I work towards a future where high performance computing systems are broadly accessible, where non-experts can easily leverage high performance computers for data analytics, scientific computing and machine learning, and computing experts can quickly translate new optimizations into fast code. Modern computing systems are rapidly growing in complexity in both the architecture of individual processors to their aggregation as distributed systems. These multiple fronts of increasing complexity result in challenges in both developing applications that run on individual compute elements and scaling those applications onto distributed clusters. As a result, most programmers are either unable to target modern high performance systems, or achieve only a small fraction of these systems' promised capabilities.

I build efficient and productive programming systems for high-performance computing, enabling experts and non-experts alike to leverage modern distributed and heterogeneous machines. We have built a multi-layered software stack that enables non-expert users to compose high performance components and expert users to productively build efficient implementations of those components. We have developed programming models [1, 2] and optimizations [3, 4] for distributed runtime systems that allow for independently written distributed libraries to compose with high performance. This work has been implemented within NVIDIA's Legate system, which has enabled, for the first time, an ecosystem of high-level libraries from different domains that users can mix and match to efficiently target distributed machines. We implemented kernels for Legate libraries with the DISTAL [5, 6] compiler, which generates distributed and accelerated dense and sparse tensor algebra computations from a compact, high-level language. My latest research targets kernel development for modern accelerators, which are rapidly changing to meet the computational demands of artificial intelligence. We proposed Cypress [7], a language for programming the asynchronous accelerators in modern GPUs without exposing concurrency and data movement, enabling optimization and eliminating entire classes of bugs.

As a professor, I aim to tackle the key problems of portability and productivity in programming high performance computing systems. High performance computers continue to scale up and specialize further, consistently requiring bespoke software tuned for each new hardware vendor and architectural generation. I am interested in developing programming systems that allow programmers to develop correct, maintainable software for these complicated machines that does not need to be rewritten for every new architectural jump. I envision the core of these programming systems to be abstractions that enable reasoning over complex architectural features, and the combination of compilers and runtime systems to manage different layers of the hierarchical hardware stack.

Composing Distributed and Heterogeneous Software

My first line of work focuses on programming modern *supercomputers*, which are distributed (multiple nodes) and heterogeneous (each node has multiple kinds of processors, such as CPUs and GPUs). Programming supercomputers today is an especially difficult task because it requires broad expertise of both low-level kernels that run on individual accelerators to high-level algorithms that minimize and overlap communication across a hierarchical network.

Problem. Computer scientists hide complexity through abstraction and composition of modular software; experts develop complex implementations and expose simple interfaces for non-experts. While distributed libraries exist for specific domains [8, 9, 10, 11], composing multiple independent distributed libraries while maintaining correctness and high performance is onerous. The burden of achieving both goals falls on the (likely non-expert) end user performing the composition. The root of this difficulty is the lack of proper abstractions for distributed data and computation; without them, distributed libraries make independent decisions (losing performance opportunities) and require users to negotiate distributed data at library boundaries. Without efficient composition, an extensible ecosystem of compatible, but domain-specific, libraries cannot exist, siloing high performance computing into narrow and isolated domains.

Approach. Our approach to distributed composition is a combination of compilers and runtime systems that coordinate and co-optimize across library boundaries. We develop abstractions for distributed data [1] and distributed computation [2] that allow libraries to flexibly define distributed computations and partitions of distributed data needed by those computations. At the core of these abstractions sit task-based programs with a distributed data model to reason about computations running on individual accelerators and the data those accelerators require. These abstractions allow our runtime systems and compilers to ensure correctness (by understanding how distributed data flows across library boundaries) and optimize performance (by selecting strategies that are efficient within the context a library is used). Our work enables developers to write independent libraries that when composed, automatically overlap computation and communication, fuse dependent computations, and minimize data movement, all across library boundaries. Our strategy centers around late-binding performance-critical decisions (what distributed data layout to use, which computational kernels to run) and just-in-time (JIT) analysis and code generation; therefore, controlling the overheads imposed by these strategies is performance-critical. We therefore architect our runtime systems like JIT compilers that automatically identify repeatedly executed traces through the application (that may span multiple libraries) and memoize analysis for these traces [3]. Finally, we developed techniques to execute task-based programs with overheads comparable to bare-metal systems like MPI, through a deep connection between actor-based and task-based programming models [4]. Taken all together, our work provides an end-to-end framework for building high-level, independent distributed libraries,

and then running programs built from these libraries as fast as codes tuned directly for the target machine and workload.

