OAK RIDGE National Laboratory

December 5, 2025

Toward Productive HPC Programming Systems for the Post-Exascale Era

Keita Teranishi

Group Leader, Programming Systems, Computer Science and Mathematics Division

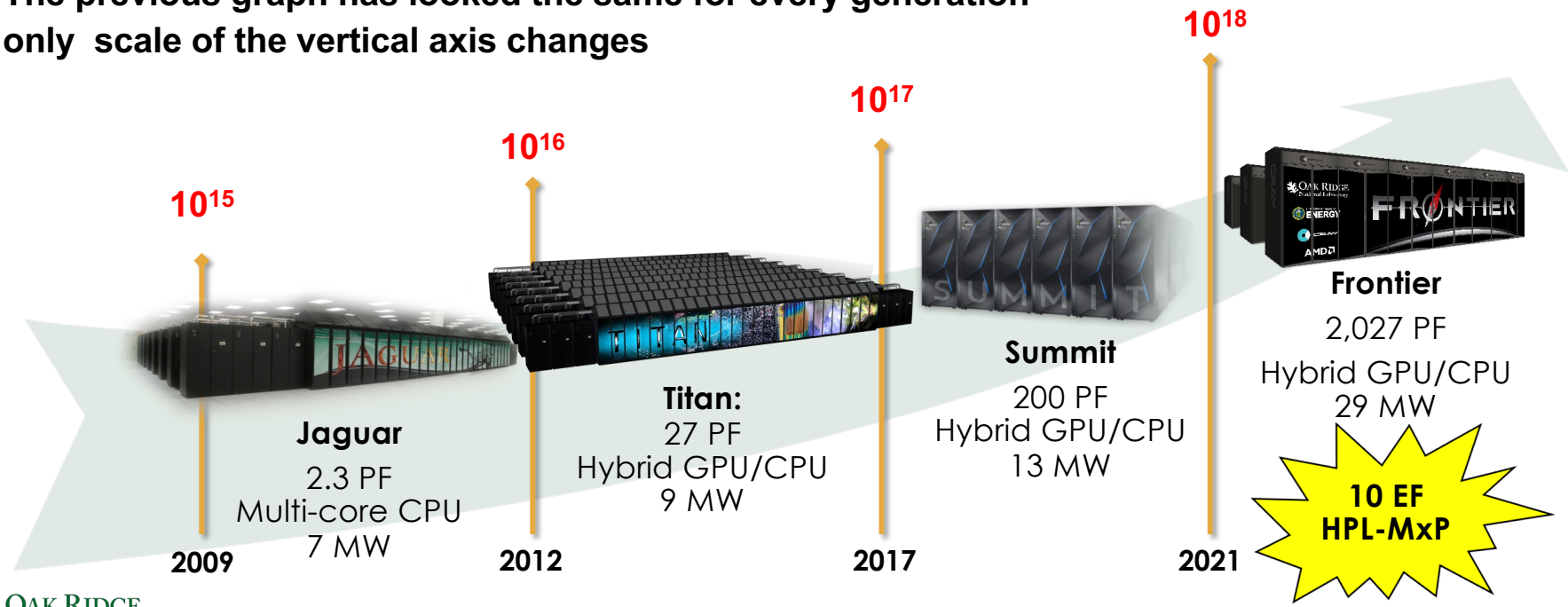
U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

Oak Ridge Leadership Computing Facility 2009-2021 from Petascale to Exascale

ORNL successfully delivered a string of #1 Leadership Computers

The previous graph has looked the same for every generation only scale of the vertical axis changes



Credit: Al Geist (ORNL)

Post-Exascale Era - AI Supercomputing Era

- Announcement by Oak Ridge National Lab and Argonne National Lab

ORNL, AMD and HPE to deliver DOE's newest AI supercomputers: Discovery and Lux

The next evolution in leadership-class artificial intelligence supercomputing systems

October 27, 2025
Last updated: October 27, 2025

RESEARCHERS


Georgia Tourassi


Matt T Sieger


Bronson Messer

Organizations

Computing and Computational Sciences Directorate

NVIDIA and Oracle to Build US Department of Energy's Largest AI Supercomputer for Scientific Discovery

Bold US Investment of 100,000 NVIDIA Blackwell GPUs Kickstarts Era of Agentic AI-Powered Science at Argonne National Laboratory for Public Researchers

October 28, 2025

OAK RIDGE National Laboratory

Genesis Mission - AI-for-science program

- Executive Order, November 24
- Manhattan/Apollo–style effort to use AI and federal scientific data to “unleash a new age of AI-accelerated innovation and discovery” and bolster U.S. technological dominance.
- Puts DOE in charge and builds a unified AI platform
- Builds in security and access control
- Forces government-wide alignment and partnerships
- Imposes aggressive implementation deadlines

Today's Talk

- Post-ECP Software Ecosystem
- Emerging Programming Language
  - Julia
  - Mojo

Stewardship of HPC Programming Software Ecosystem

Why do we care about programming environment?

From *Hidden Figures*, 2016.

"FORTRAN(IV) is a new and exciting language used by programmers to **communicate with computers**. It is exciting as **it is the wave of the future.**" --Octavia Spencer as Dorothy Vaughan, Langley Research Center, NASA in 1961.

(Vaughan was using IBM 7090 Mainframe Computer)

Today, **we continue to use programming systems as a critical tool to communicate with computers and shape the future.**

Courtesy: Damian Rouson at LBL and Kengo Nakajima at RIKEN/U of Tokyo who found the line in the movie.

Scientific Computing Software Development During Exascale Computing Project (ECP) and Post-ECP Era

US DOE Exascale Computing Project (2016-2023)

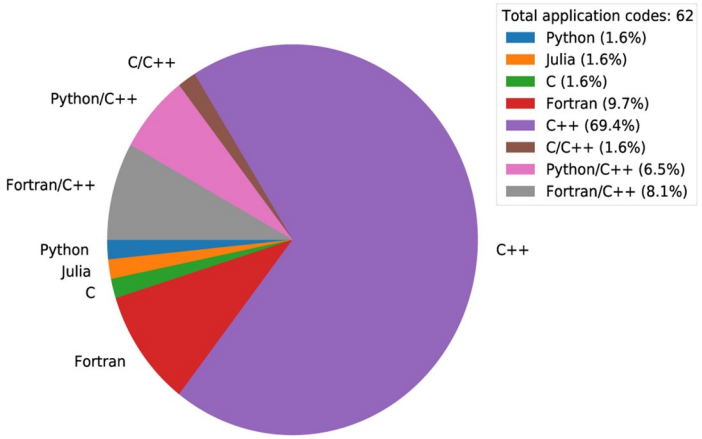
- **Software Technology Project**
  - Development of 70 software products
  - Annual budget of **$70M**
- **Programming Systems**
  - Developed over 10 programming system products
- **Supercomputing Systems**
  - All DOE supercomputing systems employ accelerator technology

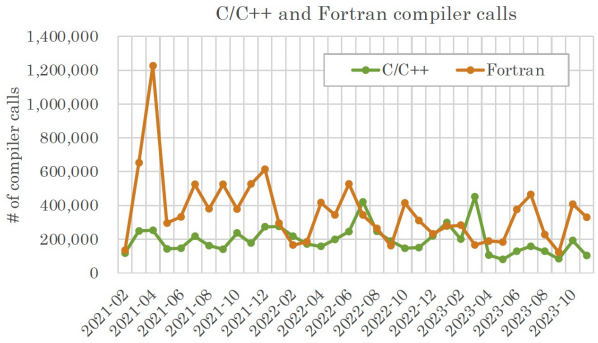
Post-ECP Initiative

- **End of 2022: Software Sustainability Proposal Call**
  - Issued by the US DOE Advanced Scientific Computing Research (ASCR)
- **April 2023: Phase 1 – Incubation**
  - **6 projects initiated**
- **Fall 2023: the Next Generation Scientific Software Technology (NGSST)**
  - **Four** existing projects and **one** new project were recommended for funding
- **January 2024: Phase 2 Commencement**
  - Annual funding of **$11.5M** allocated through 2028

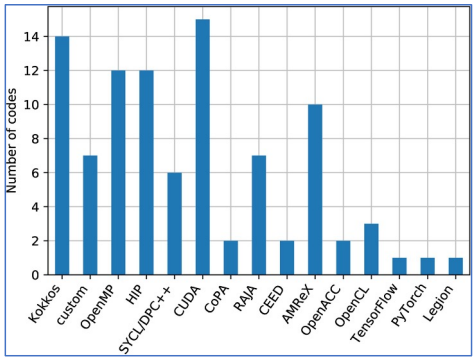
Observations during ECP

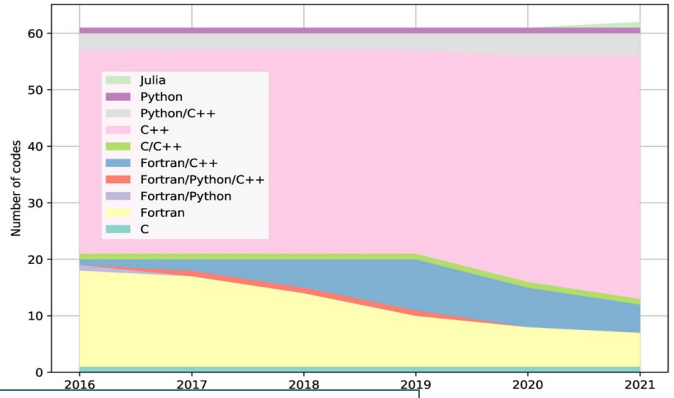
- Adapting to Heterogeneous Computing Norms
  - NVIDIA, AMD and Intel GPUs
- Enhancing Productivity Through **Performance Portability**
  - Same source code across different platforms
- Evolving Programming Systems
  - **C++ has become dominant language for ECP apps**
  - **High productivity Languages, AI/ML**
- Legacy Applications
  - Many Fortran applications are still actively used in other DOE programs.





Courtesy: Kento Sato, RIKEN, Japan





Courtesy: Evans TM, Siegel A, Draeger EW, et al. *A survey of software implementations used by application codes in the Exascale Computing Project.* IJHPCA. 2022;36(1):5-12

Change in Performance Portability Landscape

TOP 500 Supercomputer

10 highest ranked 11/2007

Intel Xeon 4C
3 Systems

AMD Opteron 2C
3 Systems

IBM PowerPC
4 Systems – 2C/4C

Change in Performance Portability Landscape

TOP 500 Supercomputer

10 highest ranked 11/2023

CPUs:

Threading
Building
Blocks

std::thread

Accelerators

Observations during ECP

HPC Vendor Shift

- Transitioning from traditional HPC to AI/ML-focused systems.

Cloud HPC Growth

- Cloud-based solutions are overtaking on-premises HPC.

DOE's Custom Solutions

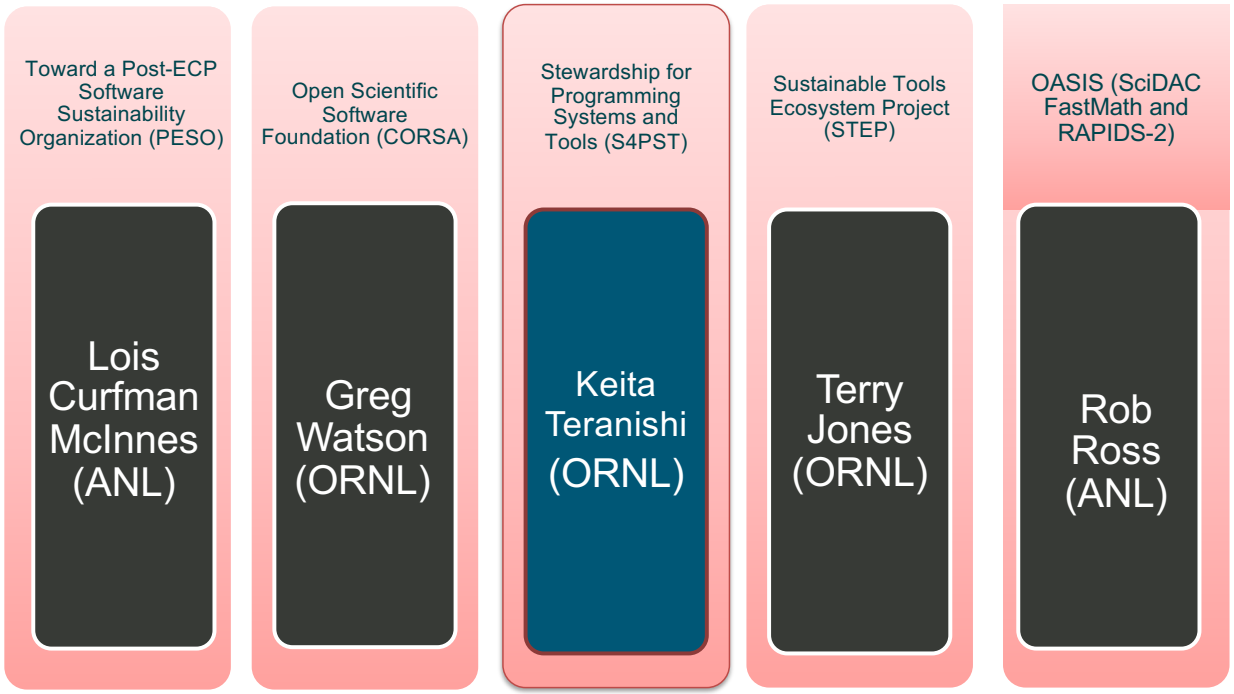
- Features compilers, runtime, APIs, and testing suites with fast support.
- Committed to standardizing C++, Fortran, OpenMP/ACC, MPI to meet DOE's specific needs.
- Aims to establish a unique position within the broader open-source software community.

Strategic Solution

- Embrace a community-wide, proactive approach to new technology integration.

5 Software Stewardship Organizations

NGSST is the next step after the successful DOE Exascale Computing Project (2016-2023) as we evolved into “Heterogeneous and AI dominated landscape”



Michael Heroux (retired, SNL) served the former PI of PESO.

What is S4PST?

- Objective: Enhance Programming Systems for next-generation high-performance computing (HPC) systems and ensure their seamless integration with emerging AI technologies for scientific advancement.
- Scope: Focus on Performance Portable Parallel Programming Frameworks, Compilers, Distributed Computing Framework, and High Productivity Languages.

