A_h.domain] real(32) = A_h;       //copy-init from host
    var B: [B_h.domain] real(32) = B_h;       //copy-init from host
    var C: [C_h.domain] real(32) = noinit;    //alloc without init
    C = A + B;                                //PROMOTED GPU add
    C_h = C;                                  //copy-from
  }                                           //release mem at GPU scope end
}
```

Kernel boxed in green

Chapel GPU on *Irregular* Apps

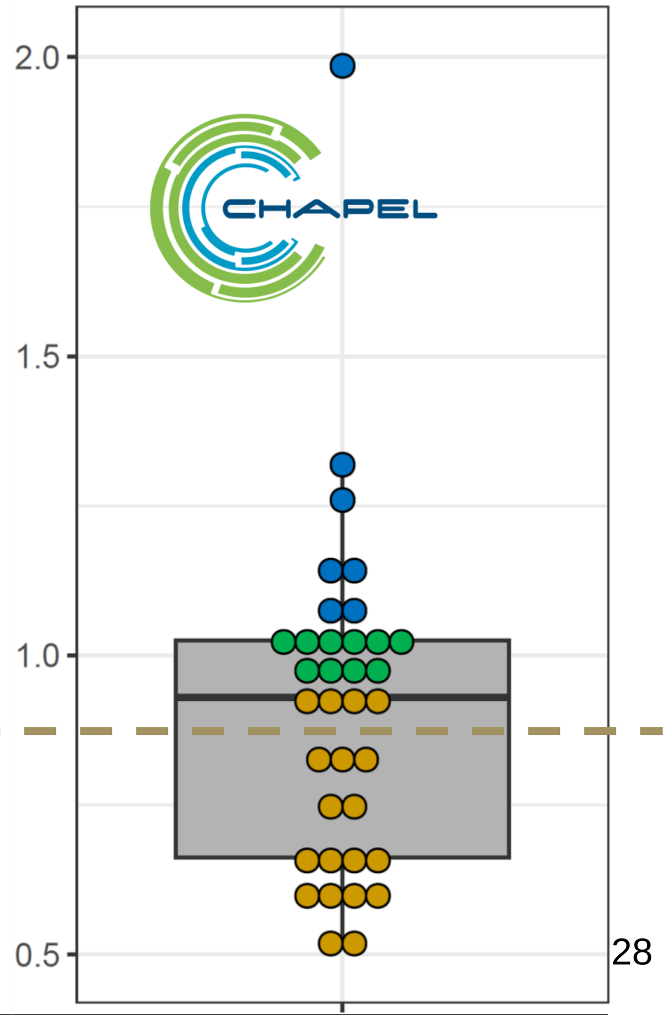
- ***Transliteration*** of CUDA Edge- and Vertex-centric graph analysis pipeline → ***Jaccard Similarity***
  - *See my CHIUW'23 talk for more detail*

$$JS(D,F) = \frac{D \cap F}{D \cup F}$$



- Non-trivial: 3D kernels, atomics, random-access
- Most recent line counts: CUDA ***1212*** vs Chapel ***599*** ***(-51%!!)***
- Relative kernel performance was great!
  - 18 graph inputs (see appendix): **|E|** =~3-500M, **avg. density**:= ~2.1-160
