The Journal of Open Source Software

graph_framework: A Domain Specific Compiler for Building Physics Applications

M. Cianciosa [1], **D. Batchelor** [2], **and W. Elwasif** [1]

1 Oak Ridge National Laboratory **2** Diditco, Oak Ridge TN 37831

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 03 December 2025
Published: unpublished

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

DRAFT

Summary[1]

The `graph_framework` is a domain specific compiler which enables domain scientists to create optimized kernels that can operate on Graphics Processing Units (GPUs) or central processing unit (CPUs). This framework works by first building data structures of the operations making up a physics equations. Algebraic simplifications are applied to the graphs to reduce them to simpler forms. Auto differentiation is supported by traversing existing graphs and creating new graphs by applying the chain rule. These graphs can be Just-In-Time (JIT) compiled to central processing unit (CPUs), Apple GPUs, NVidia GPUs, and initial support for AMD GPUs.

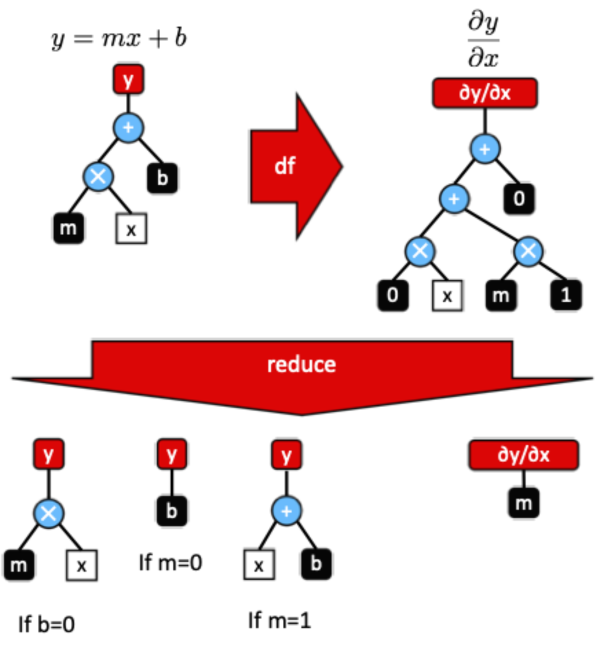

Figure 1: Mathematical operations are defined as a tree of operations. A df method transforms the tree by applying the derivative chain rule to each node. A reduce method applies algebraic rules removing nodes from the graph.

[1]Notice of Copyright This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paidup, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

This framework focuses on the domain of physics problems where a the same physics is being applied to large ensemble of particles or rays. Applications have been developed for tracing large numbers of Radio Frequency (RF) rays in fusion devices and particle tracing for understanding how particles distributions are lost or evolve over time. The exploitation of GPU resources afforded by this framework allows high fidelity simulations at low computational cost.

Statement of need

Modern supercomputers are increasingly relying on Graphic Processing Units (GPUs) and other accelerators to achieve exa-scale performance at reasonable energy usage. A major challenge of exploiting these accelerators is the incompatibility between different vendors. A scientific code written using CUDA will not operate on a AMD gpu. Frameworks that can abstract the physics from the accelerator kernel code are needed to exploit the current and future hardware. In the world of machine learning, several auto differentiation frameworks have been developed that have the promise of abstracting the math from the compute hardware. However in practice, these framework often lag in supporting non-CUDA platforms. Their reliance on python makes them challenging to embed within non python based applications.

Fusion energy is a grand engineering challenge to make into a viable power source. Beyond the technical challenges towards making it work in the first place, there is an economic challenge that it needs to be addressed. For fusion energy to be competitive in the market place. Addressing the economic challenge is tackled though design optimization. However, a barrier to optimization is the computational costs associated with exploring the different configurations.

Low fidelity models like systems codes(Kovari et al., 2014),(Kovari et al., 2016), can miss critical physics that enable optimized designs. High fidelity models, are too costly to run for multiple configurations. GPUs offer tremendous processing power that is largely untapped in codes developed by domain scientists. Due to the challenges of exploiting GPUs they are largely relegated to hero class codes which can use a large percentage of Exa-scale machines.