Impact. Our work has become an integral part of NVIDIA's recent Legate ecosystem, which is a production runtime system for building compositional distributed and heterogeneous libraries. I developed Legate Sparse [1], a distributed drop-in replacement for SciPy Sparse, which is one of the many Legate libraries available today as NVIDIA products; this set includes cuPyNumeric [12], a distributed NumPy library, as well as gradient boosting, dataframe and ML libraries. Other components of this research are either implemented directly in production Legate and its underlying system Legion [13], or are road-map features for the NVIDIA engineering team. Legate is already being used by different groups across academia and industry to scale workloads to clusters of GPUs and enable new science.

Domain-Specific Languages for Distributed Tensor Computations

Legate proposes several general-purpose techniques for composing computations on supercomputers. This line of work instead focuses on the specific problem of mapping dense and sparse tensor computations to distributed machines, which is an important class of workloads in scientific computing and artificial intelligence.

Problem. Developing high performance distributed tensor algebra computations intertwines choices of data and compute partitioning, communication strategy, data structures and low-level compute kernels. While an expert may have different strategies for a target computation, implementing and optimizing each one can require hundreds or thousands of lines of low-level code. Additionally, due to the interleaving of the many decisions described above, many existing libraries support only a subset of the space of tensor algebra expressions, data structures and distribution strategies. When a user requires a computation not present in the library, or has distributed data in an unsupported orientation, they must either write new kernels or pay significant overhead to fit their use case into what the library supports.

Approach. We developed DISTAL [5, 6], a DSL for dense and sparse tensor algebra that targets distributed and heterogeneous machines. The key insight of DISTAL is the separation of the tangled components involved in developing distributed tensor algebra programs: users independently specify the target tensor computation, data structures, distributed computation strategy and distributed data layout. DISTAL then weaves these independent specifications into an efficient distributed implementation. We show that many existing algorithms for distributed dense and sparse tensor computations can be expressed concisely through the composition of specifications for computation distribution, data distribution and data layout within DISTAL. Similarly to Legate, the analysis required to realize DISTAL programs is made possible through a combination of compilers and runtime systems, dispatching different portions of the analyses to the component with the right information. For example, DISTAL leverages a hybrid static-dynamic analysis for sparse tensor computations, where DISTAL statically generates code according to the shapes of sparse data structures in the computation and dynamically resolves the data-dependent indirections of the sparse data structures themselves.

Impact. DISTAL enables rapid exploration of the design space of distributed tensor computations through compact and logically separate languages for functional specification and optimization. DISTAL was used to implement the performance-critical set of kernels in Legate Sparse [1], enabling the library to be developed more quickly than if every kernel had been written by hand. Intellectually, DISTAL's scheduling and data distribution languages exposed the underlying structure in describing and optimizing distributed tensor computations.

Programming Systems For Emerging Hardware

My recent focus is on programming emerging accelerator architectures, which are growing increasingly powerful to keep pace with the compute demands of artificial intelligence. These accelerators come with increasingly complex interfaces and semantics, which makes each new generation more difficult to program than the previous. A concrete example is NVIDIA Hopper and Blackwell GPUs; these GPUs have transitioned to exposing asynchronous matrix multiplication and data movement units that must be managed by software to achieve peak performance.

Problem. Asynchronous compute and data movement operations add a new kind of difficulty to the already complex task of developing high performance linear algebra computations on GPUs. To utilize these asynchronous units, kernel implementations transition from a classic bulk-synchronous style to a task-parallel style, where asynchronous pipelines negotiate through shared queues and concurrent communication mechanisms. Achieving both correctness and high performance for these concurrent programs is extremely difficult even for experts. High-level languages like Triton [14] attempt to shield programmers from this complexity, but restrict control over low-level optimization decisions. As a result, programmers have no recourse to improve the often suboptimal decisions made by compilers.

Approach. We developed Cypress [7], a DSL and compiler that enables high performance linear algebra computations for modern GPUs to be expressed with *sequential semantics*. Cypress programs are organized as a collection of tasks that describe the data they read from and write to. Tasks may recursively launch other tasks, and appear to the programmer as executing sequentially. Optimization decisions for Cypress programs are expressed through a separate *mapping specification* that externalizes choices like tile shapes and memory placement for program data. Cypress combines the task-based functional specification with the mapping to generate code with asynchronous computation

and data movement, and automatically overlaps independent computations and communication. Cypress automatically constructs asynchronous pipelines and extracts parallelism from the sequentially specified source program, removing these burdens from the programmer and eliminating entire classes of bugs. At the same time, Cypress enables precise control over important optimization decisions, such as the decomposition of data and computation, which allows Cypress programs to achieve performance competitive with codes hand-tuned by professional engineers.