  - Min: **0.51x**, Max: **1.98x**, Geo. Mean: **0.87x**
  - Performance gap on the sparser inputs
  - Performance parity (± 5%) on the denser inputs
  - Performance **gain** on a handful!
- Currently working on partitionable multi-GPU, multi-locale version

Chapel Speedup vs CUDA on RTX 3090



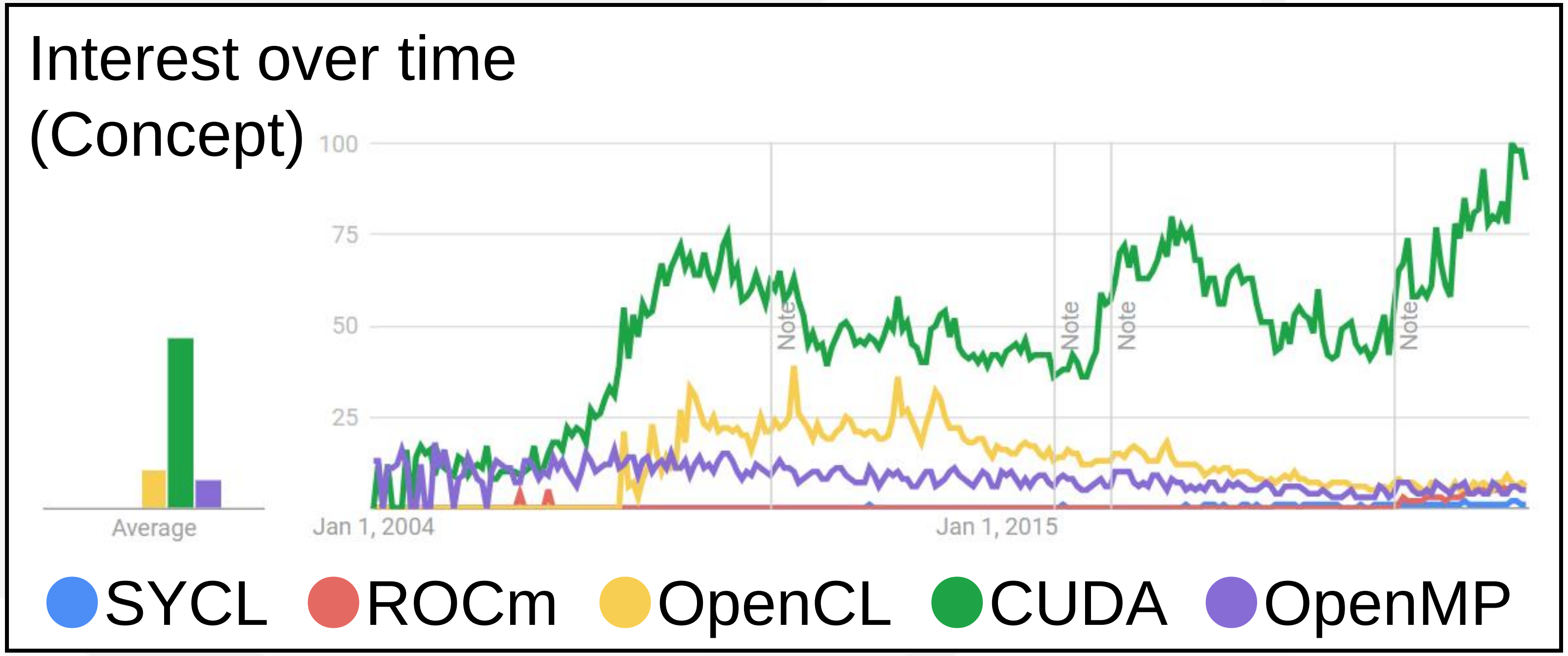
As a language for GPGPU kernels, Chapel is *pretty good!*

More *productive*, more *maintainable*,
similar (and *improving*) *performance*

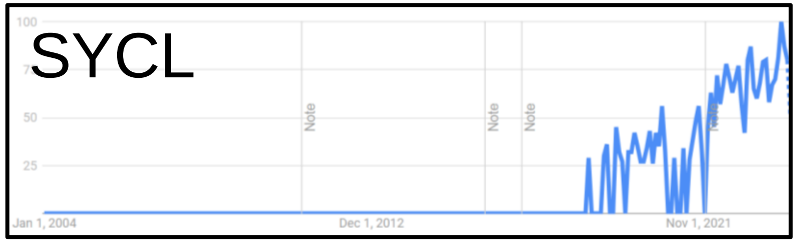
But how are people actually *using* GPGPU?

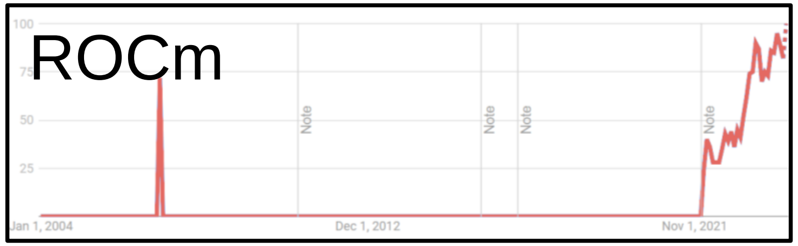
Is it post-serial GPGPU languages?

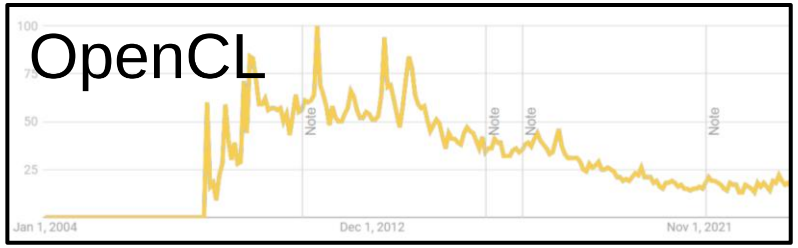
- CUDA clearly the market leader
- OpenMP and OpenCL on the decline
- ROCm and SYCL on the rise

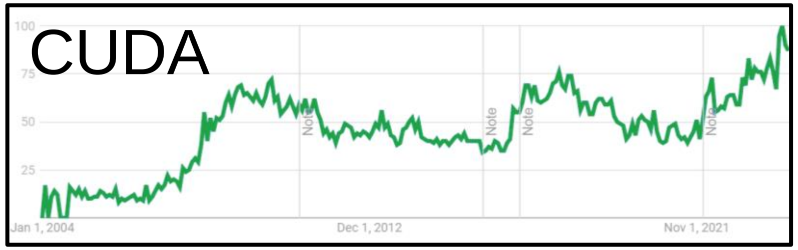


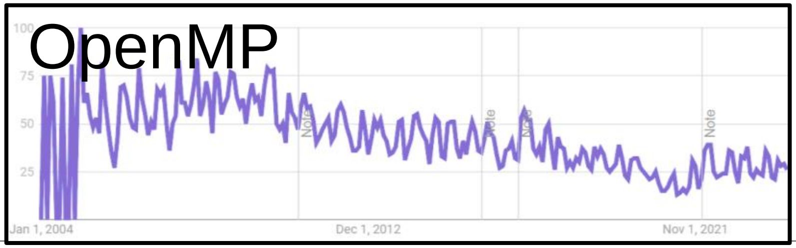
Google Trends Snapshot: 5/23/2024







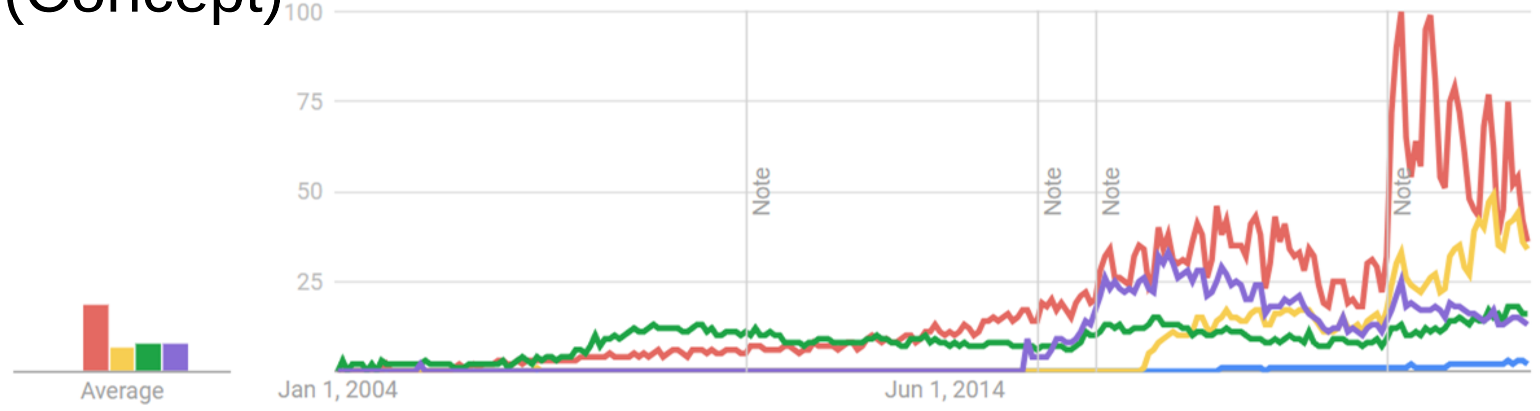