However, there is an intermediate scale of problems which individually can operate using modest computational requirements but become a challenge when generating large ensembles. These codes are typically CPU only due to the challenges of adopting GPUs. As more super computers are diminishing CPU capacity in favor of GPU support, we are losing the capacity computing necessary to explore large ensembles necessary for device optimization.

The goal of the graph_framework is to lower the barrier of entry for adopting GPU code. While there are many different solutions to the problem of performance portable code, different solutions have different drawbacks or trade offs. With that in mind the graph_framework was developed to address the specific capabilities of:

- Transparently support multiple CPUs and GPUs including Apple GPUs.
- Use an API that is as simple as writing equations.
- Allow easy embedding in legacy code (Doesn't rely on python).
- Enables automatic differentiation.

With these design goals in mind this framework is limited to the classes of problems which the same physics is applied to a large ensemble of particles. This limitation simplifies the complexity of this framework making future extensibility simpler as a need arises for a new problem domain. In this paper will describe the frameworks design and capabilities. Demonstrate applications to problems in radio frequency (RF) heating and particle tracing, and show its performance scaling.

Background

DRAFT

Table 1: Overview of GPU capable frameworks.

Framework	Language	Cuda Support	Metal Support	RocM Support	Auto Differentiation
graph_framework	C++, C, Fortran	Official	Official	Preliminary	Yes
Cuda	C	Official	None	None	No
Metal	Objective C, Swift	None	Official	Depreciated	No
Kokkos	C++	Official	None	Official	No
OpenACC	C, C++, Fortran	Official	None	None	No
OpenMP	C, C++, Fortran	Compiler Dependent	None	Compiler Dependent	No
OpenCL	C	Official	Deprecated	Official	No
Vulcan	C	Official	Unofficial	Official	No
HIP	C	Official	None	Official	No
TensorFlow	Python, C++	Official	Unofficial/Incomplete	Unofficial	Yes
JAX	Python	Official	Unofficial/Incomplete	Official	Yes
PyTorch	Python, C++, Java	Official	Official	Official	Yes
mlx	Python, C++, Swift	Official	Official	Experimental	Yes

Standardized programming languages such as Fortran(Backus & Heising, 1964), C(Ritchie, 1993), C++(Stroustrup, 2013), have simplified the development of cross platform programs. Scientific codes have relied on the ability to write source code which can operate on multiple processor architectures and operating systems (OSs) with no or little changes given an appropriate compiler. However, modern super computers rely on graphical processing units (GPUs) to achieve exa-scale performance(Hines, 2018),(Yang & Deslippe, 2020),(Schneider, 2022) with reasonable energy usage. Unlike central processing units (CPUs), the instruction sets of GPUs are proprietary information. Additionally, since accelerators typically are hardware accessories, an OS requires device drivers which are also proprietary. NVidia GPUs are best programmed using CUDA(*Cuda Documentation*, n.d.) while Apple GPUs use Metal(*Metal Documentation*, n.d.) and AMD GPUs use HIP(*Hip Documentation*, n.d.).

There are many potential solutions to cross performance portable support. Low level cross platform frameworks general purpose GPU (GPGPU) programming frameworks such as OpenCL(Munshi et al., 2011) and Vulkan(*Vulkan Specification*, n.d.) requires direct vendor support. HIP can support NVidia GPUs by abstracting the driver API and rewriting kernel code. However these frameworks are the lowest level and require GPU programming expertise to utilize them effectively that a domain scientist may not have. A higher level approach used in OpenACC(Farber, 2016) and OpenMP(*OpenMP Specification*, n.d.) use source code annotation to transform loops and code blocks into GPU kernels. The drawback of this approach is that source code written for CPUs can result in poor GPU performance. Kokkos(Edwards et al., 2011) is a collection of performance portable array operations for building device agnostic applications. However, the framework only support AMD and Nvidia GPUs and doesn't have out of box support for auto differentiation.

With the advent of Machine learning, several machine learning frameworks have been created such as TensorFlow(Abadi et al., 2015), JAX(Bradbury et al., 2018), PyTorch(Paszke et al., 2017), and MLX(Hannun et al., 2023). These frameworks build a graph representation operations that can be auto-differentiated and compiled to GPUs. These frameworks are intended to be used through a python interface which lowers one barrier to using but also introduces new barriers. For instance, it's not straight forward to embed these frameworks in non-python codes and their non-python API's don't always support all the features or are as well documented as their python API's. Additionally performance is not guaranteed. It is not always straight forward to understand what the framework is doing. Additionally cross platform support is often unofficial and can be incomplete. Table 1 shows an overview of these frameworks.