- Challenges:
  - **Technical:** Requires unique skill sets, including comprehensive knowledge of full-stack technologies, system architecture, and application needs.
  - **Portfolio:** Commitment to serving existing users and computing systems while embracing new technologies, meeting evolving application demands, and fostering the next generation of HPC experts.
  - **People: Dedicated engagement is essential for the development of individual products, support of users, and mentoring of emerging talents (future HPC experts and leaders).**

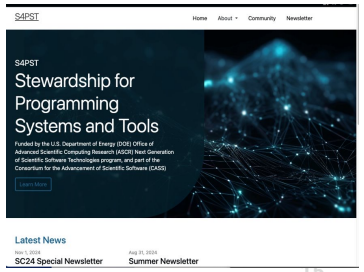
S4PST: Our portfolio for DOE's Scientific Mission

Product Stewardship

- OpenMP/OpenACC
- LLVM
- Fortran
- Kokkos
- MPICH
- Open MPI
- Legion
- GASNet-EX/UPC++
- HIP
- SYCL

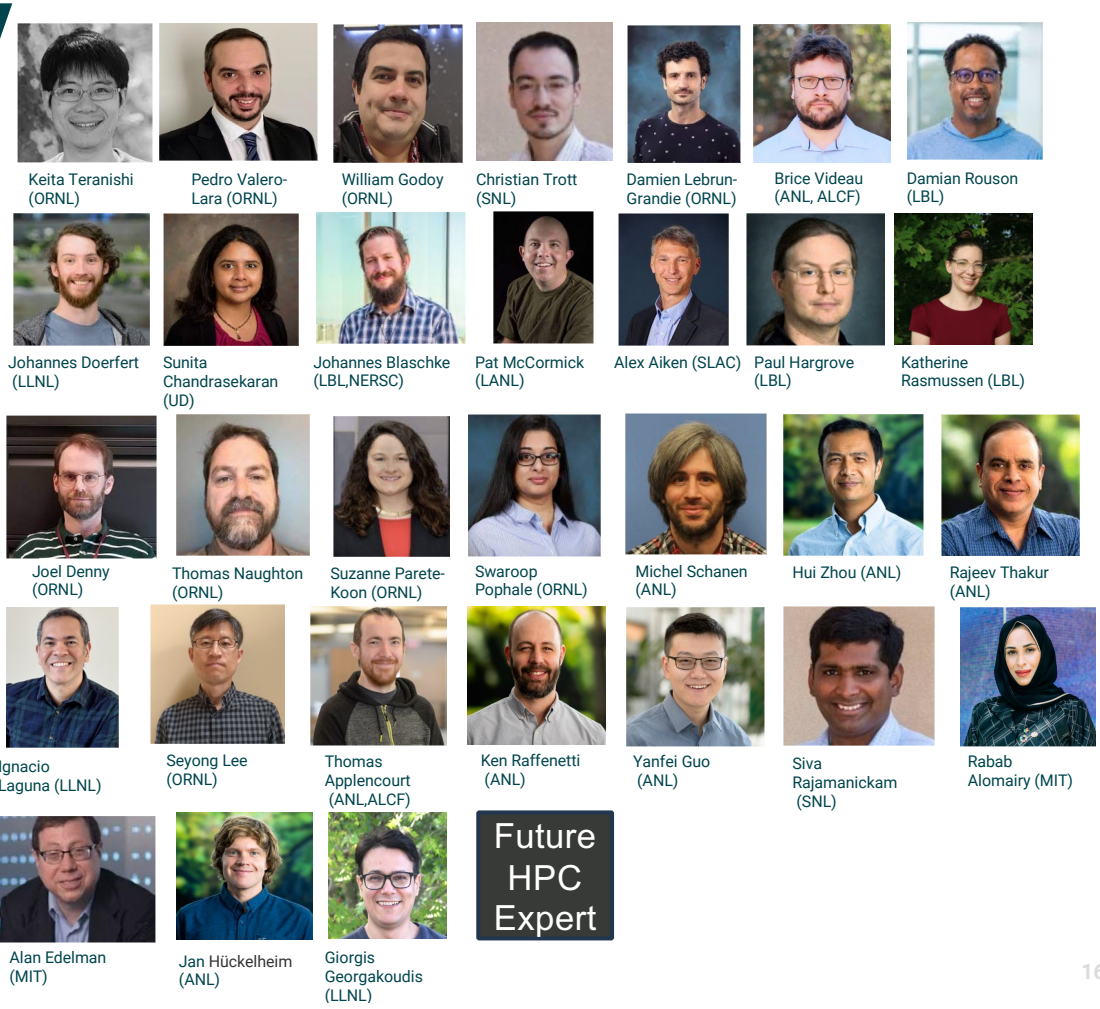
Emerging Technologies

- Automatic Differentiation (AD)
- HPC+AI ecosystems: Julia, Mojo, Python
- Large language models for HPC
- Kokkos ecosystems adapting AI-hardware/special arithmetic units



S4PST Community

- PI: Keita Teranishi (ORNL)
- CoPI: William Godoy and Pedro Valero Lara
- 7 National Laboratories:
  - Oak Ridge National Laboratory
  - Argonne National Laboratory
  - Lawrence Livermore National Laboratory
  - Lawrence Berkeley National Laboratory
  - Sandia National Laboratories
  - Los Alamos National Laboratory
  - SLAC National Accelerator Laboratory
- University Partners:
  - University of Delaware
  - Massachusetts Institute of Technology
- Collaborations:
  - Louisiana State University
  - Pacific Northwest National Laboratory
  - Carnegie Mellon University
  - University of Tennessee, Knoxville
  - Stanford University
  - Other 6 NSSGT projects



HPSF Conference 2025

Partnership: NNSA – ASC, Industry

Achievement

- Coorganized the inaugural HPSF Conference
- HPSF is the High Performance Software Foundation a part of the Linux Foundation
- HPSF partners include DOE labs, universities, and industry (hpsf.io/members)
- HPSF is the home for DOE led Open Source projects, including Kokkos and Spack enabling multi-institutional collaboration through open governance

Significance and Impact

- The conference brought together over 200 developers and users of the HPSF projects
- The event enables users of projects such as Kokkos to learn from each other, and provide feedback to developers to inform future directions of the project
- HPSF Conference is engineering focused: how do we manage open source projects, what technical challenges are users encountering, how do we improve robustness and sustain Open Source Software for scientific and engineering HPC

The HPSF Conference 2025 facilitated the coming together of the HPSF community to exchange experiences and deepen collaborations. Feedback on the conference was extremely positive with planning for the next event in 2026 now underway.

Member Products

- Kokkos, Chapel, OpenMPI (planning)

S4PST – Highlights

Software Release:

- Kokkos 4.7
- OpenMPI 5.0.2, MPICH 4.3
- OpenMP 5.2 and 6.0 Verification Suites
- OpenACC 3.3 Verification Suites
- ChipStar v1.1 (HIP on Aurora)
- Julia 1.1
- JACC (Julia for Accelerators)

Technical Accomplishments:

- **Kokkos-3** won IEEE TPDC Best Paper Award
- **Legion** runs on 8,000 nodes of Frontier
- ComPile **LLVM-IR LLM** released
- 5+ IJHPCA Journal paper submissions

Collaborations

- OpenACC Specification Committee
- OpenMP Specification Committee
- Enzyme team and several universities under NSF project
- SciDAC Next Generation Power Grid Analysis

Outreach:

- S4PST Presentation in Japan (HPC-AI Council)
- Kokkos User Group Meeting at SNL, NM.
- Kokkos Developers Meeting at ORNL
- Kokkos Presentation and Tutorial in Japan
- LBL hosted Fortran Standards Committee Meeting in Berkeley
- ANL hosted IWOCL24 in Chicago
- MPICH and OpenMPI teams collaborate for **Abstract Binary Interface** standardization
- SIAM PP24: OpenMP, LLVM, Kokkos, Fortran, Julia
- SC24/SC25 BoFs, Tutorials,
- SIAM CSE25: Julia, CASS, etc.

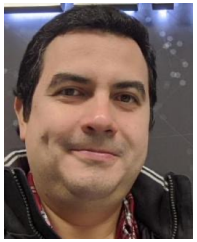
International Collaboration:

- DOE-MEXT (Japan) on Fortran and Kokkos
- Kokkos (DOE team) - CEA Collaboration
- ADAC: The Accelerated Data Analytics and Computing Institute

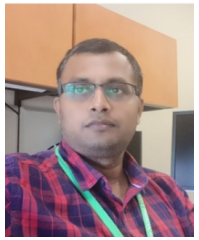
Emerging High Productivity Languages

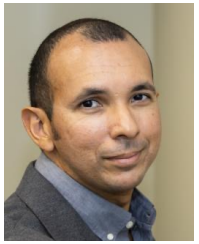
Thanks!

- Tatiana Melnichenko
- William Godoy
- Pedro Valero-Lara
- Philip Fackler
- Steven Hahn
- Narasinga Rao Miniskar
- Het Mankad
- Rafael Ferreira da Silva
- Jeff Vetter





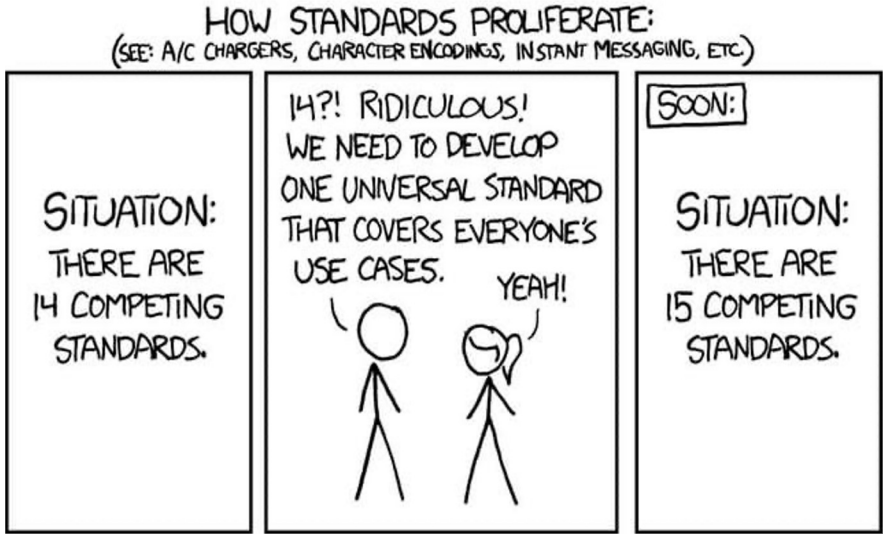



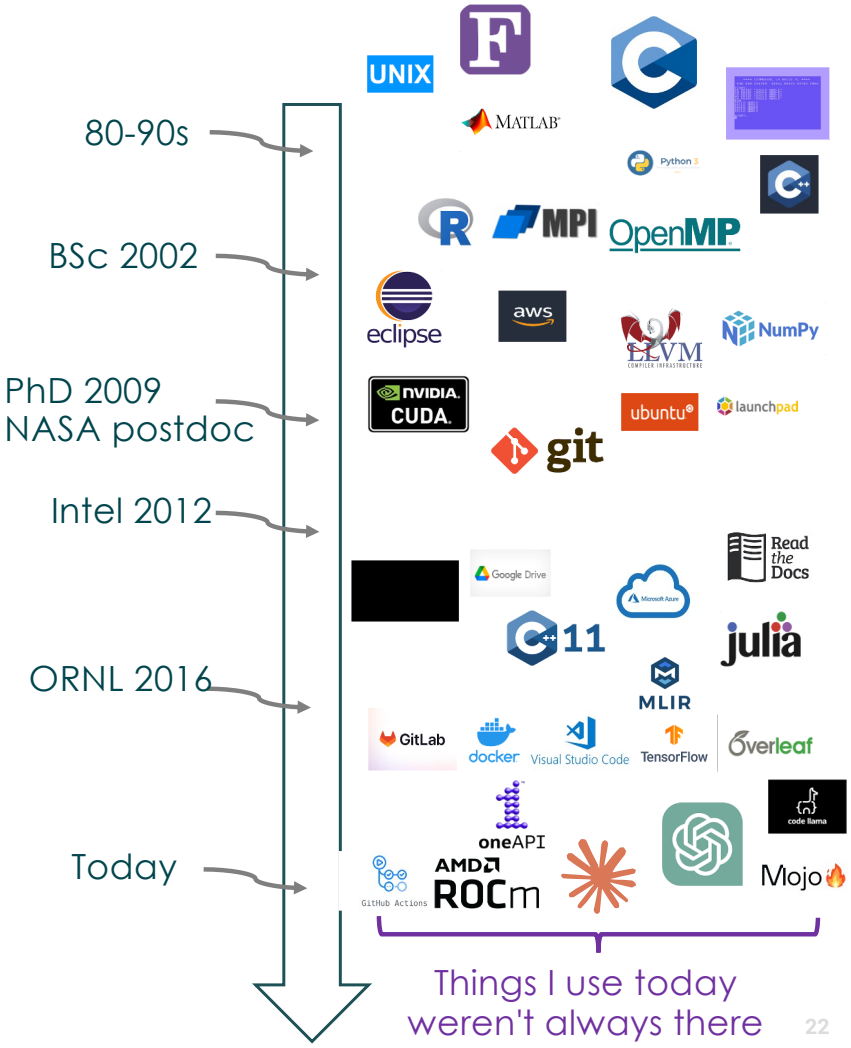



Research directions toward solutions

- Researchers are actively developing languages and tools to bridge these gaps

Yet another language?



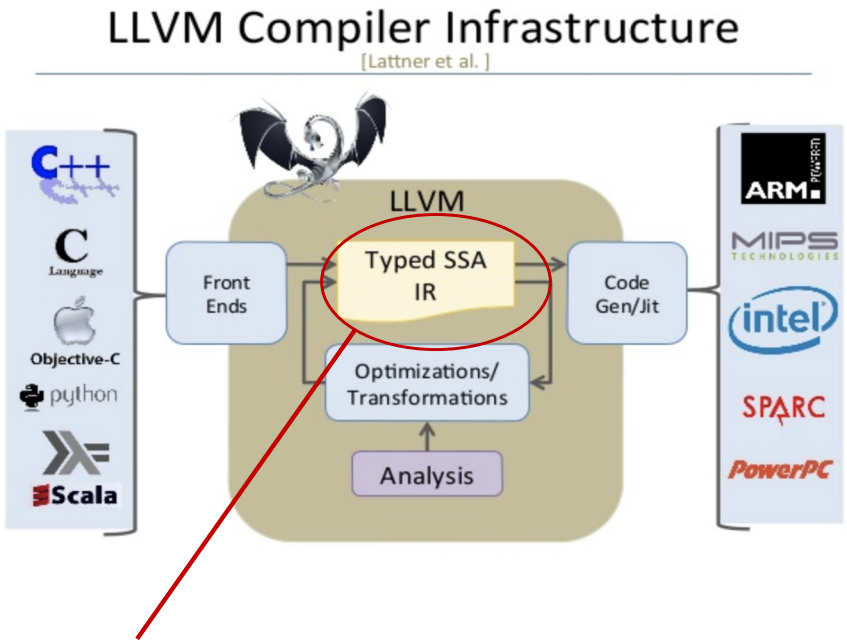


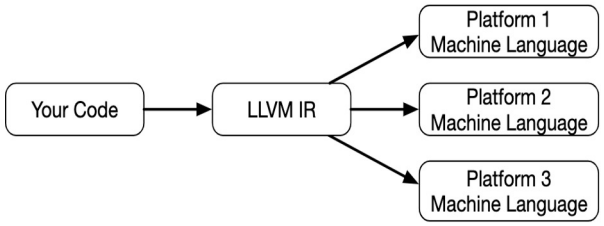
LLVM: a game changer

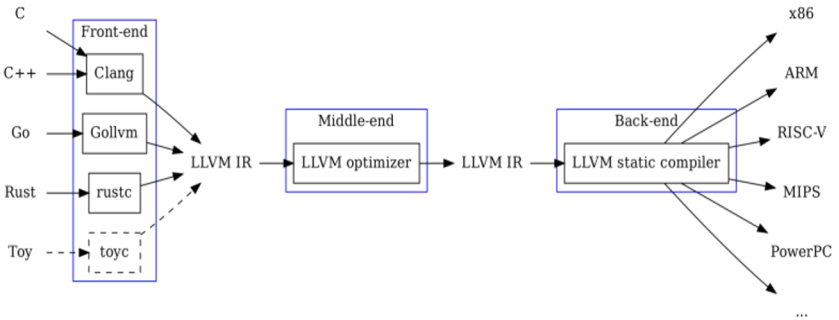
https://llvm.org/

http://www.aosabook.org/en/llvm.html

C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75-86, https://doi.org/10.1109/CGO.2004.1281665 .







LLVM Typed Static Single Assignment (SSA) Intermediate Representation (IR) aka LLVM-IR: https://patshaughnessy.net/2022/2/19/llvm-ir-the-esperanto-of-computer-languages

%57 = call %"Array(Int32)"* @"*Array(Int32)@Array(T)::unsafe_build:Array(Int32)"(i32 610, i32 2), !dbg !89

Julia: “scientific” (Fortran-like) access to LLVM and paid by industry

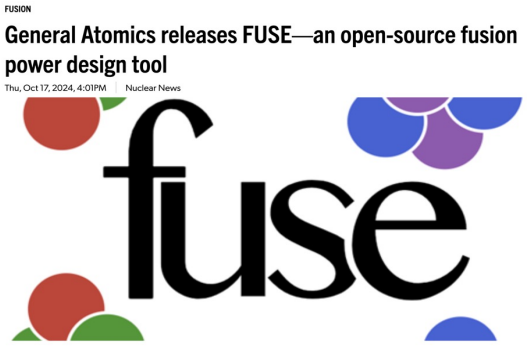
```
julia> function add(x,y)
         return x+y
       end

julia> @code_llvm add(2,3)
;  @ REPL[1]:1 within `add`
define i64 @julia_add_132(i64 signext %0, i64 signext %1) #0 {
top:
;  @ REPL[1]:2 within `add`
; ┌ @ int.jl:87 within `+`
   %2 = add i64 %1, %0
; └
  ret i64 %2
}
```

https://godbolt.org/

17 July, 2023

Newsletter July 2023 - JuliaHub Receives $13 Million Strategic Investment from Boeing-Backed AEI HorizonX

Written by JuliaHub



CUDA Zone

CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python, Julia and MATLAB and express parallelism through extensions in the form of a few basic keywords.

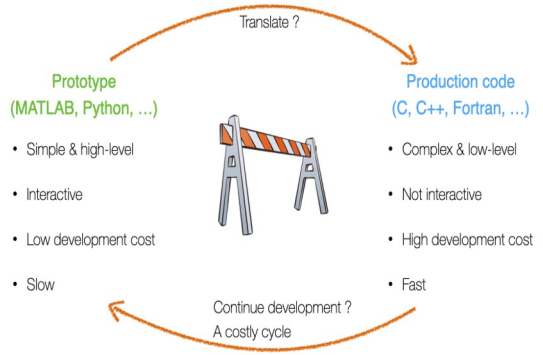
The CUDA Toolkit from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.