Or is it *Python with Libraries?*

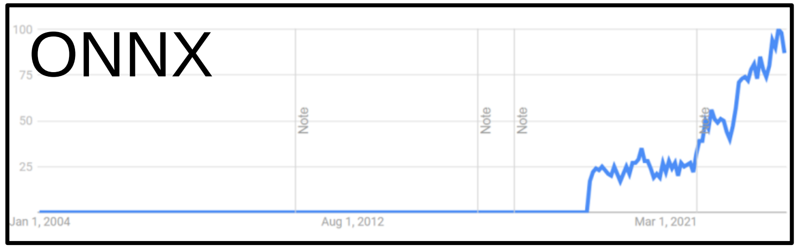
- High-performance Python → NumPy, et al
  - CuPy for GPU w/ compatible API is new, but growing
- TensorFlow & PyTorch compete w/ CUDA
- ONNX is following trends, but w/ lower share

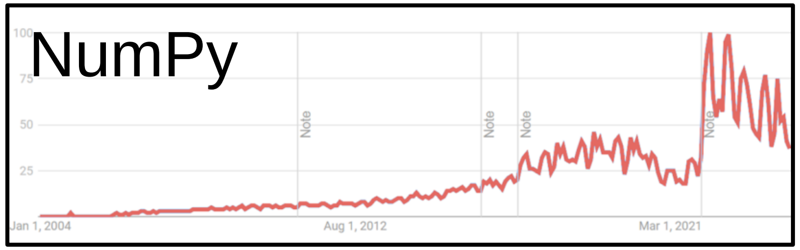
Interest over time (Concept)

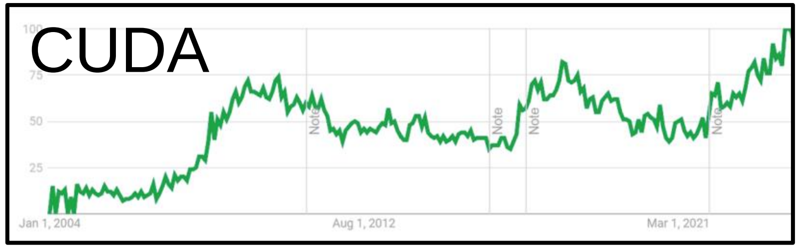


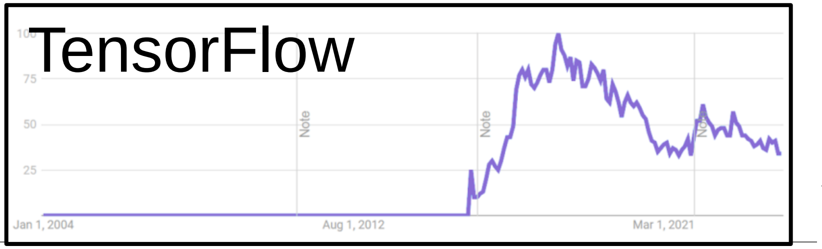
ONNX NumPy PyTorch CUDA TensorFlow

Google Trends Snapshot: 6/5/2024









Intermezzo

- GPUs are critical to modern performance, ***both at exa- and laptop scales***
- Chapel's GPU support is young but ***productive, performant*** and ***portable***
- High-level Python has more ***energy*** & ***interest*** than low-level GPU
  - AI / Data science communities → new ***federated*** frontiers for HPC
- Chapel with GPU could ***fill the gap*** between ***library-driven*** Python and existing ***post-serial*** GPU languages!