Performance

To demonstrate the performance of the optimized kernels created using this framework we measured the strong scaling using the the RF ray tracing problem in a realistic tokamak geometry. To to compare against other frameworks we benchmarked the achieved throughput for simulating gyro motion in a uniform magnetic field.

Strong Scaling

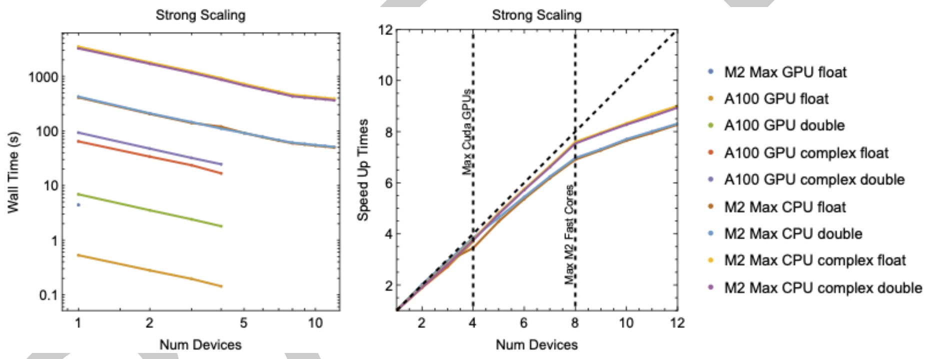

Figure 2: Left: Strong scaling wall time for 100000 Rays traced in a realistic tokamak equilibrium. Right: Strong scaling speedup normalized to the wall time for a single device or core. The dashed diagonal line references the best possible scaling. The M2 Max has 8 fast performance cores and 4 slower energy efficiency cores resulting drop off in improvement beyond 8 cores.

To benchmark code performance we traced 10^6 rays for 10^3 time steps using the cold plasma dispersion relation in a realistic tokamak equilibrium. A benchmarking application is available in the git repository. The figure above shows the strong scaling of wall time as the number of GPU and CPU devices are increased. The figure above shows the strong scaling speed up

$$SpeedUp = \frac{time\,(1)}{time\,(n)}$$

Benchmarking was prepared on two different setups. The first set up as a Mac Studio with an Apple M2 Max chip. The M2 chip contains a 12 core CPU where 8 cores are faster performance codes and the remaining 4 are slower efficiency cores. The M2 Max also contains a single 38-core GPU which only support single precision operations. The second setup is a server with 4 Nvidia A100 GPUs. Benchmarking measures the time to trace 10^6 rays but does not include the setup and JIT times.

Figure 2 shows the advantage even a single GPU has over CPU execution. In single precision, the M2's GPU is almost $100\times$ faster than single CPU core while the a single A100 has a nearly $800\times$ advantage. An interesting thing to note is the M2 Max CPU show no advantage between single and double precision execution.

DRAFT

For large problem sizes the framework is expected to show good scaling with number of devices as the problems we are applying are embarrassingly parallel in nature. The figure above shows the strong scaling speed up with the number of devices. The framework shows good strong scaling as the problem is split among more devices. The architecture of the M2 Chip contains 8 fast performance cores and 4 slower energy efficiency cores. This produces a noticeable knee in the scaling after 8 core are used. Overall, the framework demonstrates good scaling across CPU and GPU devices.

Comparison to other frameworks

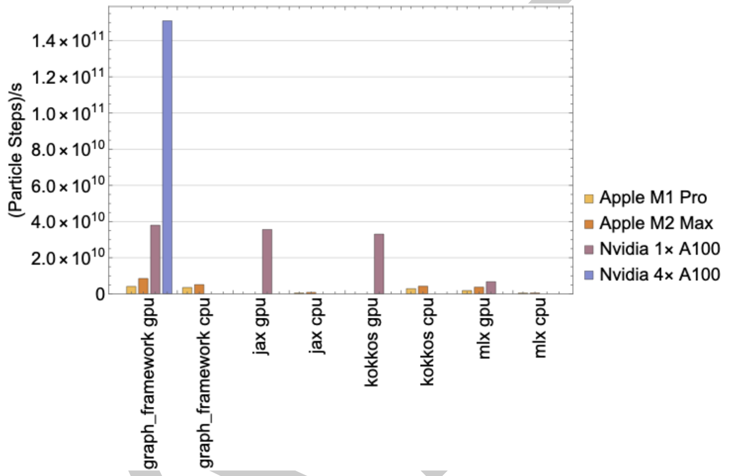

Figure 3: Particle throughput for graph framework compared to MLX and JAX.