Download Now

Julia's value proposition for science

- Designed for “scientific computing” (Fortran-like) and “data science” (Python-like) with **performant kernel code via LLVM compilation**
- Lightweight **interoperability with existing Fortran and C libraries**
- Julia is a **unifying workflow language** with a **coordinated ecosystem**

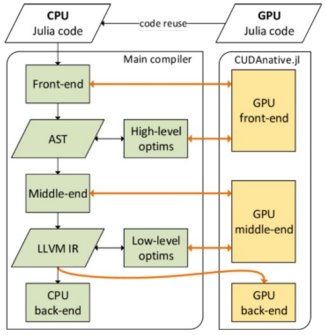
“Julia **does not** replace Python, but the costly workflow process around **Fortran+Python+X**, **C+X**, **Python+X** or **Fortran+X** (e.g. GPUs, simulation + data analysis)”
where X = { conda, pip, pybind11, Cython, Python, C, Fortran, C++, OpenMP, OpenACC, CUDA, HIP, CMake, numpy, scipy, Matplotlib, Jupyter, **…}**



https://pde-on-gpu.vaw.ethz.ch/lecture7

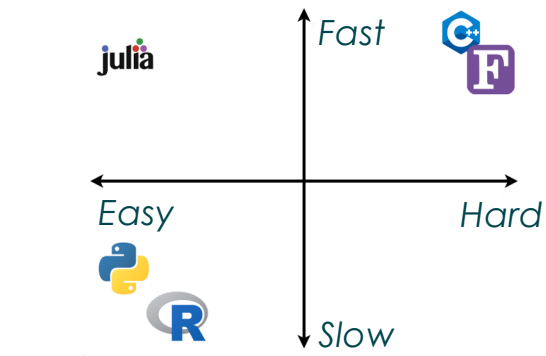
OAK RIDGE National Laboratory

LLVM



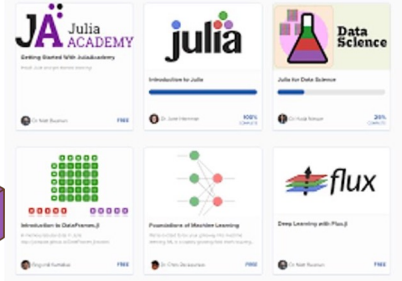
https://developer.nvidia.com/blog/gpu-computing-julia-programming-language/

Pkg.jl



https://juliadatascience.io/

Rich data science ecosystem

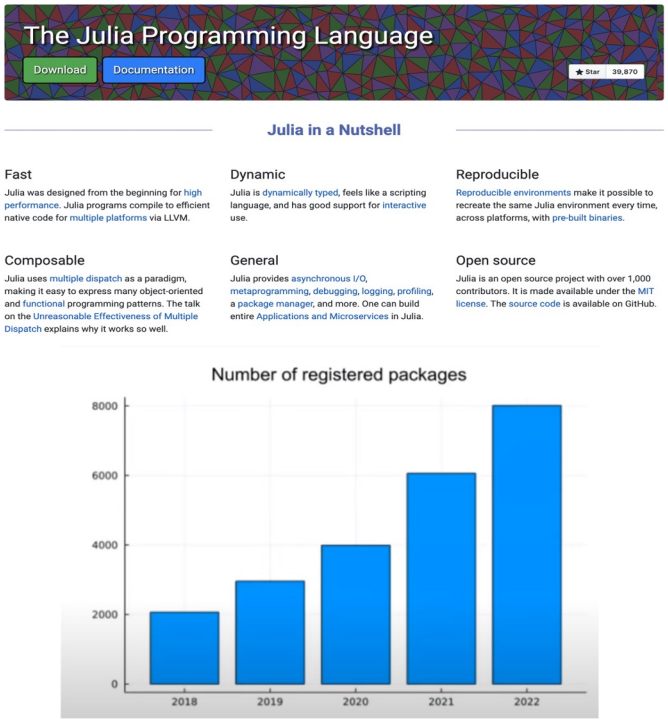


https://quantumzeitgeist.com/learning-the-julia-programming-language-for-free/

Julia Brief Walkthrough

- History: started at MIT in the early 2010s (predates Python Numba) https://julialang.org/blog/2022/02/10years/
- **JuliaHub (formerly Julia Computing)** and **MIT** are major contributors: https://info.juliahub.com/case-studies
- First stable release v1.0 in 2018, v1.11 as of 2025 https://julialang.org/
- Open-source GitHub-hosted packages and ecosystem with MIT permissive license: https://github.com/JuliaLang/julia
- Community: annual JuliaCon summer conference: https://juliacon.org/2025/

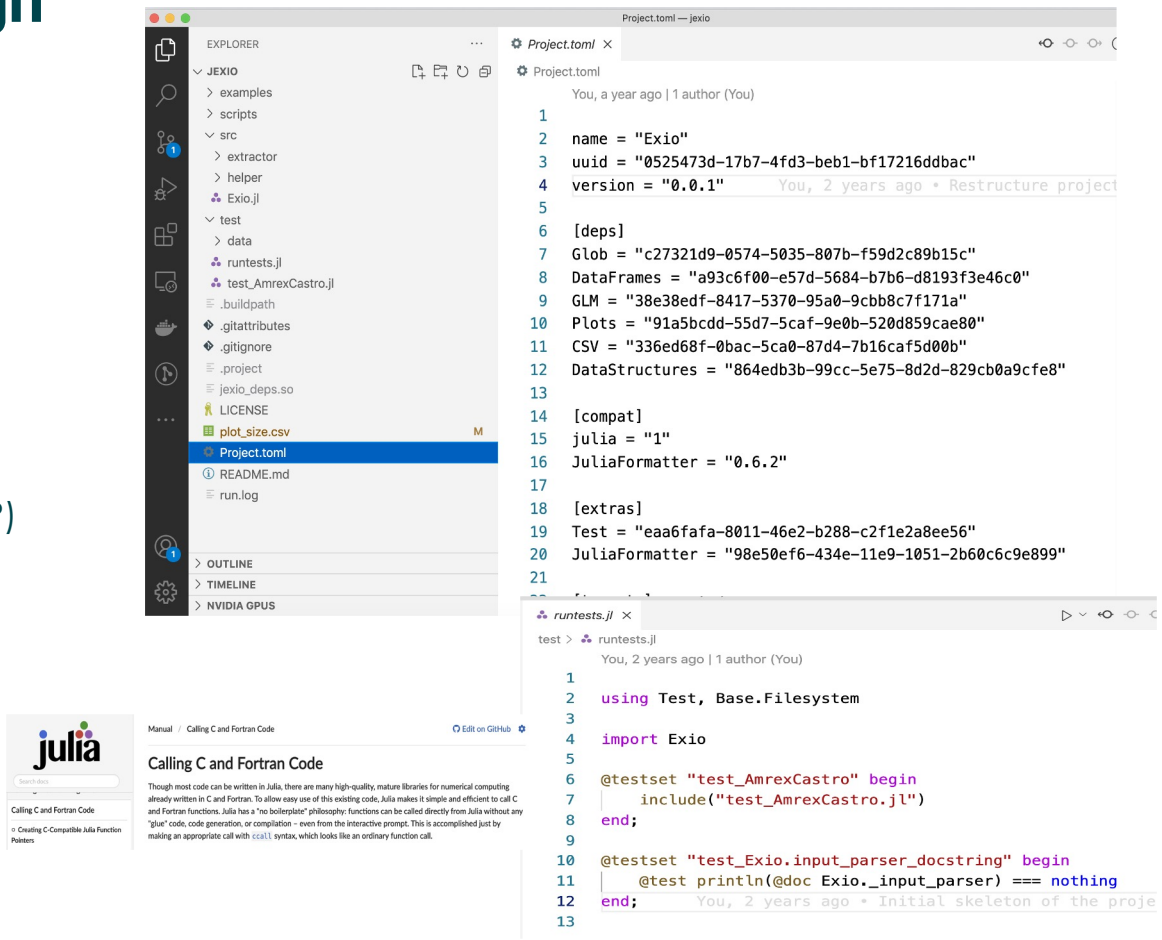
julia Pluto.jl jupyter



95% of Julia packages in the registry had some form of CI (youtube.com/watch?v=9YWwiFbaRx8)

Julia Brief Walkthrough

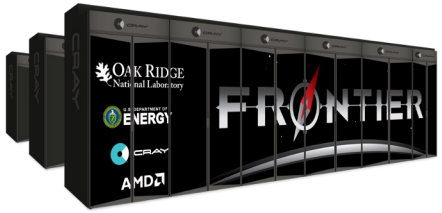
- Reproducibility is in the core of the language:
  - Interactive: Jupyter, Pluto.jl
  - Packaging Pkg.jl
  - Environment Project.toml
  - Testing Test.jl
- Just-in-time or Ahead-of-time compilation with PackageCompiler.jl (juliac is WIP)
- Powerful metaprogramming for code instrumentation: @profile, @time, @testset, @test, @code_llvm, @code_native, @inbounds,
- Interoperability is key: @ccall, @cxx, PyCall, CxxWrap.jl

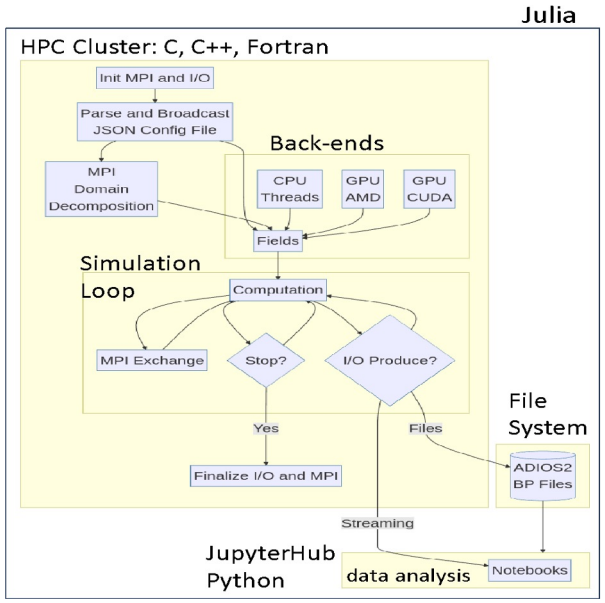


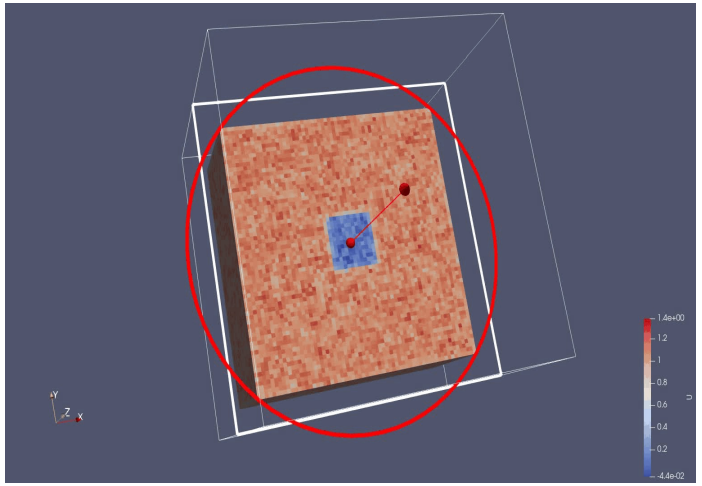
Gray-Scott app: https://github.com/JuliaORNL/GrayScott.jl

Simple 3D 2-variable diffusion-reaction solver

- CPU Threads, CUDA.jl and **AMDGPU.jl** backends using multiple dispatch
- Parallel I/O ADIOS2, can be visualized with ParaView
- MPI.jl for communication
- Configuration and job scripts for Frontier, Crusher and Summit under ./scripts/
- Data analysis on JupyterHub







$$\frac{\partial U}{\partial t} = D_U \nabla^2 U - UV^2 + F(1-U) + nr \qquad (1a)$$

$$\frac{\partial V}{\partial t} = D_V \nabla^2 V + UV^2 + -(F+k)V \qquad (1b)$$

Research question: Can I write an entire HPC "hard-coupled" workflow in Julia?

Best paper at SC23 WORKS

RESEARCH-ARTICLE

Julia as a unifying end-to-end workflow language on the Frontier exascale system

Authors: William F. Godoy, Pedro Valero-Lara, Caira Anderson, Katrina W. Lee, Ana Gainaru, Rafael Ferreira Da Silva, Jeffrey S. Vetter Authors Info & Claims

SC-W '23: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis • November 2023 • Pages 1989–1999 • https://doi.org/10.1145/3624062.3624278

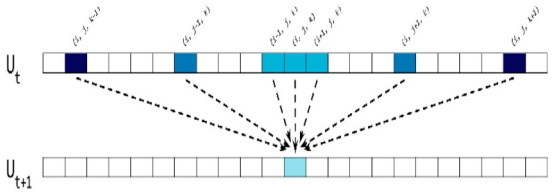
https://doi.org/10.1145/3624062.3624278

https://www.nextplatform.com/2023/09/26/julia-still-not-grown-up-enough-to-ride-exascale-train/

OAK RIDGE National Laboratory

Frontier on-node scalability using AMDGPU.jl

7-point stencil



$$fetch\_size_{effective} = \left[L^3 - 8 - 12\,(L-2)\right] \cdot sizeof(T) \tag{4a}$$

$$write\_size_{effective} = (L-2)^3 \cdot sizeof(T) \tag{4b}$$

$$bandwidth_{effective} = \frac{(fetch\_size + write\_size)_{effective}}{kernel\_time} \tag{5a}$$

$$bandwidth_{total} = \frac{(FETCH\_SIZE + WRITE\_SIZE)_{rocprof}}{kernel\_time} \tag{5b}$$

Table 2: Average bandwidth comparison of different stencil implementations on a single GPU.

Kernel	Bandwidth (GB/s)	
	Effective	Total
Julia GrayScott.jl		
- 2-variable (application)	312	570
- 1-variable no random	312	625
HIP single variable	599	1,163
Theoretical peak MI250x	1,600	

Table 3: rocprof outputs for HIP 1-variable and Julia Gray-Scott.jl implementations

kernel	HIP	GrayScott.jl	
metric	1-var	1-variable no random	2-variable (application)
wgr	256	512	512
lds	0	29,184	29,184
scr	0	8,192	8,192
FETCH_SIZE (GB)	25.08	25.40	50.80
WRITE_SIZE (GB)	8.35	8.38	16.78
TCC_HIT (M)	9.14	10.80	24.60
TCC_MISS (M)	8.36	8.69	17.19
Avg Duration (ms)	28.74	54.03	111.07

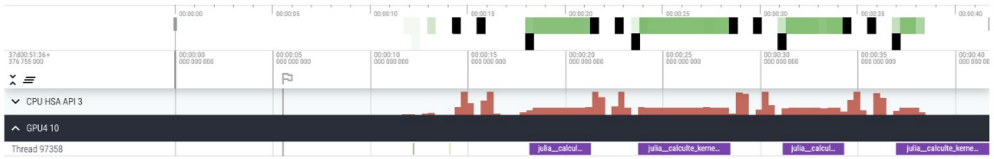


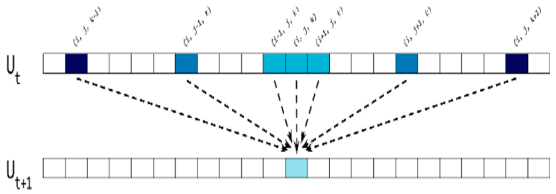
Figure 5: Gray-Scott simulation trace obtained with rocprof showing computational load on GPU and memory transfer to CPU for communication.

Listing 2: Julia AMDGPU.jl Gray-Scott kernel

```
using AMDGPU
using Distributions

function _laplacian(i, j, k, var)
  l =   var[i - 1, j, k] + var[i + 1, j, k]
      + var[i, j - 1, k] + var[i, j + 1, k]
      + var[i, j, k - 1] + var[i, j, k + 1]
      - 6.0 * var[i, j, k]
  return l / 6.0
end

function _kernel_amdgpu!(u, v, u_temp, v_temp,
                         sizes, Du, Dv, F, K,
                         noise, dt)

  k = (workgroupIdx().x - 1) * workgroupDim().x
      + workitemIdx().x
  j = (workgroupIdx().y - 1) * workgroupDim().y
      + workitemIdx().y
  i = (workgroupIdx().z - 1) * workgroupDim().z
      + workitemIdx().z

  if k == 1 || k >= sizes[3] ||
     j == 1 || j >= sizes[2] ||
     i == 1 || i >= sizes[1]
    return
  end

  @inbounds begin
    u_ijk = u[i, j, k]
    v_ijk = v[i, j, k]

    du = Du * _laplacian(i, j, k, u)
         - u_ijk * v_ijk^2 + F * (1.0 - u_ijk)
         + noise * rand(Uniform(-1, 1))

    dv = Dv * _laplacian(i, j, k, v)
         + u_ijk * v_ijk^2 - (F + K) * v_ijk

    u_temp[i, j, k] = u_ijk + du * dt
    v_temp[i, j, k] = v_ijk + dv * dt
  end

  return nothing
end
```

Frontier on-node scalability using AMDGPU.jl

7-point stencil



$$fetch\_size_{effective} = \left[L^3 - 8 - 12\,(L-2)\right] \cdot sizeof(T) \quad (4a)$$

$$write\_size_{effective} = (L-2)^3 \cdot sizeof(T) \quad (4b)$$

$$bandwidth_{effective} = \frac{(fetch\_size + write\_size)_{effective}}{kernel\_time} \quad (5a)$$

$$bandwidth_{total} = \frac{(FETCH\_SIZE + WRITE\_SIZE)_{rocprof}}{kernel\_time} \quad (5b)$$

Table 2: Average bandwidth comparison of different stencil implementations on a single GPU.