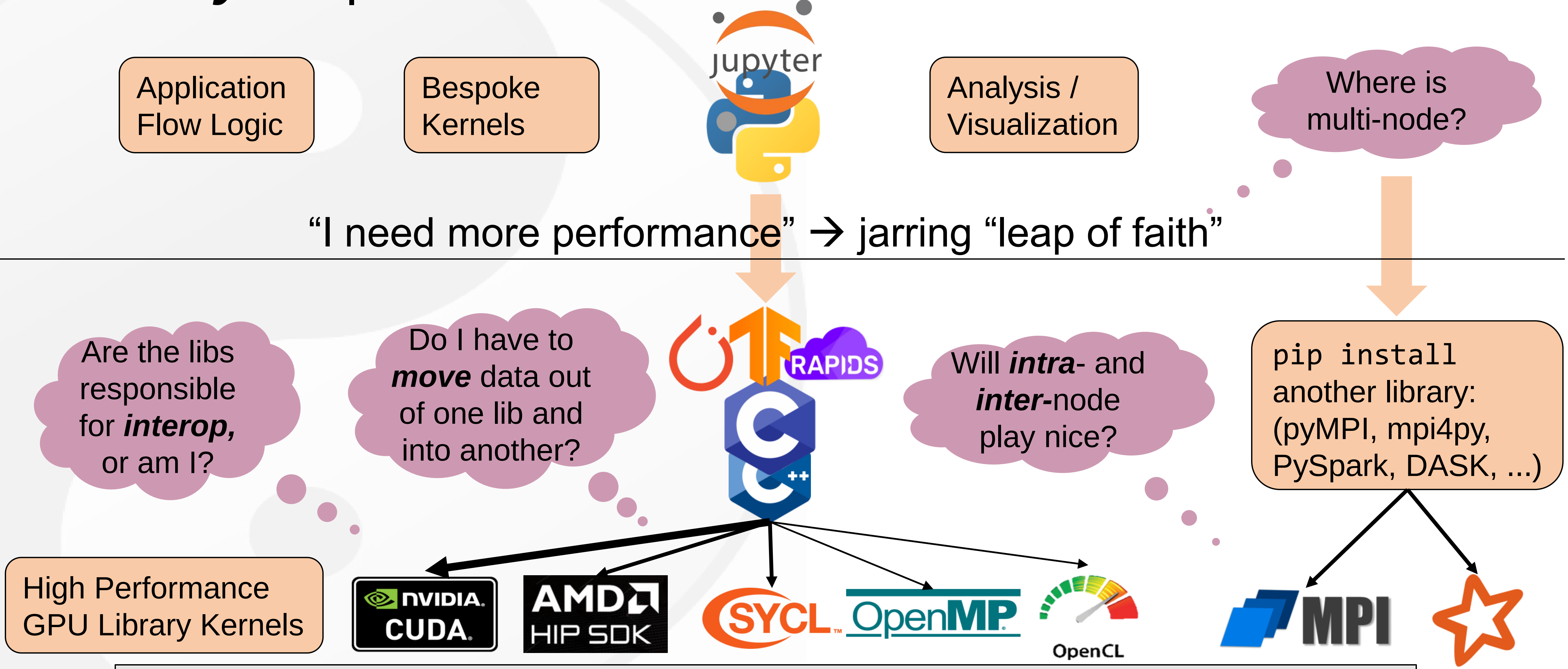
Setlist

- Intro
- Act I: Parallelism is everywhere, start acting like it
- Act II: The rise of GPUs, up through today
- [you are here]
- Act III: Chapel's role in our GPU future, and our role in Chapel's
- Outro

Act III

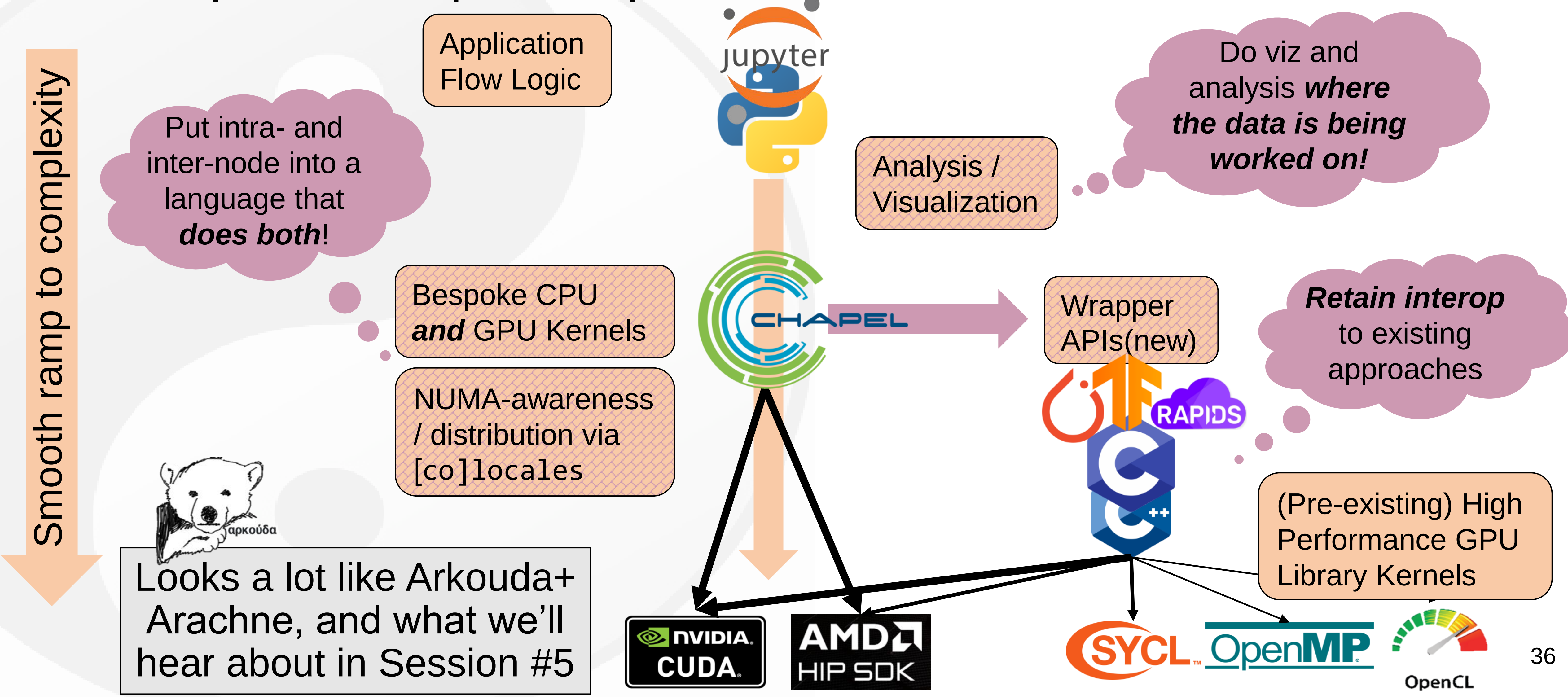
Chapel’s role in our GPU future, and ***our*** role in Chapel’s

A *very* simplified view of AI & Data Science



Too many [inter-]dependencies can lead to ***fragility*** and ***overhead!***

Chapel eases path to parallel/GPU/distributed



Available parallel and GPU Hardware ✓

Parallel-first, ***portable*** GPU Language ✓

AI and Data Science ***energy*** towards GPU ✓

So what is still in Chapel's way?