To benchmark against other frameworks we will look at the simple case of gyro motion in a uniform magnetic field $\vec{B}=B_0\hat{z}$.

$$\frac{\partial\vec{v}}{\partial t}=dt\vec{v}\times\vec{B}$$
$$\frac{\partial\vec{x}}{\partial t}=dt\vec{v}$$

We compared the graph framework against the MLX framework since it supports Apple GPUs and JAX due to it's popularity. Source codes for this benchmark case is available in the `graph_framework` documentation. Figure 3 shows the throughput of pushing 10^8 particles for 10^3 time steps. The `graph_framework` consistently shows the best throughput on both CPUs and GPUs. Note MLX CPU throughput could by improved by splitting the problem to multiple threads.

DRAFT

Acknowledgements

The authors would like to thank Dr. Yashika Ghai, Dr. Rhea Barnett, and Dr. David Green for their valuable insights when setting up test cases for the RF-Ray Tracing.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* https://www.tensorflow.org/

Backus, J. W., & Heising, W. P. (1964). Fortran. *IEEE Transactions on Electronic Computers*, *EC-13*(4), 382–385. https://doi.org/10.1109/PGEC.1964.263818

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Cuda documentation. (n.d.). https://docs.nvidia.com/cuda/.

Edwards, H. C., Sunderland, D., Amsler, C., & Mish, S. (2011). Multicore/GPGPU portable computational kernels via multidimensional arrays. *2011 IEEE International Conference on Cluster Computing*, 363–370. https://doi.org/10.1109/CLUSTER.2011.47

Farber, R. (2016). *Parallel programming with OpenACC*. Newnes.

Hannun, A., Digani, J., Katharopoulos, A., & Collobert, R. (2023). *MLX: Efficient and flexible machine learning on apple silicon* (Version 0.0). https://github.com/ml-explore

Hines, J. (2018). Stepping up to summit. *Computing in Science & Engineering*, *20*(2), 78–82. https://doi.org/10.1109/MCSE.2018.021651341

Hip documentation. (n.d.). https://rocm.docs.amd.com/projects/HIP/en/latest/index.html.

Kovari, M., Fox, F., Harrington, C., Kembleton, R., Knight, P., Lux, H., & Morris, J. (2016). "PROCESS": A systems code for fusion power plants – part 2: engineering. *Fusion Engineering and Design*, *104*, 9–20. https://doi.org/https://doi.org/10.1016/j.fusengdes.2016.01.007

Kovari, M., Kemp, R., Lux, H., Knight, P., Morris, J., & Ward, D. J. (2014). "PROCESS": A systems code for fusion power plants—part 1: physics. *Fusion Engineering and Design*, *89*(12), 3054–3069. https://doi.org/https://doi.org/10.1016/j.fusengdes.2014.09.018

Metal documentation. (n.d.). https://developer.apple.com/metal/.

Munshi, A., Gaster, B., Mattson, T. G., & Ginsburg, D. (2011). *OpenCL programming guide*. Pearson Education.

OpenMP specification. (n.d.). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). *Automatic differentiation in PyTorch*.

Ritchie, D. M. (1993). The development of the c language. *ACM Sigplan Notices*, *28*(3), 201–208.

Schneider, D. (2022). The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum*, *59*(1), 34–35. https://doi.org/10.1109/MSPEC.2022.9676353

Stroustrup, B. (2013). *The c++ programming language*. Pearson Education.

Vulkan specification. (n.d.). https://docs.vulkan.org/spec/latest/index.html.

Yang, C., & Deslippe, J. (2020). Accelerate science on perlmutter with NERSC. *Bulletin of the American Physical Society*, *65*.

DRAFT