Kernel	Bandwidth (GB/s) Effective	Total
Julia GrayScott.jl		
- 2-variable (application)	312	570
- 1-variable no random	312	625
HIP single variable	599	1,163
Theoretical peak MI250x	1,600	

Table 3: rocprof outputs for HIP 1-variable and Julia Gray-Scott.jl implementations

kernel	HIP	GrayScott.jl	
metric	1-var	1-variable no random	2-variable (application)
wgr	256	512	512
lds	0	29,184	29,184
scr	0	8,192	8,192
FETCH_SIZE (GB)	25.08	25.40	50.80
WRITE_SIZE (GB)	8.35	8.38	16.78
TCC_HIT (M)	9.14	10.80	24.60
TCC_MISS (M)	8.36	8.69	17.19
Avg Duration (ms)	28.74	54.03	111.07

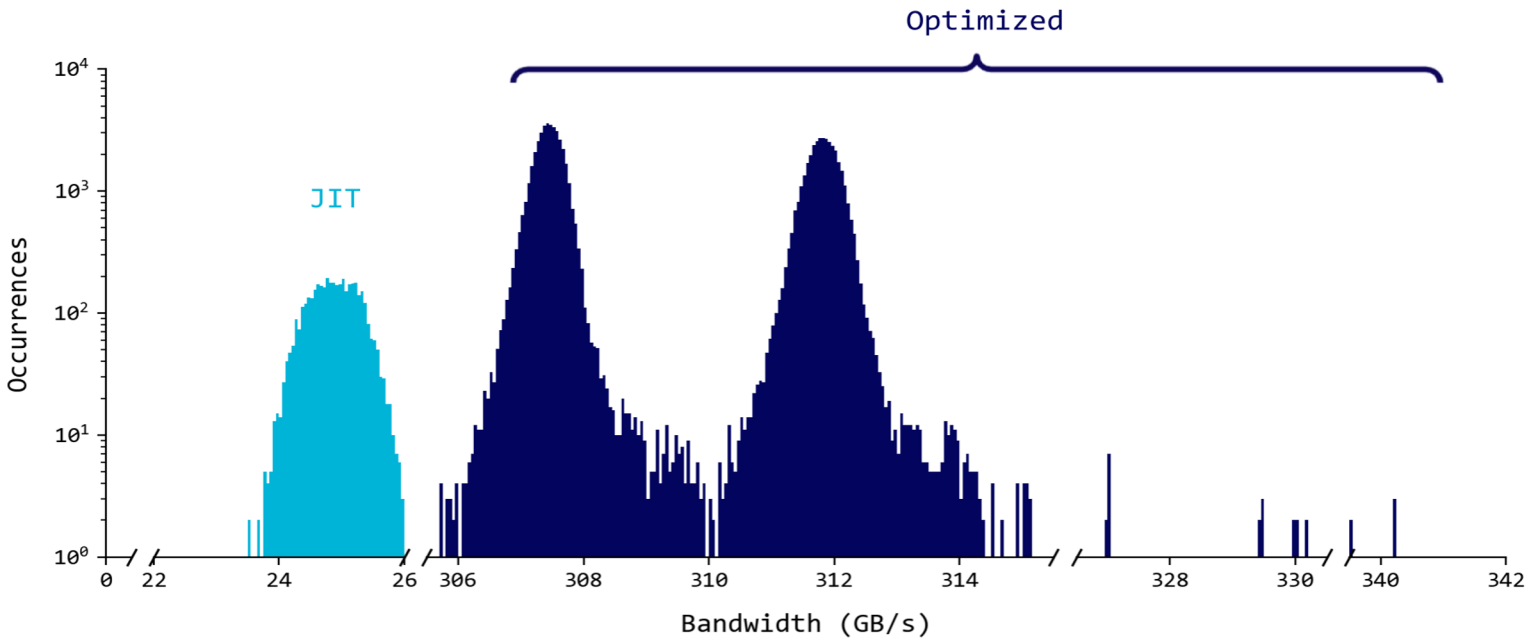
Listing 4: GrayScott.jl application kernel unique memory loads (14) and store (2) in LLVM-IR

```
  %94 = load double, double addrspace(1)*  %93, align 8
%103 = load double, double addrspace(1)* %102, align 8
%107 = load double, double addrspace(1)* %106, align 8
%110 = load double, double addrspace(1)* %109, align 8
%114 = load double, double addrspace(1)* %113, align 8
%117 = load double, double addrspace(1)* %116, align 8
%122 = load double, double addrspace(1)* %121, align 8
%126 = load double, double addrspace(1)* %125, align 8
%312 = load double, double addrspace(1)* %311, align 8
%315 = load double, double addrspace(1)* %314, align 8
%318 = load double, double addrspace(1)* %317, align 8
%321 = load double, double addrspace(1)* %320, align 8
%325 = load double, double addrspace(1)* %324, align 8
%329 = load double, double addrspace(1)* %328, align 8
...
store double %345, double addrspace(1)* %353, align 8
store double %355, double addrspace(1)* %363, align 8
```

Julia AMDGPU.jl reaches ~50% bandwidth (performance) of HIP
No surprises on: FETCH/WRITE SIZE, LLVM-IR
rocprof reports more activity “lds” on Julia

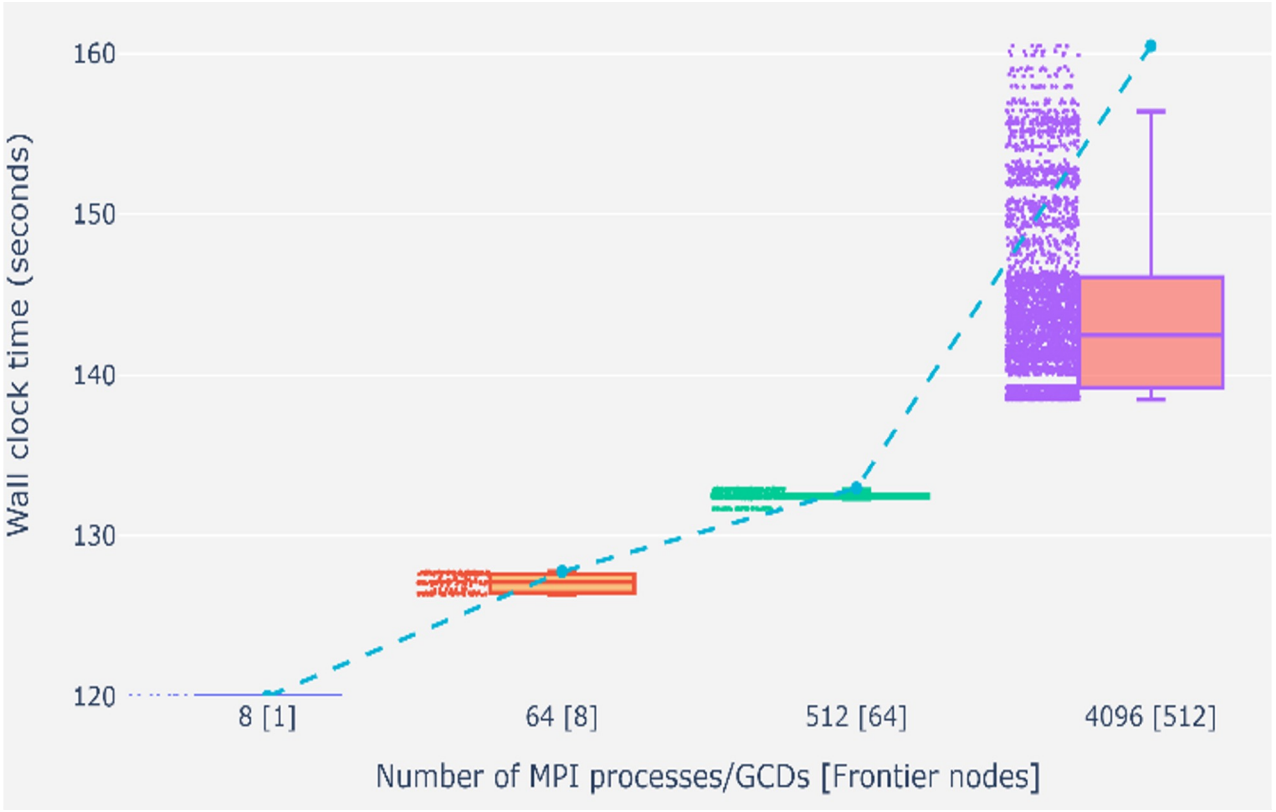
Frontier on-node scalability using AMDGPU.jl for several GPUs

Bandwidth distribution for 4,096 GCD (GPUs) and 20 timesteps. HIP ~ 600 GB/s (800 GB/s claimed on MI250x), Theoretical Peak on MI250x = 1,600 GB/s



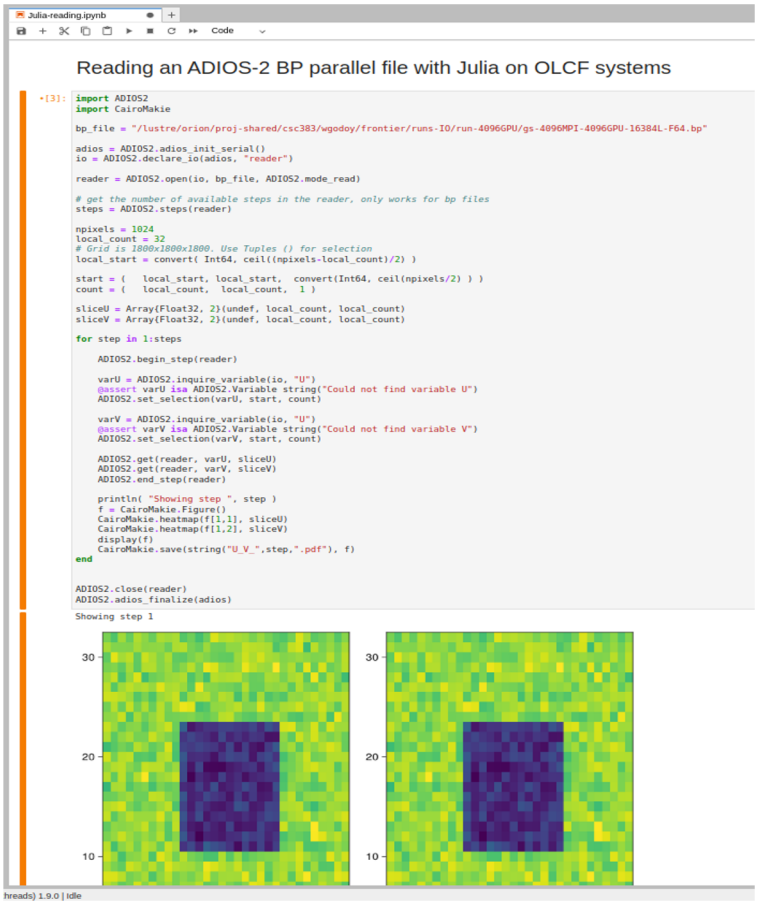
GrayScott.jl Weak Scaling on Frontier

- Tested successfully up to 512 nodes (5% of Frontier) 1 GCD/MPI proc using MPI.jl
- Tried 4K nodes (50% of Frontier) resulted in a libfabric error
- 2-3% variability up to 64 nodes
- 12-15% variability at 512 nodes



Data analysis on JupyterHub at OLCF

- https://jupyter.olcf.ornl.gov/
- We launched a Julia kernel on JupyterHub to read and analyze data
- We read with ADIOS2.jl and visualize with Makie.jl
- JIT and TTFX (time to first plot) can be a nuance
- Pluto.jl?



Godoy, W. F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., Parashar M. ... & Klasky, S. (2020). Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, *12*, 100561. https://doi.org/10.1016/j.softx.2020.100561

Community Efforts in HPC: more frameworks written in Julia

HPC "backends":

- https://juliagpu.org/: AMDGPU.jl , CUDA.jl OneAPI.jl , Metal.jl
- KernelAbstractions.jl, JACC.jl
- MPI.jl
- Threads (part of Base)
- ADIOS2.jl , HDF5.jl

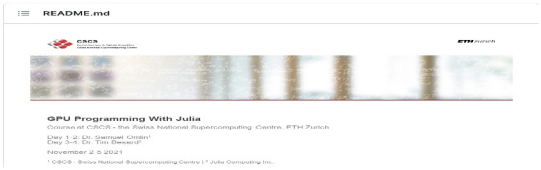
Monthly HPC Call (Valentin Churavy, MIT)

Porting miniWeather App to Julia (Youngsung Kim, Hyun Kang, and Sarat Sreepathi, CSED)

Julia + Sunway + QC at SC22

OAK RIDGE National Laboratory

https://github.com/omlins/julia-gpu-course



https://enccs.github.io/Julia-for-HPC

https://github.com/CliMA

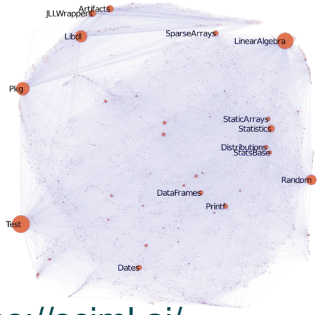
https://github.com/SunnySuite/Sunny.jl

https://docs.dftk.org/stable

https://juliaastro.github.io/dev

https://github.com/JuliaParallel

Top15 most popular packages

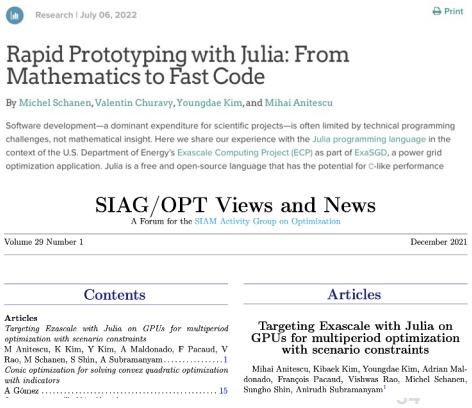


https://sciml.ai/

General Atomics:

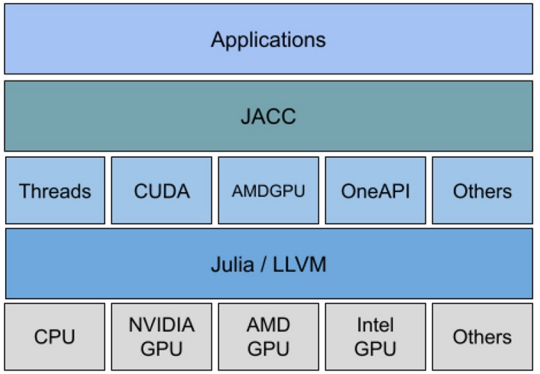
https://github.com/ProjectTorreyPines

ECP ExaSDG on Summit



JACC.jl (Julia ACCelerated), What is that??

- ***Think in Kokkos, but now imagine that it is easy to use***
- The metaprogramming and performance portability model of Julia
  - One "parallel_for" code running everywhere
- **JACC is a unified Julia front-end in top of multiple backends**
  - Threads (CPUs), CUDA (NVIDIA GPUs), AMDGPU (AMD GPUs), and OneAPI (Intel GPUs)
- Hide low-level and device specific implementation
  - Memory, granularity, etc.
- Improve programming productivity for Science and HPC
- A growing community (family)
  - BNL(NERSC), Argonne, MIT, ETHZ, FI/CCQ, …
  - You are welcome to join (JACC meetings once a month)



What is a parallel_for?