Inertia ✘ ***Friction*** ✘ ***Exposure***

Getting from A to B → $x_t = x_0 + t*v_0 + t^2*a_0/2$

- Inertia (Position → ***language share***)
  - Old codes & langs are ***battleships***: big, expensive, ***powerful***, moved by committee
  - ***Transliteration*** is necessary to "prove it works" but high effort relative to reward

- Friction (Velocity → ***rate of new codes***)
  - Programming is not just about the code, ***it's about the whole ecosystem!***
    - Installation, editor tooling, documentation, debugging, support forums, visualization, ...
  - Ecosystem ease of use can ***help*** or ***hinder*** adoption
  - Familiar workflows must be ***trivial to reproduce***, or better yet, ***improved upon***

- Exposure (Acceleration → ***rate of change in rate of new codes***)
  - The ***earlier*** and ***more broadly*** you can reach people, the better
  - Users' ***struggles*** will identify friction points, and ***successes*** will feed excitement loop!

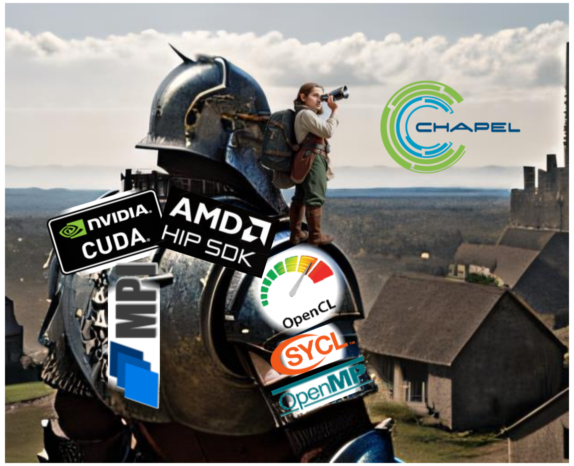
What can we learn from CUDA?

NVIDIA CUDA

- Consistency is king → ***Positive Inertia***
- Reach people ***before they learn something else*** → Pre-Inertia ***Exposure***
- Give away your tools → ***Exposure*** and ***Low Friction***
- Welcome the small fish, they are many and ***some will grow bigger*** → ***Exposure, Low Friction*** and ***Positive Inertia***
- Drive the conversation → ***Exposure***

How are we really doing on these?

Progress on Inertia



- We're building on lessons from post-serial ***giants***
  - Not going anywhere, but we grow as a ***complement to them***
    - ***Higher-level*** for most, dip into post-serial via interop when needed
- Year-over-year we've increased breadth and depth of Chapel codes
  - Most successes are from ***new codes and users***, not transliteration!
- Rigorous, at-scale successes are slowly chipping away
- I think it wins on clarity for maintainability, but I'm not sure there's enough Chapel-native developers for a robust HR ecosystem (yet)
  - It's not just the code, it's *the availability of people who know how to work on it*

New codes and app. areas in Session #4!

Friction points

Check out Session #1!

- ***2.0 improved*** stability, docs, diags.! Work ongoing on LSP tools, debugging, interactivity
- ***"Where .exe?"*** Need better install, migration, *in situ* viz → where is our `matplotlib`?

Approach	Deb package?	GPU? Portability?	From source?	New compiler?
OpenMP	Yes	Optional	No	No
MPI	Yes	Advanced only	Advanced only	Yes
NVIDIA CUDA	Yes	NVIDIA required	No	Yes
AMD HIP SDK	Yes w/ Repo	AMD or NVIDIA required	No	Yes
OpenCL	Yes	Optional	No	Host No, Dev. Yes
SYCL oneAPI	Yes w/ Repo	Optional	No	Yes
SYCL AdaptiveCpp	No	Optional	Yes	Yes
Chapel CPU/single-loc	Soon!!	No	No	Yes
Chapel GPU/multi-loc	No	Optional AMD or NVIDIA	Yes	Yes

What about Exposure?

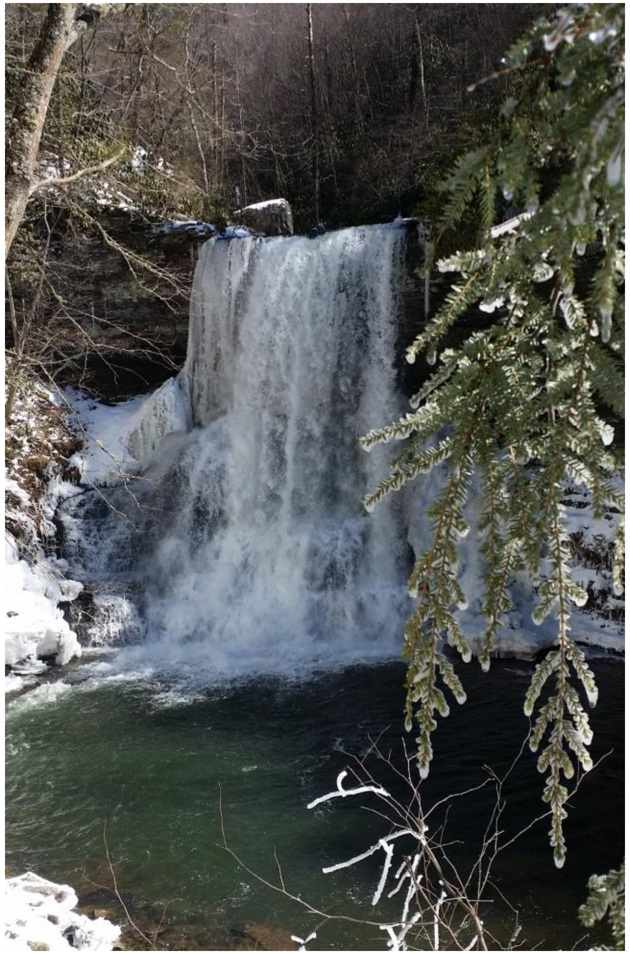
- ***We***'re here, so how'd we get here? → ***Community survey*** might show common themes
- ***Curricula*** are the long term answer, but ***generality*** is a precursor!
  - Right now part of ***inertia***: slow battleships only moved by committees
  - But there are ***other places to teach***: bootcamps, pre-college, online platforms!

Check out Luca's talk ***next*** in Session #3

- 3rd party blogs are ***great*** → independent credibility
- Increased social media needs to align to where prospective users ***already are***
  - There's a ton of CS, AI, Python, Rust, ProgrammerHumor on Reddit, but r/Chapel is quite thin
- Need to reach ***beyond*** HPC bastions, because ***everyone*** has parallel hardware now
  - Instead of "pushing out" from HPC spaces, "pull in" concepts and people from more general spaces

Where to put our efforts? → ***Cascade**s start **upstream***

- ✘ ***Inertia*** is a giant / battleship, strong and slow
  - Small splashes won't move it, but ***riding a tide could → GPGPU!***
- ✓✓ ***Friction*** is a solvable engineering problem
  - Make moving to Chapel ***as easy as drifting downstream***
  - Programmers we've reached have ***interest*** and ***energy*** that we shouldn't ***waste on turbulence***
    - 2.0 stability***,*** LSP tools, ***installation***, interactivity, debugging, ***visualization***
- ✓ ***Exposure*** is important, but not a guarantee!
  - AI / data science devs are gravitating to more ***general*** languages
  - Reach into ***general spaces*** to show ***a better path downstream***
- Focusing on the ***upstream experience*** and ***potential energy*** of newer users, could finally shift the tide to ***parallel-first***



Cascades in winter
Photo Credit: Paul Sathre, 2015

Conclusion

- The real world and everyday hardware are parallel, ***so lets act like it***
  - Use ***parallel-first*** languages, like Chapel and ***teach them to new users***
- Chapel's GPU support learned quickly from prior GPU approaches
  - Now a ***productive, performant,*** and ***portable*** contender
- Let's ***align*** to broader GPU trends towards a future in AI / Data science
  - ***Smooth the gap*** between high-level libraries, and low-level post-serial
- To grow a Cascade, seek ***general*** users & provide a ***low-friction*** experience
  - Reduce barriers to entry, expand reach from HPC to ***general parallelism***

Thanks

- ... to current members and alums of Synergy Lab @ VT
- ... to the Chapel team at HPE for Q&A, feature requests, discussion
  - ... and inviting me to speak today!
  - Particular shoutouts to GPU and IO teams: Brad, Engin, Andy, Ben H., Michael, Lydia
- ... to the Chapel community for fighting hard on inertia, exposure, and friction
- Sponsorship
  - The work detailed herein has been supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC).

Mission-Critical Computing
NSF CENTER FOR SPACE, HIGH-PERFORMANCE, AND RESILIENT COMPUTING (SHREC)

Question & Answer

- The real world and everyday hardware are parallel, ***so lets act like it***
  - Use ***parallel-first*** languages, like Chapel and ***teach them to new users***
- Chapel's GPU support learned quickly from prior GPU approaches
  - Now a ***productive, performant,*** and ***portable*** contender
- Let's ***align*** to broader GPU trends towards a future in AI / Data science
  - ***Smooth the gap*** between high-level libraries, and low-level post-serial
- To grow a Cascade, seek ***general*** users & provide a ***low-friction*** experience
  - Reduce barriers to entry, expand reach from HPC to ***general parallelism***

Encore Slides

Difficult Discussion Questions

- If a business has a Chapel-driven product or internal tool, how quickly can they bring a new grad up to speed to work on the language?
- If we use Chapel's higher-level parallelism for teaching, are we hiding parts of the underlying parallelism that are critical for developing deep understanding?
- After improving base language installation, what is our package ecosystem like? Is / could Mason be as easy as Pip, Cargo, etc.?
- How well and portably do we expose GPU features like threadfence, tensor cores?
- Is a new language more or less rigid than coding to a [large] C/++ framework?

Wishlist

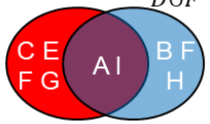
- Viz Viz Viz and Interactivity
- GPU-enabled install packages
  - WSL for students, deb for me, rpm and IaaS images for business
- Evangelism, show how good 2.0 is *on real, important problems*
- Tutorial series, in places where people are learning about code
  - YouTube, Twitch, Reddit, Blogs?
  - Installation all the way through to first GPGPU code
- Curricula examples for profs to grab-n-go
  - Auto-grading plugin(s) via LSP?
- *Automatically* co-schedule `forall`/`foreach` to CPU+GPU(s)
  - i.e. ***not*** an explicitly-partitioned `cobegin`
  - Ok to start with strictly unified memory within a single locale
- Intel GPU support eventually → 3 competitors is good for user pricing

Select artifacts

- OpenDwarfs: https://github.com/vtsynergy/OpenDwarfs
  - 13 computational idioms for OpenCL. Would love to see “Chapel-tastic” versions!