- *For loops that are "ideally" independent*

```
for i in 1:Nx
  for j in 1:Ny
      c[i,j] = a[i,j] + b[i,j]
  end
end
```

Domain

Kernel workload per (i,j)

parallel_for(domain, kernel, args…)

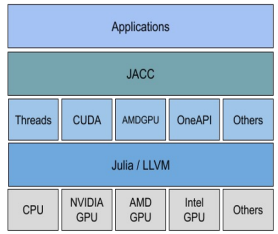
parallel_for((Nx,Ny), add2D, a, b, c)

function add2D

```
  c[i,j] = a[i,j] + b[i,j]
```

end

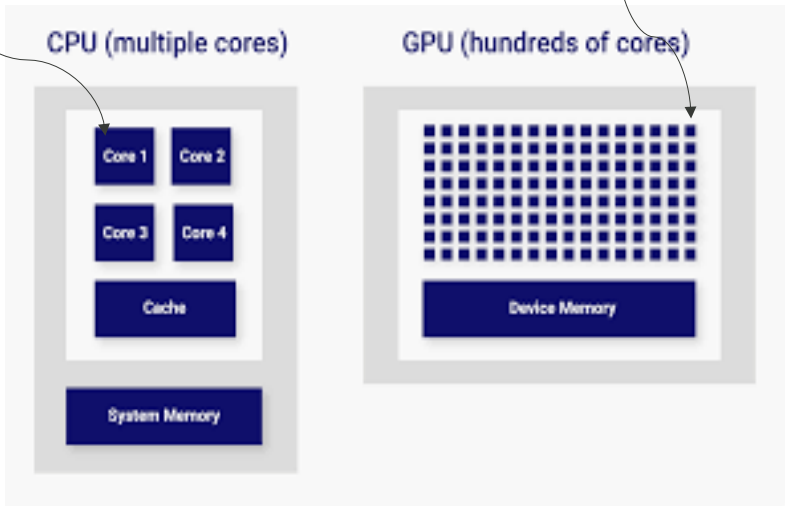
https://github.com/JuliaORNL/JACC.jl



JACC
JULIA FOR ACCELERATORS

Coarse granularity (i,j)k-(i,j)k+Ncore

Fine granularity (i,j)



JACC model, How to use it??

- ***Descriptive, not prescriptive***
- Two main API components
  - **Memory: JACC.array, JACC.shared** high-level: **JACC.ones/JACC.zeros**
    - An alias to the corresponding back end memory
  - **Kernels: JACC.parallel_for** and **JACC.parallel_reduce**
    - Kernel and arguments passed to functions
- Back end selection pre-compilation: **JACC.set_backend("AMDGPU");** LocalPreferences.toml: [JACC] backend: **Threads, CUDA, AMDGPU**

```
u = JACC.ones(T, size_x + 2, size_y + 2, size_z + 2)
v = JACC.zeros(T, size_x + 2, size_y + 2, size_z + 2)

u_temp = JACC.zeros(T, size_x + 2, size_y + 2, size_z + 2)
v_temp = JACC.zeros(T, size_x + 2, size_y + 2, size_z + 2)

offsets = JACC.array(mcd.proc_offsets)
sizes = JACC.array(mcd.proc_sizes)

d::Int64 = 6
minL = Int64(settings.L / 2 - d)
maxL = Int64(settings.L / 2 + d)

# ncenter_cells = maxL - minL + 1
Lx, Ly, Lz = mcd.proc_sizes[1], mcd.proc_sizes[2],
mcd.proc_sizes[3]

JACC.parallel_for((Lx, Ly, Lz), _init_fields_kernel!,
u, v, offsets, sizes, minL, maxL)
```

Gray-Scott simulation code

https://github.com/JuliaORNL/JACC.jl

How is JACC implemented??

- *The simpler the better, use everything that Julia can provide*
- One implementation per backend

```
#JACC.Array and JACC.parallel_for on top of
    Threads
function __init__()
  const JACC.Array = Base.Array{T,N} where {T,N}
end
#Unidimensional
function parallel_for(N::I, f::F, x...) where {I<:
    Integer,F<:Function}
  Threads.@sync Threads.@threads for i in 1:N
    f(i, x...)
  end
end
#Multidimensional
function parallel_for((M, N)::Tuple{I,I}, f::F, x
    ...) where {I<:Integer,F<:Function}
  Threads.@sync Threads.@threads for j in 1:N
    for i in 1:M
      f(i, j, x...)
    end
  end
end
```

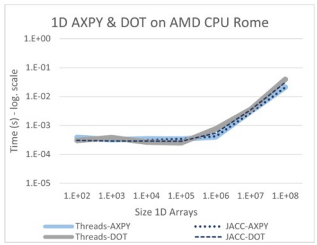
```
#JACC.Array and JACC.parallel_for on top of CUDA
function __init__()
  const JACC.Array = CUDA.CuArray{T,N} where {T,N}
end
#Unidimensional
function _parallel_for_cuda(f, x...)
  i = ( blockIdx().x - 1) * blockDim().x +
      threadIdx().x
  f(i, x...)
  return nothing
end
function JACC.parallel_for(N::I, f::F, x...) where
    {I<:Integer,F<:Function}
  maxPossibleThreads = attribute(device(),CUDA.
      DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X)
  cuda_threads = min(N, maxPossibleThreads)
  cuda_blocks = ceil(Int, N/cuda_threads)
  CUDA.@sync @cuda threads=cuda_threads blocks=
      cuda_blocks _parallel_for_cuda(N, f, x...)
end
#Multidimensional
function _parallel_for_cuda_MN(f,x...)
  i = ( blockIdx().x - 1) * blockDim().x +
      threadIdx().x
  j = ( blockIdx().y - 1) * blockDim().y +
      threadIdx().y
  f(i, j, x...)
  return nothing
end
function JACC.parallel_for((M, N)::Tuple{I,I}, f::
    F, x...) where {I<:Integer,F<:Function}
  numThreads = 16
  Mthreads = min(M, numThreads)
  Nthreads = min(N, numThreads)
  Mblocks = ceil(Int, M/Mthreads)
  Nblocks = ceil(Int, N/Nthreads)
  CUDA.@sync @cuda threads=(Mthreads, Nthreads)
      blocks=(Mblocks, Nblocks)
      _parallel_for_cuda_MN(f, x...)
end
```

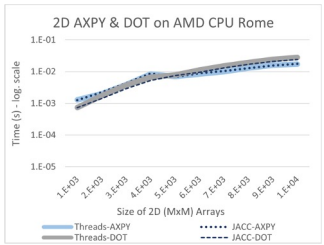
OAK RIDGE National Laboratory

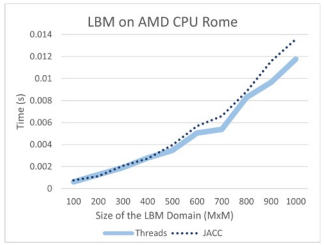
JACC
JULIA FOR ACCELERATORS

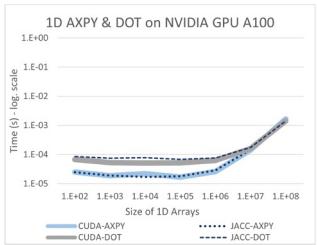
https://github.com/JuliaORNL/JACC.jl

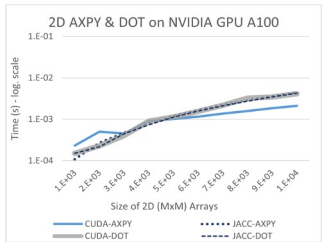
OK, but this is HPC, What about performance??

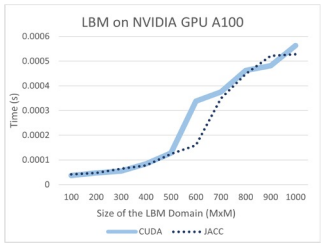


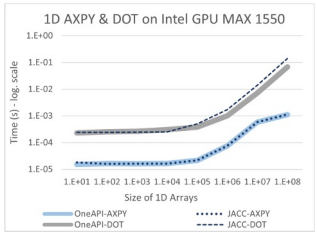


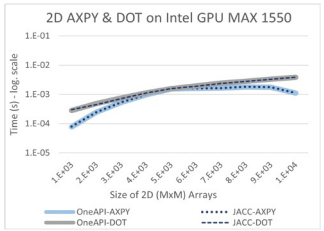


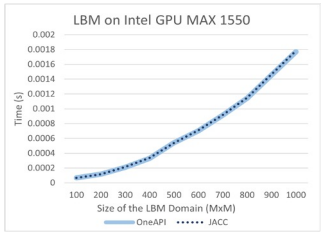


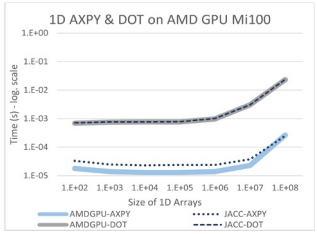


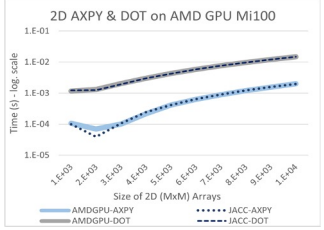


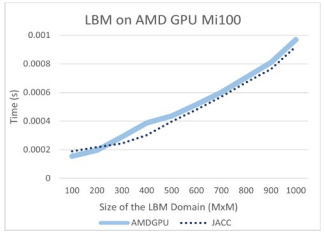












Julia as a unifying end-to-end workflow language on the Frontier exascale system. SC WORKS 2023

Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. IPDPS HIPS 2023

SC24 WACCPD

JACC: Leveraging HPC Meta-Programming and Performance Portability with the Just-in-Time and LLVM-based Julia Language

Pedro Valero-Lara, William F. Godoy, Het Mankad, Keita Teranishi, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{valerolarap},{godoywf},{mankadhy},{teranishik},{vetter}@ornl.gov

Johannes Blaschke
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
jpblaschke@lbl.gov

Michel Schanen
Argonne National Laboratory
Lemont, IL, USA
mschanen@anl.gov

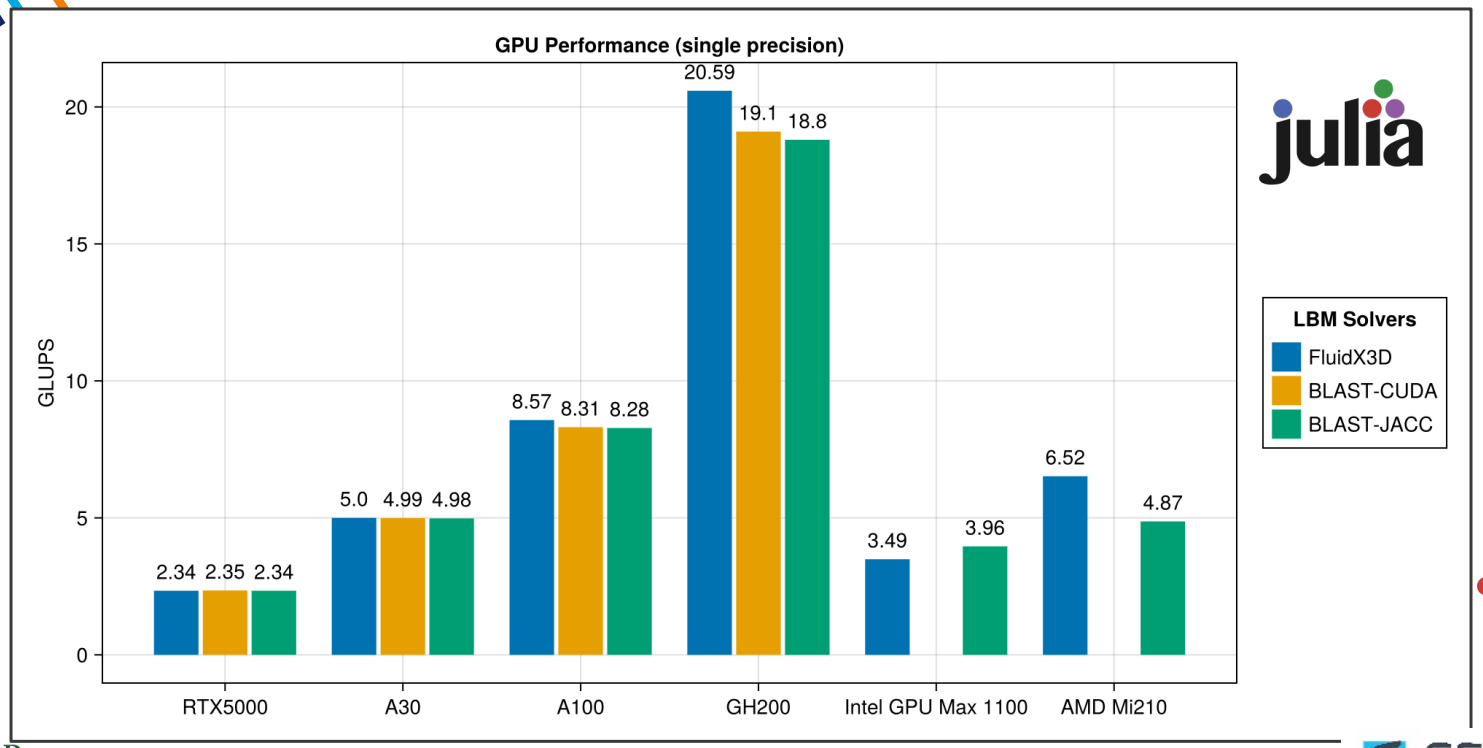
JACC
JULIA FOR ACCELERATORS

OAK RIDGE National Laboratory

- Lattice: D3Q19
- Mesh: 256x256x256
- Streaming algorithm: Esoteric Pull
- Test case: Taylor Green Vortex

LBM Prototype in Julia

Boltzmann Lattice Advanced Simulation Tool



julia

JACC
JULIA FOR ACCELERATORS

OAK RIDGE National Laboratory

https://cerfacs.fr/

CERFACS

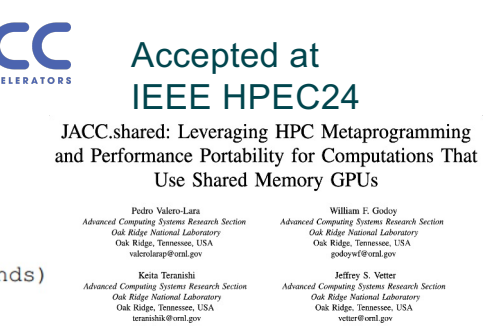
Ongoing efforts??

JACC
JULIA FOR ACCELERATORS

Accepted at
IEEE HPEC24

- JACC.shared
  - Exploiting high-bandwidth programable in-chip GPUs memory

```
function spectral(i, j, image, filter, num_bands)
 for b in 1:bands
    @inbounds image[b, i, j] *= filter[j]
  end
end

function spectral_shared(i, j, image, filter,
    num_bands)
 #Shared memory initialization
 filter_shared = JACC.shared(filter)
 for b in 1:bands
  @inbounds image[b, i, j] *= filter_shared[j]
 end
end

num_bands = 60
num_voxel = 10_240
size_voxel = 64*64
image = init_image(Float32,
        num_bands, num_voxel, size_voxel)
filter = init_filter(Float32, size_voxel)
jimage = JACC.Array(image)
jimage_shared = JACC.Array(image)
jfilter = JACC.Array(filter)
JACC.parallel_for((num_voxel,size_voxel),
     spectral[_shared], jimage, jfilter, num_bands)
```

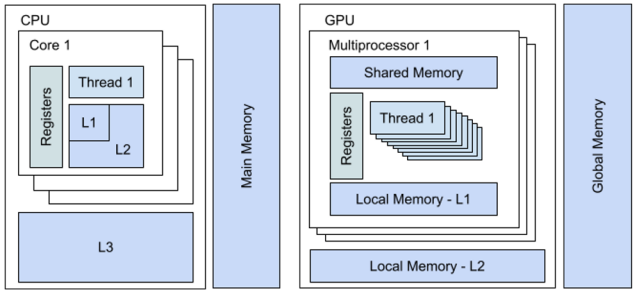

JACC.shared: Leveraging HPC Metaprogramming and Performance Portability for Computations That Use Shared Memory GPUs

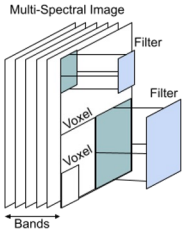
Pedro Valero-Lara
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
valerolarap@ornl.gov

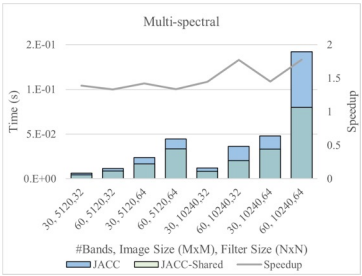
William F. Godoy
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
godoywf@ornl.gov

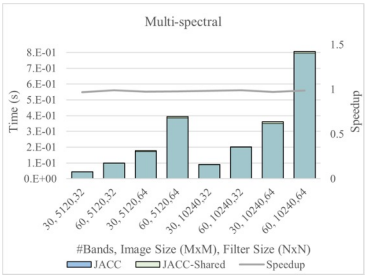
Keita Teranishi
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
teranishik@ornl.gov