- CU2CL: https://github.com/vtsynergy/CU2CL
- MetaMorph: https://github.com/vtsynergy/MetaMorph
- MetaCL: https://github.com/vtsynergy/MetaMorph/tree/master/metamorph-generators/opencl
- Unpartitioned Chapel Jaccard
  - CHIUW’23 Talk: https://chapel-lang.org/CHIUW/2023/SathreSlides.pdf
    - Paper: https://chapel-lang.org/CHIUW/2023/Sathre.pdf
  - Code: https://github.com/vtsynergy/Chapel-Examples
  - Input data: https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html
- Other lab code: https://github.com/vtsynergy
  - And papers: https://synergy.cs.vt.edu/publications.php
  - Points of contact: {sath6220, feng} at cs dot vt dot edu

Code Example: GPU 3D vertex-centric Jaccard Similarity

$$JS(D,F) = \frac{D \cap F}{D \cup F}$$



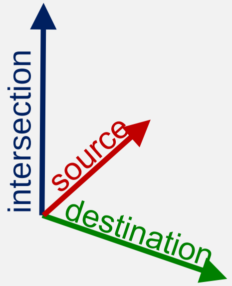
NVIDIA

We did run into some issues porting a 3D CUDA kernel

- GPU foralls are only 1D (for now)
  - Solution: Linearize loop range, then de-
- `for-by` loops do *NOT* GPU-ize (yet)
  - Problem: non-constant by clause can `halt`
  - Solution: Replace with `while-count`
- Accumulate via `atomicAdd` did *NOT* GPU-ize before Chapel 1.31
  - Solution: Call CUDA's via `extern C`
  - Now has a `gpuAtomic<Func>` API
- Could *incrementally validate* as kernels were transparently mapped to CPU

See my talk at CHIUW'23 for more on this

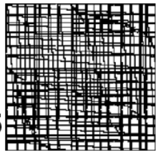
```
var intersect : [0..<numEdges] real(32);
forall id in srcIters*destIters*isectIters {
  var nd_id : 3*int = get_ND_ID(id);
  var zCount = nd_id(2);
  while (zCount < srcIters) {
    var yCount = nd_id(1);
    while (yCount < destIters) {
      var xCount = nd_id(0);
      while (xCount < isectIters) {
        ... // bin-search
        gpuAtomicAdd(intersect[writeAddr], 1.0);
      xCount += nd_id.global_dim(0); }
    yCount += nd_id.global_dim(1); }
  zCount += nd_id.global_dim(2); }
}
```



Edge- and Vertex-centric Jaccard: Data

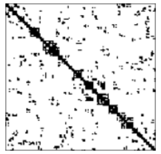
Sparsest to densest

Sparsest/Largest

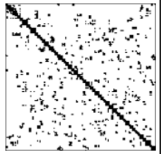
Kmer_A2a
Protein *k*-mers
V: 171M
E: 361M
Degree
Avg: 2.11
Range: 39
Std. Dev.: 0.56
Gini Index: 0.055


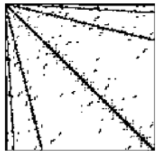
Europe_osm
European roads
V: 50.9M
E: 108M
Degree
Avg: 2.12
Range: 12
Std. Dev.: 0.48
Gini Index: 0.085

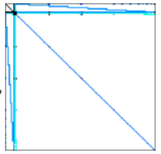

Road_usa
US roads
V: 23.9M
E: 57.7M
Degree
Avg: 2.41
Range: 8
Std. Dev.: 0.93
Gini Index: 0.211


Road-roadNet-CA
California roads
V: 1.96M
E: 5.52M
Degree
Avg: 2.82
Range: 11
Std. Dev.: 0.99
Gini Index: 0.185


Smallest

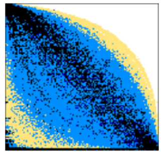
Road-roadNet-PA
Pennsylvania roads
V: 1.09M
E: 3.08M
Degree
Avg: 2.83
Range: 8
Std. Dev.: 1.02
Gini Index: 0.188


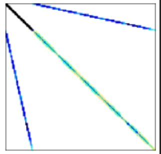
Delaunay_n24
Random Triangulations
V: 16.8M
E: 101M
Degree
Avg: 6.00
Range: 23
Std. Dev.: 1.34
Gini Index: 0.122


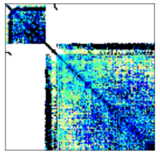
circuit5M
Large circuit
V: 5.56M
E: 54.0M
Degree
Avg: 9.71
Range: 1.29M
Std. Dev.: 1357
Gini Index: 0.577


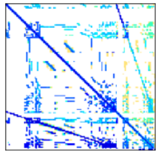
Soc-LiveJournal1
Social network
V: 4.85M
E: 85.7M
Degree
Avg: 17.7
Range: 20.3K
Std. Dev.: 52.0
Gini Index: 0.711

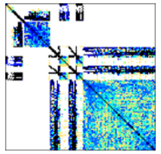

Wikipedia-20070206
Web page links
V: 3.57M
E: 84.8M
Degree
Avg: 23.8
Range: 188K
Std. Dev.: 255
Gini Index: 0.759


GL7d19
Voronoi differentials
V: 1.96M
E: 74.6M
Degree
Avg: 38.2
Range: 134
Std. Dev.: 6.73
Gini Index: 0.088


dielFilterV2real
Dielectric resonator
V: 1.16M
E: 47.4M
Degree
Avg: 40.9
Range: 104
Std. Dev.: 16.1
Gini Index: 0.201


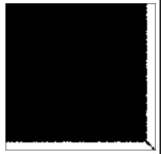
Sc-ldoor
Large door
V: 952K
E: 41.5M
Degree
Avg: 43.6
Range: 76
Std. Dev.: 14.8
Gini Index: 0.183


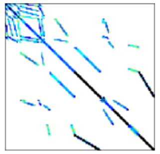
Stokes
VLSI process sim.
V: 11.4M
E: 516M
Degree
Avg: 45.1
Range: 1728
Std. Dev.: 61.8
Gini Index: 0.392


Sc-msdoor
Medium Door
V: 416K
E: 18.8M
Degree
Avg: 45.1
Range: 76
Std. Dev.: 13.7
Gini Index: 0.166


Ca-coauthors-dblp
Coauthorship
V: 540K
E: 30.5M
Degree
Avg: 56.4
Range: 3298
Std. Dev.: 66.2
Gini Index: 0.544


Soc-orkut
Social network
V: 3.00M
E: 213M
Degree
Avg: 71.0
Range: 27.5K
Std. Dev.: 140
Gini Index: 0.558


Hollywood-2009
Costarring Actors
V: 1.14M
E: 113M
Degree
Avg: 98.9
Range: 11.5K
Std. Dev.: 272
Gini Index: 0.750


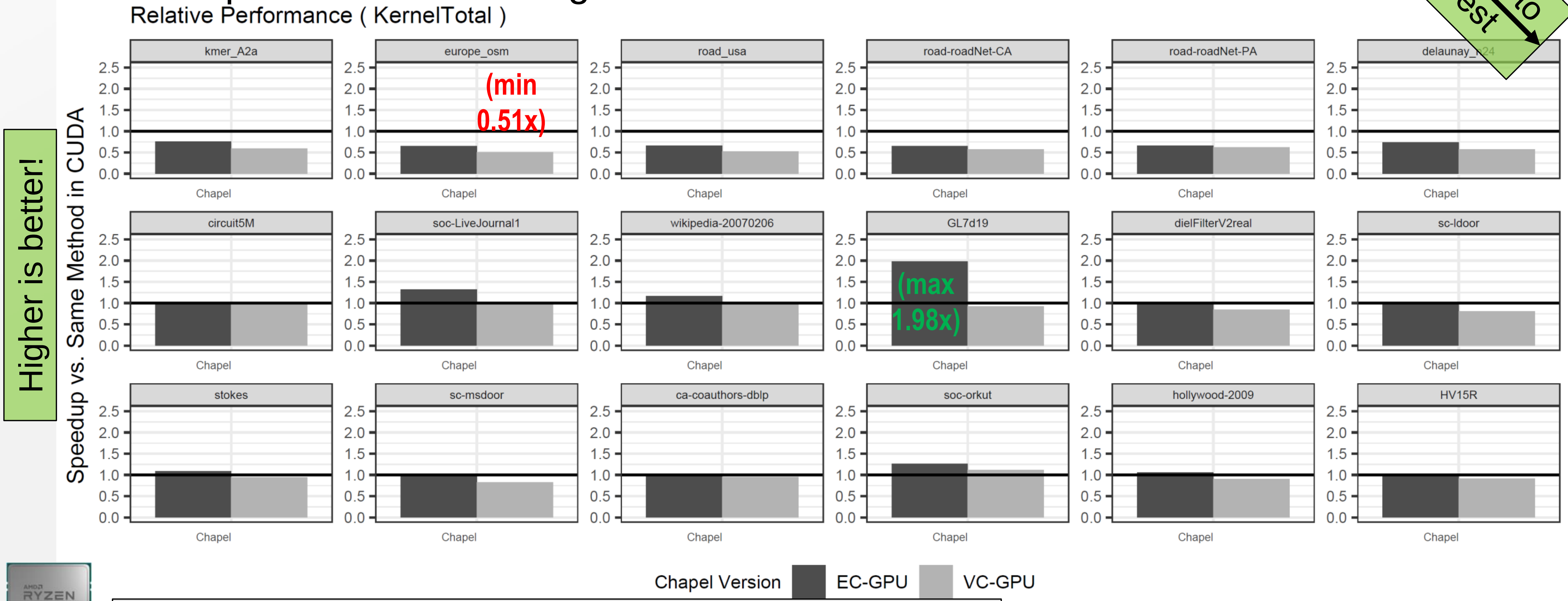
HV15R
CFD of engine fan
V: 2.02M
E: 325M
Degree
Avg: 161
Range: 491
Std. Dev.: 47.8
Gini Index: 0.155


Densest

Preprocessed CSR binary files: https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html

Edge- and Vertex-centric Jaccard:

Kernel performance vs. original CUDA on RTX 3090



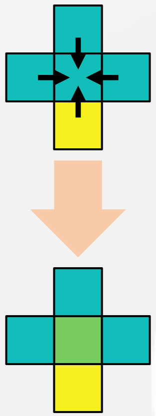
CPU: AMD Threadripper 3960X **CUDA**: 11.6 / driver 510.108.03
GPU: Nvidia RTX 3090 **Chapel**: pre-1.31 (d7664c9d81)

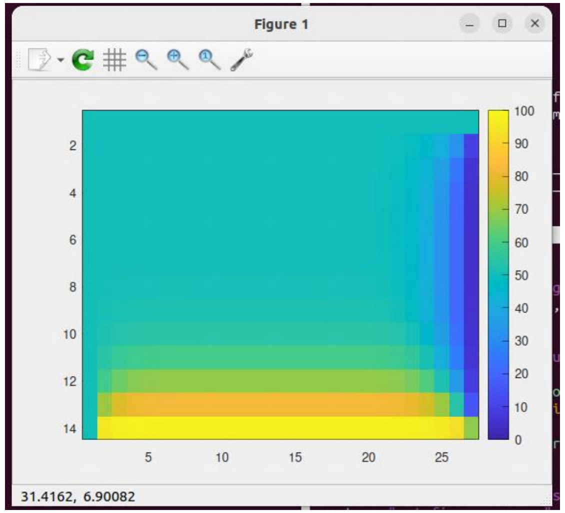
See my talk at CHIUW'23 for more on this

{AI, Data, Domain} Scientists need visualization!

- Python/Jupyter drive an ***interactive*** code → viz → analysis loop
  - Need to support visualization *within the human thought loop* to attract these workflows
  - Chapel doesn't have a plotting module, but could C interop fill the role? "*How hard can it be*?"

- Experiment with Chapel → C → C++ interop via `matplot++` GitHub
  - 1024x512, 5-point stencil Jacobi heat diffusion toy, after 10k iters →
    - ✓ 62 lines of Chapel application with 8 calls to library
      - Grid subsampled to 26x13 ***in a single line*** for slow `gnuplot` backend
    - ✘ ... but 32 lines of manual Chapel→C binding
      - not great, but could be automated via `c2chapel`
    - ✘ ... and 97 lines of manual C→C++ binding





Starting condition: 50˚
Boundary conditions:
S=100, E=0, N&W=50
1-thick ghost padding

- There is a gold mine of wrappable AI and Data Science libraries
  - ... but would need both Chapel & C++ experience, rough otherwise