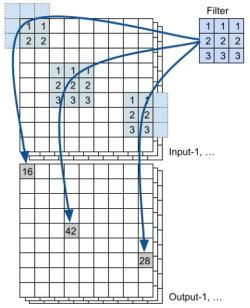
Jeffrey S. Vetter
Advanced Computing Systems Research Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vetter@ornl.gov

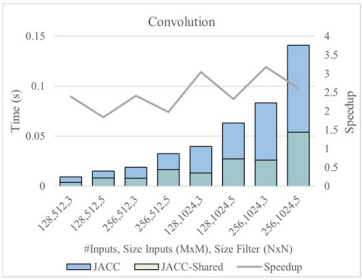


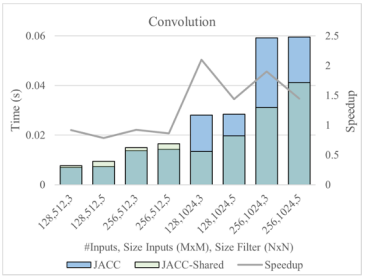












OAK RIDGE National Laboratory

https://github.com/JuliaORNL/JACC.jl

Ongoing JACC efforts

- JACC.experimental
  - A separate JACC module to explore new ideas
- JACC Proxies
  - Compare JACC in science workloads (LULESH, XSBench, BabelStream, Hartree-Fock)
- JACC.BLAS
  - BLAS library on top of JACC
- JACC.multi
  - Support for multi-device
- JACC.auto
  - Support for auto-tuning
- Task-based – JACC.async
  - DAGGER.jl, IRIS - R&D100

OAK RIDGE National Laboratory

JACC
JULIA FOR ACCELERATORS

Presented at SC24 WACCPD

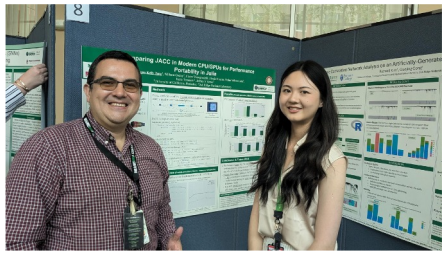
JACC: Leveraging HPC Meta-Programming and Performance Portability with the Just-in-Time and LLVM-based Julia Language

Pedro Valero-Lara, William F. Godoy, Het Mankad, Keita Teranishi, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{valerolarap},{godoywf},{mankadhy}, {teranishik},{vetter}@ornl.gov

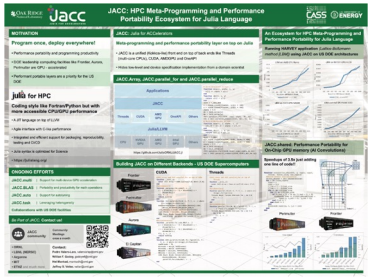
Johannes Blaschke
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
jpblaschke@lbl.gov

Michel Schanen
Argonne National Laboratory
Lemont, IL, USA
mschanen@anl.gov

Best ORNL CS intern poster by Kelly Tang



JACC: SC24 Best Poster Finalist (6/120)



SC24 AI4Science using Julia

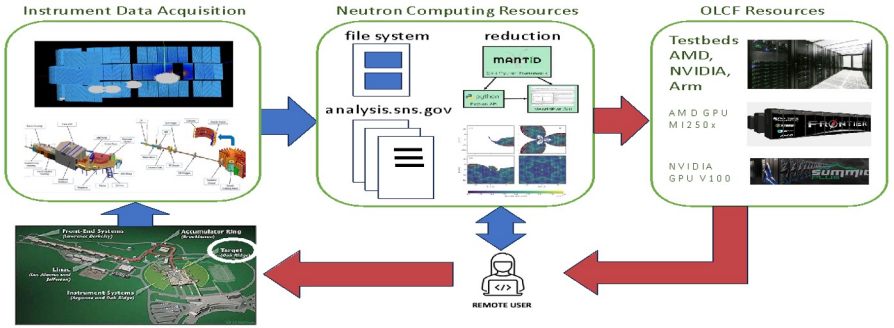
ChatBLAS: The First AI-Generated and Portable BLAS Library

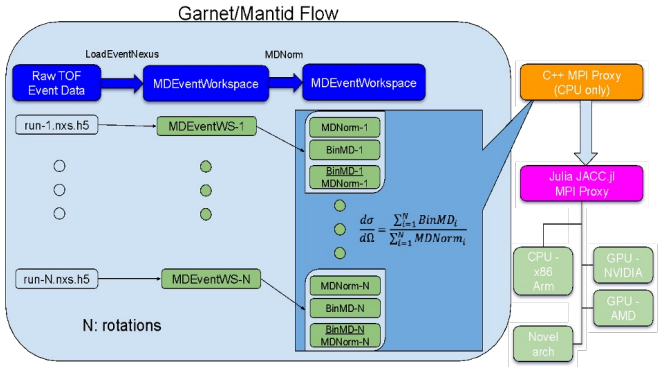
SC24: Julia for HPC 1st Tutorial and 3rd BoF w/ MIT and LBNL

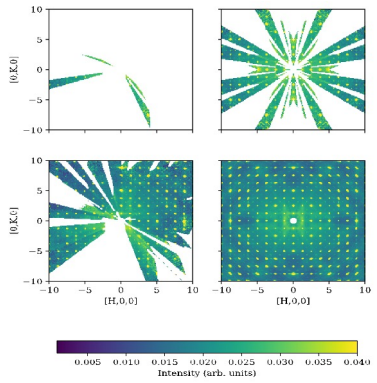
https://github.com/JuliaORNL/JACC.jl

Ongoing JACC efforts: facilities

- Integrated Research Infrastructure: provide an **accessible** performance portable ecosystem
- CPU only workflows -> CPU/GPU on HPC systems







OAK RIDGE National Laboratory

Best Paper at SC24 XLOOP

Integrating ORNL's HPC and Neutron Facilities with a Performance-Portable CPU/GPU Ecosystem

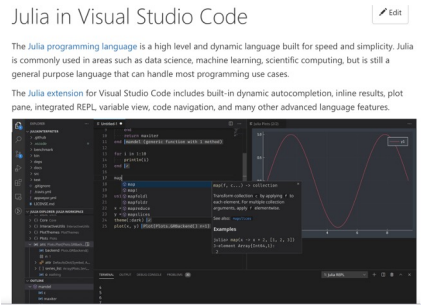
Steven E. Hahn, Philip W. Fackler, William F. Godoy, Ketan Maheshwari, Zachary Morgan, Andrei T. Savici, Christina M. Hoffmann, Pedro Valero-Lara, Jeffrey S. Vetter, and Rafael Ferreira da Silva
Oak Ridge National Laboratory, Oak Ridge, TN, USA
{hahnse,facklerpw,godoywf,km0,morganzj,saviciat,choffmann,valerolarap,vetter,silvarf}@ornl.gov


```
function binEvents!(h::Hist3, events::AbstractArray,
    transforms::Array1{SquareMatrix3c})
  JACC.parallel_for(
    (length(transforms), size(events, 2)),
    (n, i, t) -> begin
      @inbounds begin
        op = t.transforms[n]
        v = op * C3[t.events[6, i], t.events[7, i],
            t.events[8, i]]
        atomic_push!(t.h, v[1], v[2], v[3],
        t.events[1, i])
      end
    end,
    (h = h, events, transforms),
  )
end
```

Listing 3. **MiniVATES.jl BinMD CPU/GPU implementation using JACC.jl.**

Where to get started?

- Pick a gentle tutorial: https://techytok.com/from-zero-to-julia/
- https://github.com/ornl-training/julia-basics (training by WF Godoy & Philip Fackler) OLCF Tutorial: https://juliaornl.github.io/TutorialJuliaHPC/applications/GrayScott/01-Solver.html
- Use VS Code as the official IDE + debugger
- JuliaCon talks are available on YouTube
- https://discourse.julialang.org/ Stackoverflow might be outdated, https://julialang.slack.com/
- Julia docs and standard library: https://docs.julialang.org/en/v1/
- Learn: Project.toml, Testing.jl @testset @test, Pluto.jl , CUDA.jl/AMDGPU.jl , JACC.jl, KernelAbstractions.jl, LinearAlgebra.jl , Makie.jl , Plots.jl and Flux.jl (AI/ML), how to build a sysimage with PackageCompiler.jl
- **Pick problems you care about! Let us know if you're interested in a hackathon.**
- Patience and community reliance: learning a language is a big investment.

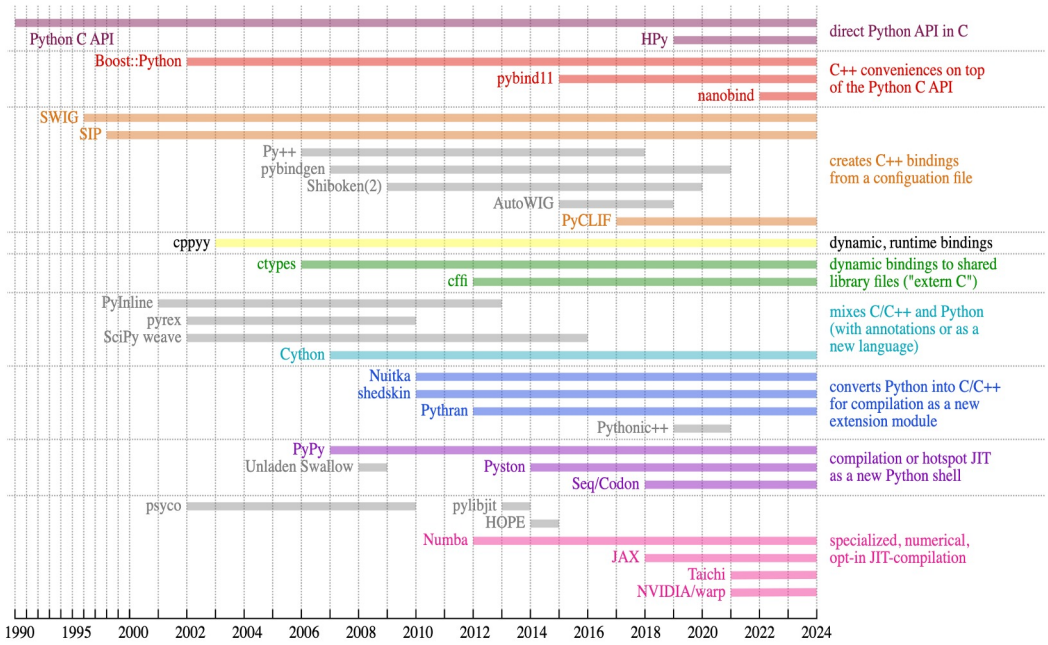


https://williamfgc.github.io/programming/scientific-computing/2020/07/22/first-project-julia-language.html

Early Exploration of Mojo

When Python is not enough…and it's a fragmented world

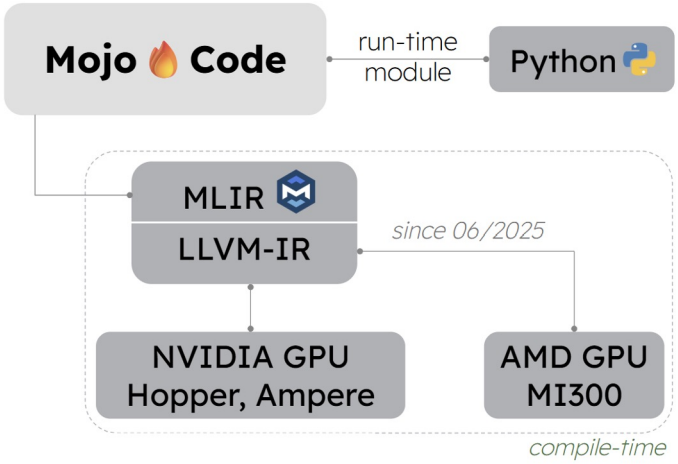
Making Python "faster"

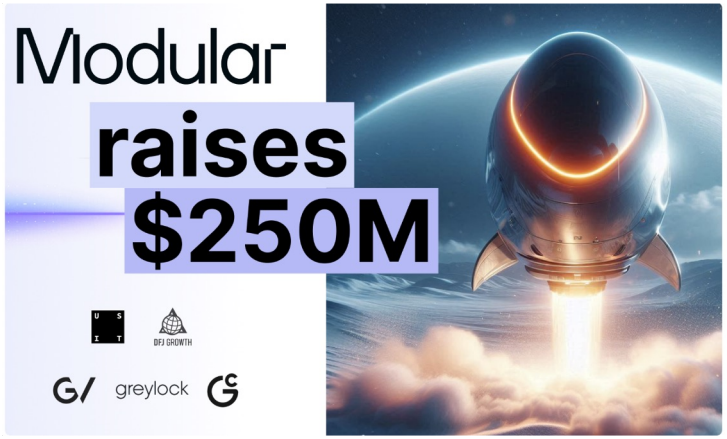


https://raw.githubusercontent.com/jpivarski-talks/2023-05-01-hsf-india-tutorial/main/img/history-of-bindings-2.svg

Mojo overview

- Mojo combines **Python's syntax** and **ecosystem** with **high performance**
- **GPU portability in the standard library:** NVIDIA, AMD, and Apple Silicon* GPUs
- **MLIR compilation**
- **Industry-funded**
- **Memory safety** via variable lifetime
- Projected **open-source** in 2026





* The most up-to-date Mojo GPU compatibility list can be found here: https://docs.modular.com/max/packages/#gpu-compatibility

Mojo overview


```mojo
from gpu.host import DeviceContext
from gpu.id import block_dim, block_idx, thread_idx
from layout import Layout, LayoutTensor
...
from python import Python
alias dtype = DType.float32
alias Nx = 1024
alias layout = Layout.row_major(Nx)
alias block_size = 256
alias num_blocks = ceildiv(Nx, block_size)
fn fill_one(tensor: LayoutTensor[mut=True, dtype,layout]):
    var tid = block_idx.x * block_dim.x + thread_idx.x
    if tid < Nx:
        tensor[tid] = 1
fn main()
    ctx = DeviceContext()
    d_u = ctx.enqueue_create_buffer[dtype](nx)
    u_tensor = LayoutTensor[dtype,layout](d_u)
    ctx.enqueue_function[fill_one](u_tensor,
        grid_dim=num_blocks,
        block_dim=block_size
    )
    ctx.synchronize()
    np = Python.import_module("numpy")
    array = np.array(Python.list(1, 2, 3))
    print(array)
```

- Compile-time GPU programming requires tensor type, size, and layout
- GPU kernel launching
- GPU memory model
- GPU kernel execution
- Python interoperability uses a separate runtime approach

Compile with Pixi package manager:

Just-in-Time:

```
pixi run mojo prog.mojo
```

Ahead-of-Time:

```
pixi shell
mojo build prog.mojo
./prog
```

Mojo FAQ:

https://docs.modular.com/mojo/faq/

Project description

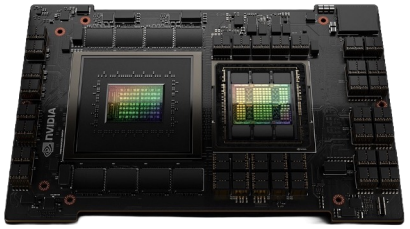
- *Can scientific users benefit from Mojo's performance-portable GPU codes?*
- First comprehensive study on Mojo
- Ported **4 scientific workloads**
  - 2 memory-bound
  - 2 compute-bound

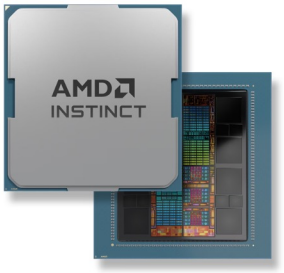
GitHub repo:

https://github.com/tdehoff/Mojo-workloads

- **Goal:** Assess Mojo performance portability vs. vendor-native baselines (CUDA/HIP)
- Tested and benchmarked on ORNL's ExCL nodes:
  - **NVIDIA H100** NVL – 94 GB
  - **AMD MI300A** – 128 GB HBM3

ExCL





Memory-bound workloads

- **BabelStream:** *Copy, Multiply, Triad, Add, Dot* memory benchmarking operations (University of Bristol: github.com/UoB-HPC/BabelStream)

```
fn dot_kernel(a: UnsafePointer[Float64],
              b: UnsafePointer[Float64],
              sums: UnsafePointer[Float64]):
    var s = stack_allocation[TBSize, Float64,
        address_space=AddressSpace.SHARED]()
    var i = block_idx.x * block_dim.x + thread_idx.x
    s[thread_idx.x] = a[i] * b[i]
    barrier()
    var off = block_dim.x // 2
    while off > 0:
        if thread_idx.x < off:
            s[thread_idx.x] += s[thread_idx.x + off]
        off //= 2
    if thread_idx.x == 0:
        sums[block_idx.x] = s[0]
```

- **Seven-point stencil:** Used for modeling diffusion phenomena (AMD lab notes: github.com/amd/amd-lab-notes)

```
fn laplacian_kernel(f: LayoutTensor[mut=True, Float32, layout],
                    u: LayoutTensor[mut=False, Float32, layout],
                    nx: Int, ny: Int, nz: Int,
                    invhx2: Float32, invhy2: Float32,
                    invhz2: Float32, invhxyz2: Float32):
    var i = thread_idx.x + block_idx.x * block_dim.x
    var j = thread_idx.y + block_idx.y * block_dim.y
    var k = thread_idx.z + block_idx.z * block_dim.z
    if 0 < i < nx-1 and 0 < j < ny-1 and 0 < k < nz-1:
        f[i,j,k] = u[i,j,k]*invhxyz2 +
                   (u[i-1,j,k]+u[i+1,j,k])*invhx2 +
                   (u[i,j-1,k]+u[i,j+1,k])*invhy2 +
                   (u[i,j,k-1]+u[i,j,k+1])*invhz2
```

- **Performance metric:** memory bandwidth (GB/s)

Compute-bound workloads

- **miniBUDE:** Models ligand-protein docking (University of Bristol: github.com/UoB-HPC/miniBUDE)

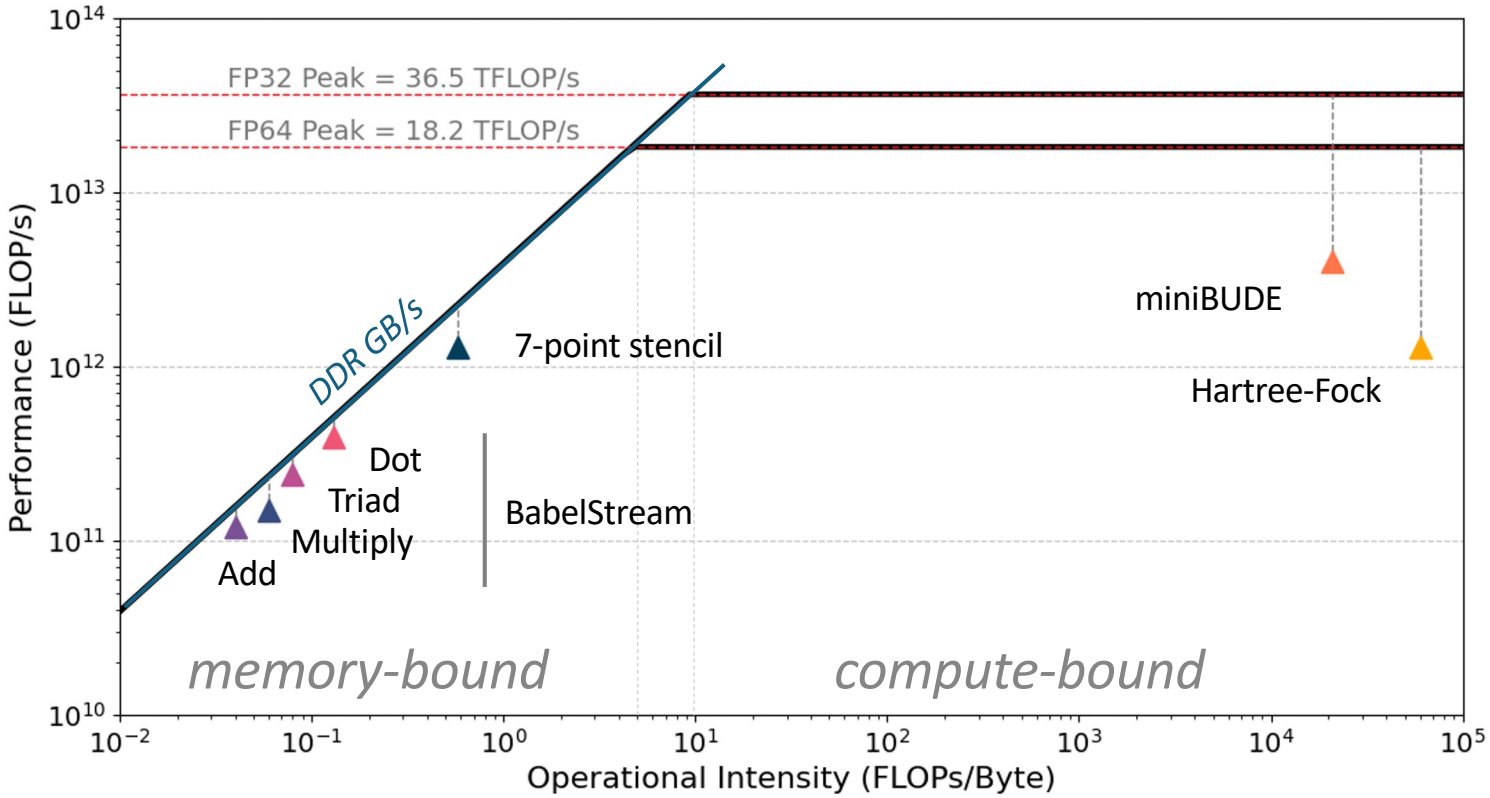
```
fn fasten_kernel[PPWI: Int](natlig: Int, natpro: Int,
                            protein: UnsafePointer[Float32],
                            ligand: UnsafePointer[Float32],
                            etotals: UnsafePointer[Float32],
                            forcefield: UnsafePointer[Float32]):
    var ix = block_idx.x * block_dim.x * PPWI + thread_idx.x
    var etot = InlineArray[Float32, PPWI](fill=0)

    for il in range(natlig):
        var lx = ligand[il*4]; var ly = ligand[il*4+1]; var lz = ligand[il*4+2]
        ...
        for ip in range(natpro):
            var px = protein[ip*4]; var py = protein[ip*4+1]; var pz = protein[ip*4+2]
            var dx = lx - px; var dy = ly - py; var dz = lz - pz
            var dist = sqrt(dx*dx + dy*dy + dz*dz)
            var tmp = 1.0 - dist * 0.1
            etot[0] += tmp * 45.0
            ...
    etotals[ix] = etot[0]
```

- **Hartree-Fock:** Includes atomics; approximates the electron behavior in quantum systems (Argonne NL: github.com/gdfletcher/basic-hf-proxy)

```
fn hartree_fock_kernel(ngauss: Int, schwarz: UnsafePointer[Float64],
                       xpnt: UnsafePointer[Float64], coef: UnsafePointer[Float64],
                       geom: LayoutTensor[mut=True, dtype, geom_layout],
                       dens: LayoutTensor[mut=True, dtype, layout],
                       fock: LayoutTensor[mut=True, dtype, layout]):
    var ijkl = block_idx.x * block_dim.x + thread_idx.x
    var i = ijkl // natoms
    var j = ijkl % natoms
    var eri: Float64 = 0.0
    for ib in range(ngauss):
        for jb in range(ngauss):
            aij = 1.0 / (xpnt[ib] + xpnt[jb])
            dij = coef[ib] * coef[jb] * exp(-xpnt[ib] * xpnt[jb] * aij *
                  pow(geom[i,0]-geom[j,0], 2))
            if abs(dij) > dtol:
                eri += dij * sqrt(aij)
                ...
    Atomic.fetch_add(fock.ptr.offset(i * natoms + j), dens[i,j] * eri)
```

- **Performance metric:** miniBUDE – GFLOP/s, Hartree-Fock – time in ms

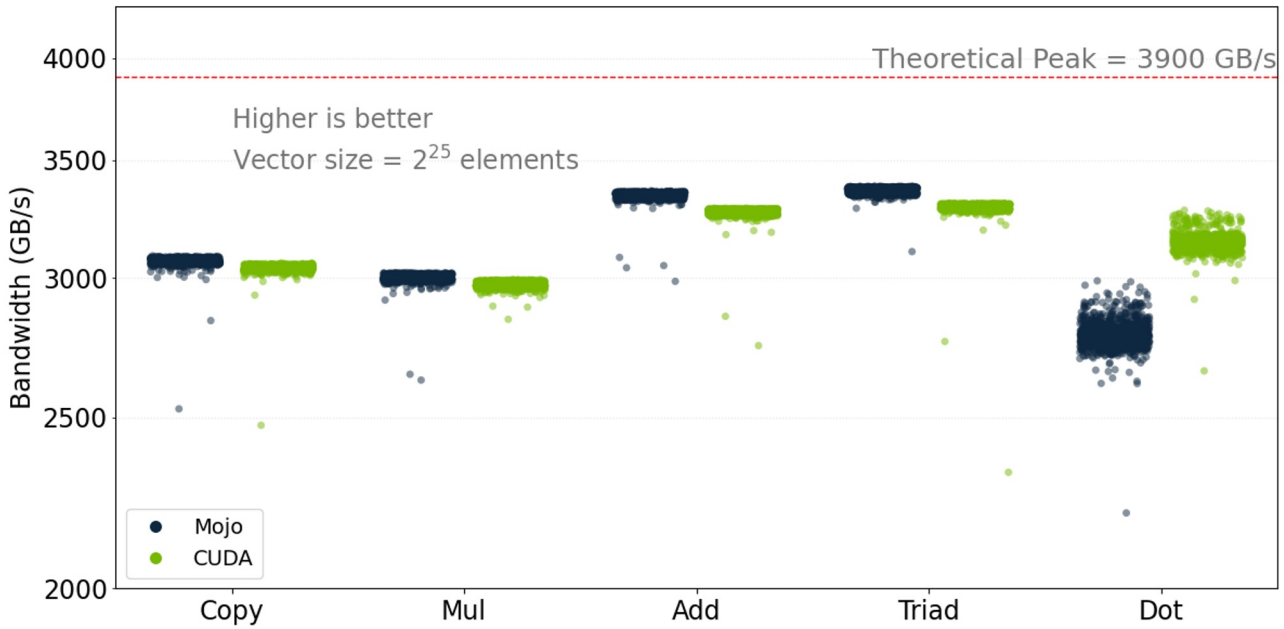
Roofline model: NVIDIA H100



Memory-bound performance results

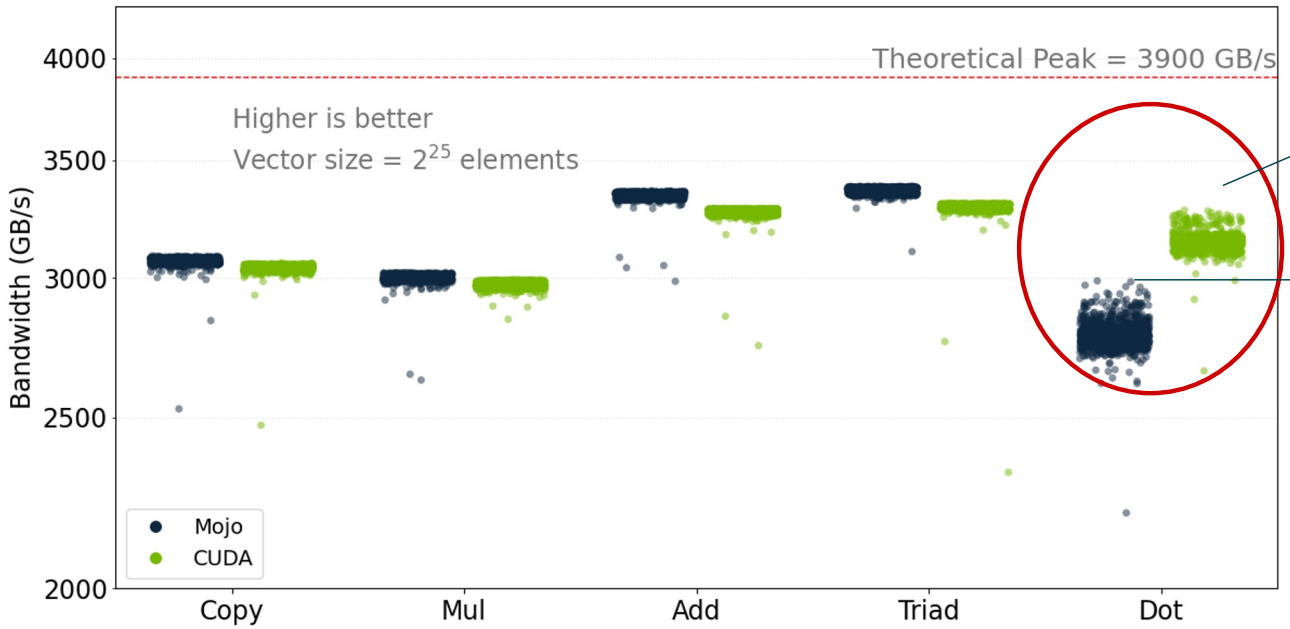
BabelStream on NVIDIA H100

- Input configuration: vector of 2^{25} doubles
- NVIDIA NCU profiler output is available in extra slides



BabelStream on NVIDIA H100

- Input configuration: vector of 2^{25} doubles
- NVIDIA NCU profiler output is available in extra slides

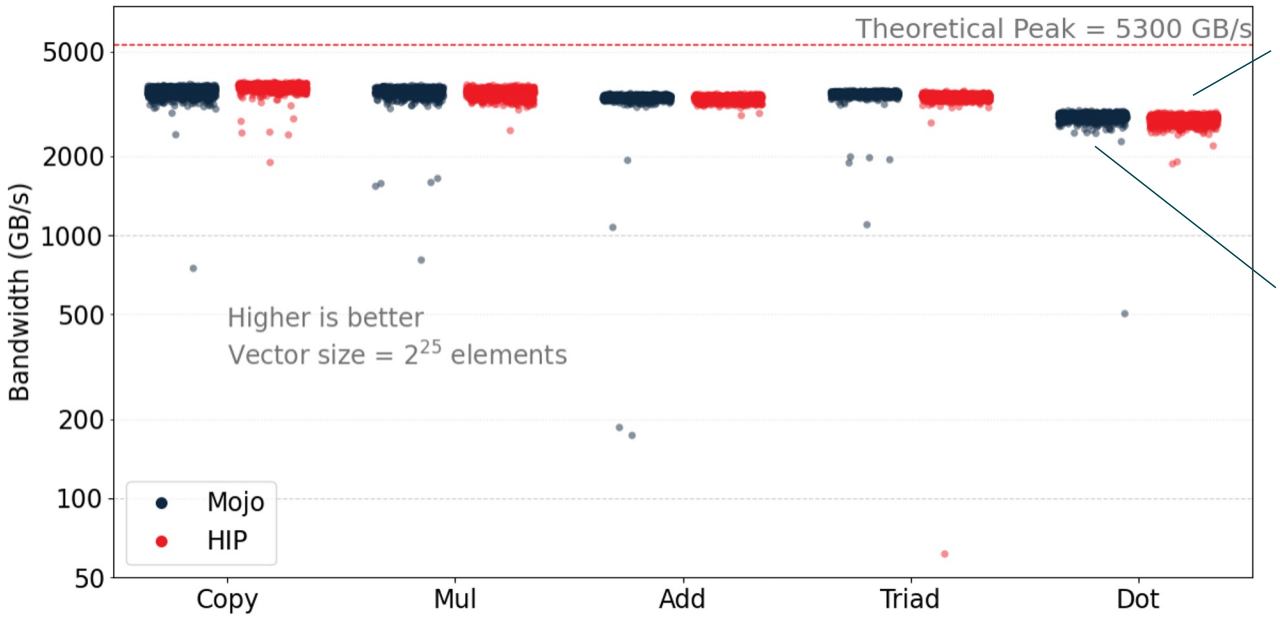


CUDA: Number of GPU thread blocks determined dynamically from SM count

Mojo: Block count fixed at compile time, computed using input size

BabelStream on AMD MI300A

- Input configuration: vector of 2^{25} doubles

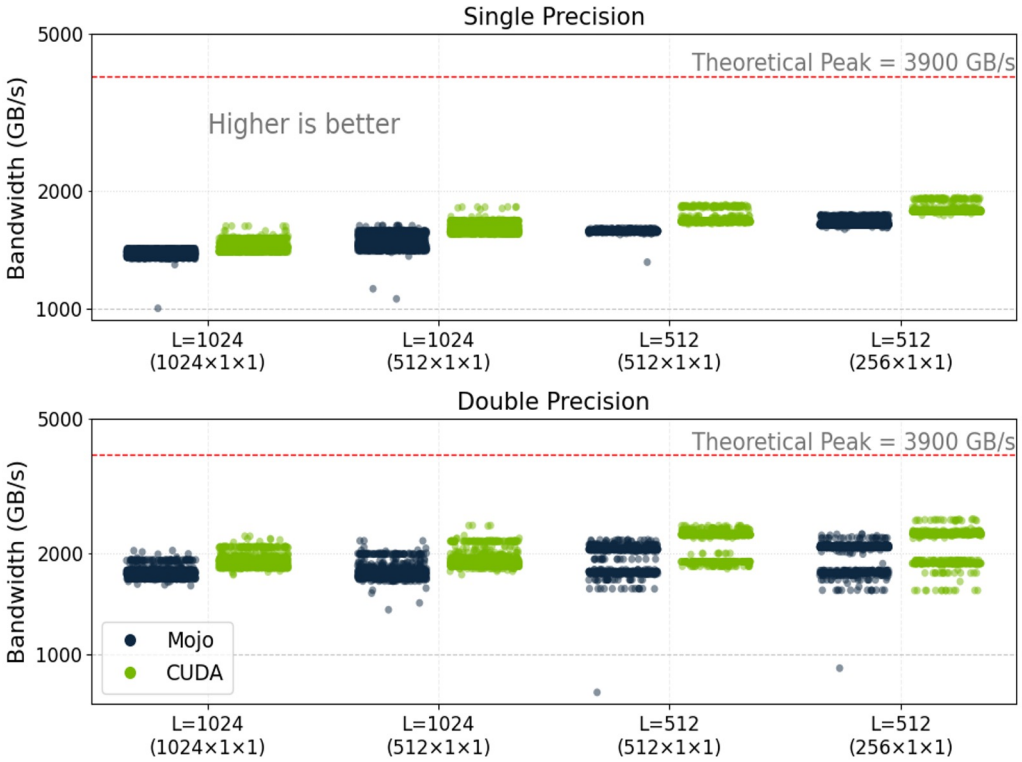


HIP: partial sums array is host-allocated (pinned) via *hipHostMalloc*

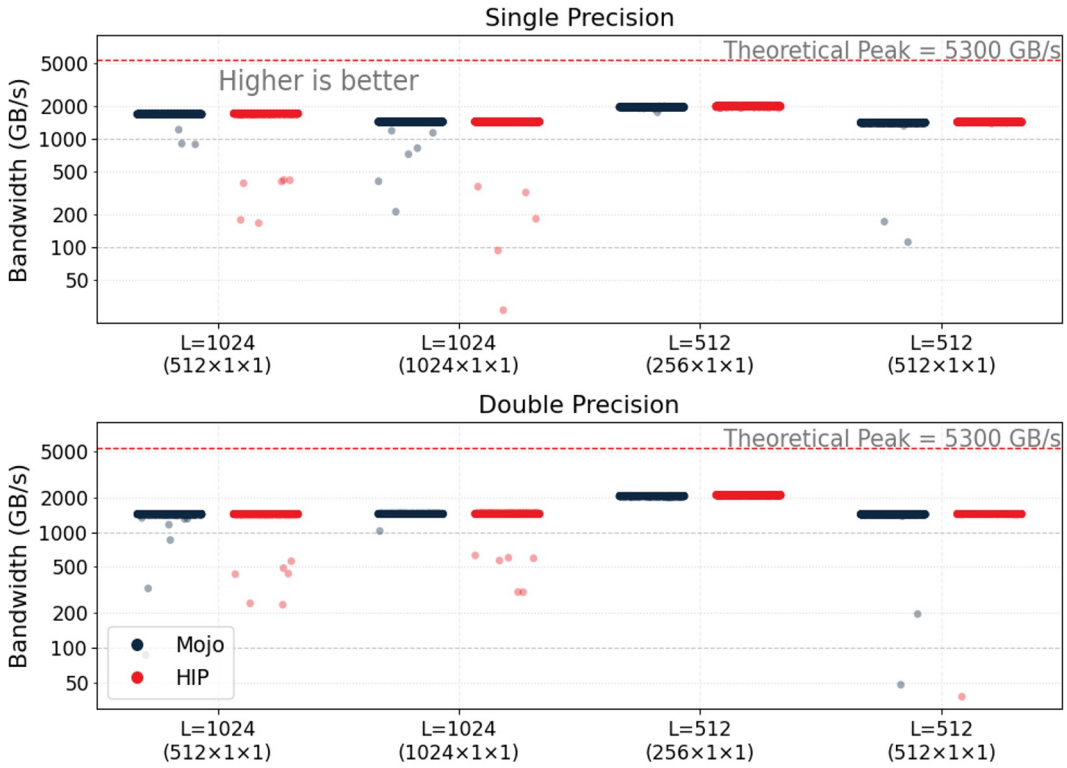
Mojo: partial sums array is device-allocated and copied back to host buffer for the final reduction

7-point stencil on NVIDIA H100

- 3-dimensional input, L denotes dimension size
- Tested with different GPU thread block configurations
- NVIDIA NCU profiler output is available for:
  - L=512, 512x1x1 block
  - L=1024, 1024x1x1 block



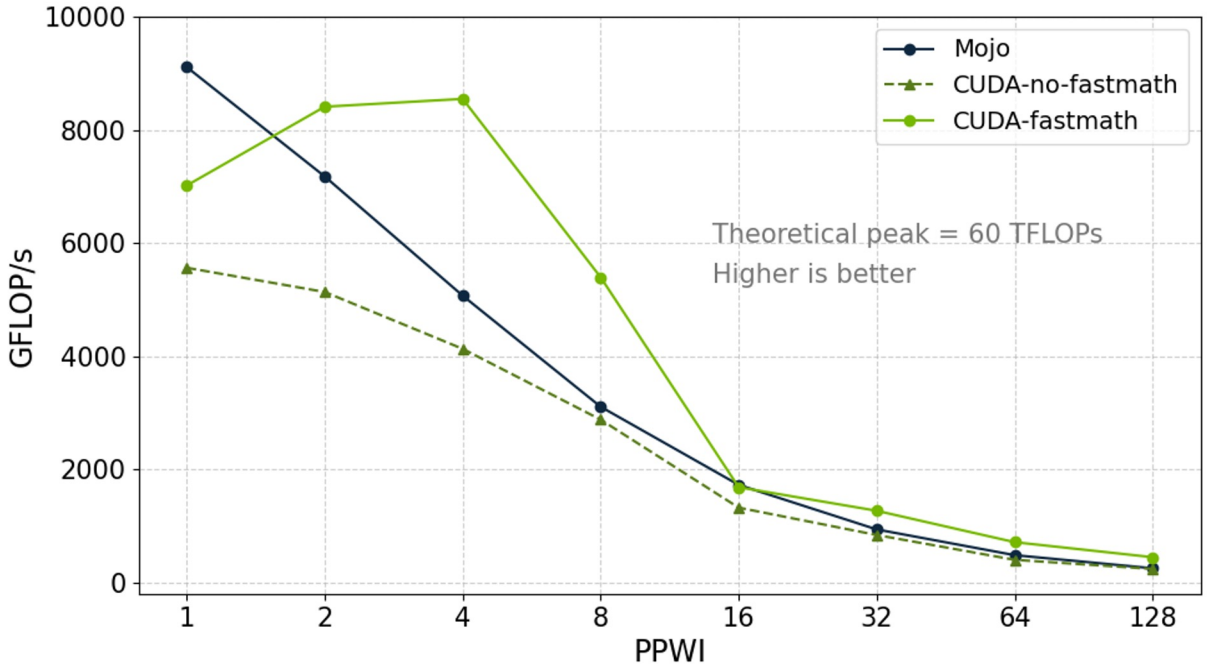
7-point stencil on AMD MI300A



Compute-bound performance results

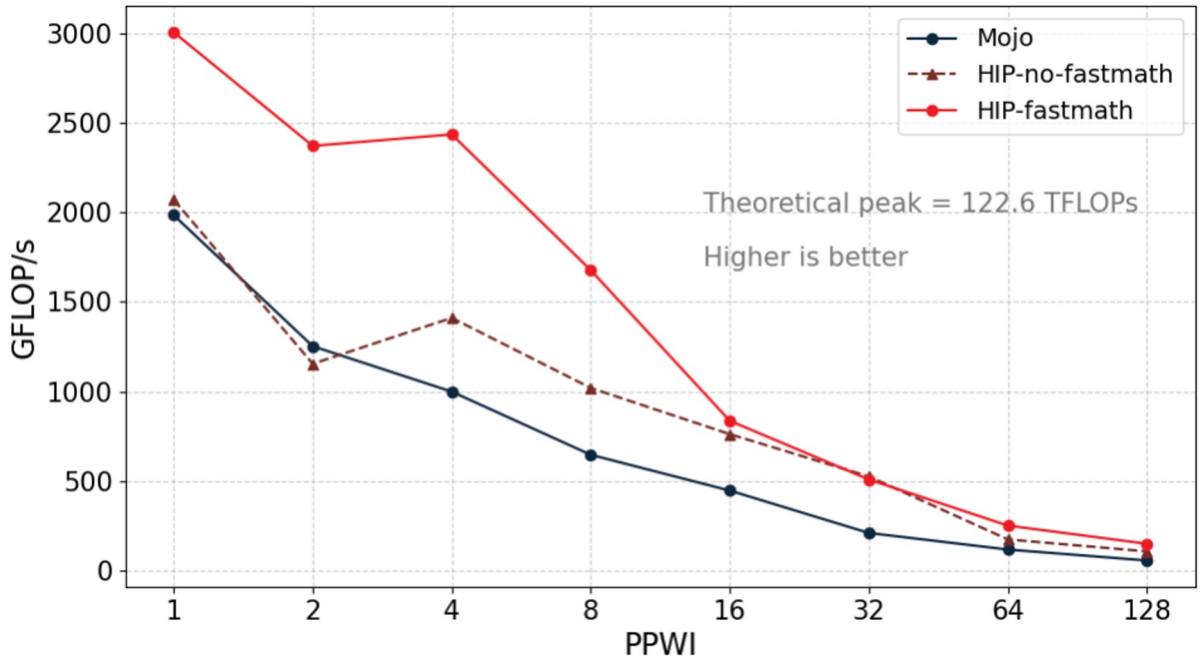
miniBUDE on NVIDIA H100

- Work-group size = 8, **PPWI** (x-axis) varies the **computational workload** per GPU thread



miniBUDE on AMD MI300A

- Work-group size = 8, PPWI (x-axis) varies the **computational workload** per GPU thread



Hartree-Fock (atomics)

Kernel execution duration *(ms)*	**NVIDIA H100**		**AMD MI300A**	
	Mojo	**CUDA**	**Mojo**	**HIP**
a=1024, ngauss=6	147,250	2,652		846
a=256, ngauss=3	187	472	25,266	178
a=128, ngauss=3	21	53	2,765	23
a=64, ngauss=3	3	7	436	4

- NVIDIA H100: Mojo is about **2.5 times faster than CUDA** for small input sizes; at 1024 Mojo's performance degrades sharply
- AMD MI300A: Mojo **significantly underperforms** HIP across input sizes

Performance portability metric (Φ)

- Adapted from Pennycook et al (2021) & Marowka (2025):

$$\overline{\Phi}_{Mojo} = \frac{\sum_{i \in T} e_i(a)}{|\ T\ |} \qquad e_i(a) = \frac{Mojo_{perf-i}}{vendor(CUDA/HIP)_{perf-i}}$$

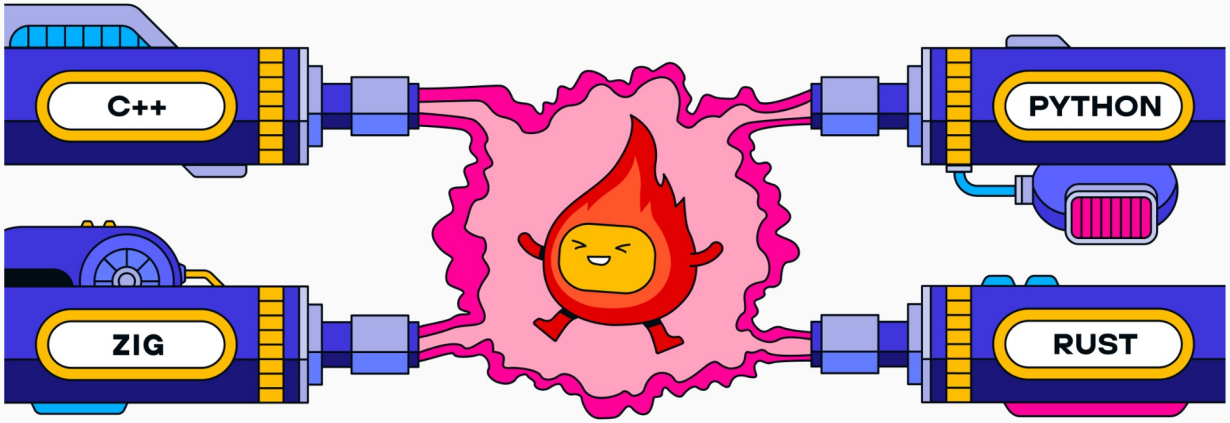
Workload	Φ NVIDIA H100	Φ AMD MI300A	Average Φ
BabelStream	0.78-1.00	1.00	0.96
7-point stencil	0.82-0.87	1.00	0.92
miniBUDE	0.59-0.82	0.38	0.54
Hartree-Fock	>2x	<< 1.0	0.92 (mixed)

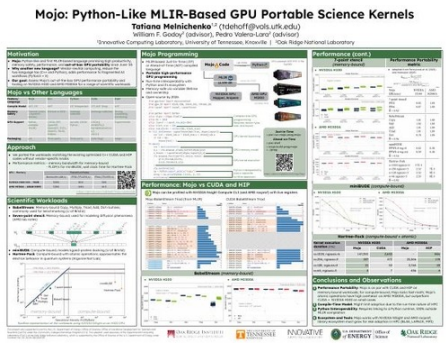
Key observations

1. **Memory-bound:** Mojo performance matched or nearly matched C++ CUDA and HIP
2. **Compute-bound:** Lacks fast-math optimizations; atomic operations need work
3. **Compile-time model:** powerful, but may be not ideal for HPC
4. **Python Interoperability:** Functional, but requires linking against a Python runtime, 100% outside MLIR compilation.
5. **Productivity:** Python-like, but still low-level
6. **Tooling and Ecosystem:** Works with NVIDIA NSight and AMD rocprof; library ecosystem still early-stage

Conclusion

- Our work is **the first comprehensive evaluation** of Mojo's GPU-portable performance
- **Promising but not perfect:** strong results for memory-bound, but workload-dependent
- **MLIR + Python + GPU portability:** strong potential unifying HPC-AI language as it matures
- **Future work:** MojoBLAS





RESEARCH-ARTICLE

Mojo: MLIR-based Performance-Portable HPC Science Kernels on GPUs for the Python Ecosystem

Authors: William Godoy, Tatiana Melnichenko, Pedro Valero-Lara, Wael Elwasif, Philip Fackler, Rafael Ferreira Da Silva, Keita Teranishi, Jeffrey Vetter | Authors Info & Claims

SC Workshops '25: Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis
Pages 2114 - 2128 • https://doi.org/10.1145/3731599.3767573

Questions?

This project was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Community College Internships Program (CCI).This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Computer Science Competitive Portfolios program, MAGMA/Fairbanks project; and the Next Generation of Scientific Software Technologies program, PESO and S4PST projects. This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

BabelStream Mojo vs. CUDA NCU profiling metrics

Nsight Compute CLI (ncu) metric	Copy		Mul		Add		Dot	
	Mojo	CUDA	Mojo	CUDA	Mojo	CUDA	Mojo	CUDA
Duration (ms)	0.202	0.205	0.203	0.208	0.264	0.269	0.215	0.168
Throughputs (%)								
- Compute SM	16.3	28.6	18.2	28.2	15.9	27.3	51.1	11.4
- Memory	69.7	68.9	69.2	68.0	81.7	80.5	69.9	87.6
L1 ai (FLOP/byte)	–		0.06		0.04		0.13	
L2 ai (FLOP/byte)	–		0.08		0.05		0.14	0.13
L3 ai (FLOP/byte)	–		0.12		0.06		0.14	0.13
L1-3 Perf (FLOP/s)	–		1.64 E11	1.61 E11	1.26 E11	1.24 E11	3.5 E11	4.01 E11
Registers	16		16		16		26	20
Load Global (LDG)	1		1		2		2	
Store Global (STG)	1		1		1		1	

7-point stencil Mojo vs. CUDA NCU profiling metrics

Nsight Compute CLI (ncu) metric	Double Precision L=512 (512×1×1) Mojo	Double Precision L=512 (512×1×1) CUDA	Single Precision L=1024 (1024×1×1) Mojo	Single Precision L=1024 (1024×1×1) CUDA
Duration (ms)	1.10	0.96	8.74	7.21
Throughputs (%)				
- Compute SM	81.41	51.96	79.8	53.7
- Memory	67.98	76.72	37.7	43.9
L1 ai (FLOP/byte)	0.14		0.24	
L2 ai (FLOP/byte)	0.26		0.51	
L3 ai (FLOP/byte)	0.62		1.24	
L1-3 Perf (FLOP/s)	1.20 E12	1.38 E12	1.22 E12	1.48 E12
Registers	24	21	26	20
Load Global (LDG)	7		7	
Store Global (STG)	1		1	