Impact. Cypress showed that new complexity in accelerators can be alleviated by advances in programming models, and that pushing the burden of asynchrony and concurrency onto the programmer is not necessary for peak performance. Cypress also demonstrates how a compiler for linear algebra on GPUs that treats asynchrony as a first-class principle might be organized, instead of as a concept bolted on to the side of an existing compilation stack.

Where To Next?

I plan to develop programming models, compilers and runtime systems for the explosion of new architectures being developed by academia and industry; advances in program representation and analysis will be required to keep up and unlock new architectures' promised capabilities. I am also interested in collaborations with application domains such as scientific computing and machine learning, as co-design of algorithms and systems brings out the best in both.

Programming Systems for Emerging Hardware. I am currently exploring and am interested in pursuing further the problem of portability of high performance programs onto new and diverse architectures. As architecture becomes more powerful, new algorithms must be discovered and software must be re-optimized to take advantage of new architectural capabilities. A concrete example is the year long gap between Flash Attention 2 [15] and Flash Attention 3 [16] — Flash Attention 3 is a version of Flash Attention optimized to take advantage of software-exposed asynchrony in the NVIDIA Hopper GPU. *Why didn't our compilers automatically generate this new algorithm?* I believe our current programming systems can be good at optimization within a fixed set of axes, but are often unable to adapt when hardware evolves to expose new axes (like Hopper's software-exposed asynchrony). Solving this problem will require us to both build compilers in a way that the existing axes are general enough to capture changes in hardware, and to potentially build compiler-generators that can discover optimizations for new axes when they arrive.

The physical constraints of latency and energy will require hardware to continue reaching for specialization, asynchrony and hierarchy to increase performance. Designs in these directions can be seen in recent dataflow [17], distributed-memory [18] and fine-grained parallel [19] architectures. I believe future programming systems will need to adapt to match the shape of programs closer to the structure of these specialized architectures. Additionally, future programming systems will need to have declarative control over core operations like data movement and synchronization (like shown in the Legate and Cypress programming models) so that compilers have enough freedom to optimize for new hardware.

Optimization Across the Software and Hardware Stack. To continue improving performance on modern clusters, optimizations that cross several boundaries of the stack (from high-level algorithmic choices to low-level kernel implementations) are becoming increasingly common. For example, distributed matrix-multiplication kernels used in large language model serving fuse collective communication within the inner multiply-accumulate loop of GEMM kernels, taking advantage of new networking capabilities. Developing applications of this level of complexity teeters at the edge of our cognitive abilities and is currently limited to the tiny subset of developers with the requisite knowledge. I believe that programming systems for the productive expression of programs that cross many layers of the stack in this manner require hierarchical abstractions that can separately describe the requisite optimizations at different layers of the machine. A hierarchical description allows programmers to encapsulate logically separated layers of the computation, and enables programming systems to perform holistic analyses that can optimize multiple layers simultaneously. An example of such a model was proposed in DISTAL, where computations could be scheduled hierarchically, using different distributed algorithms at the cluster-level than across the multiple GPUs within a node. Alternatively, a model like Cypress that decomposes computations from the GPU as a whole down to individual threads could be extended upwards, where computations are first broken down across clusters of GPUs.

Blurring the Lines Between Compilers, Runtime Systems and Architecture. A key approach of my research so far is a collaboration between compilers and runtime systems for both optimization and the tractability of analysis. As computer architecture continues to evolve, the standard delineations between the domain of the compiler and the runtime system are increasingly blurred. When components of architecture get more specialized, compilers are increasingly responsible for decisions like parallelism extraction and scheduling. At the same time, increased sophistication in the architecture can provide new capabilities exploitable by runtime systems and obviate the need for complex static analyses. I am interested in exploring these changing boundaries and revisiting system designs that we take for granted today; moving transformations across these boundaries opens the door to previously unattainable performance or the description of previously inexpressible computations. An example of such a system could be a hybrid model between the distributed Legate programming model and Cypress; this unified system could leverage runtime analysis for flexible scheduling and communication at the multi-node level like Legate, but drop into a static analysis for kernels within a single GPU like Cypress, where overheads are unacceptable.

Referenced Work

[1] **Rohan Yadav**, Wonchan Lee, Melih Elibol, Manolis Papadakis, Taylor Lee-Patti, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. "Legate Sparse: Distributed Sparse Computing in Python". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23. Denver, CO, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: 10.1145/3581784.3607033. URL: https://doi.org/10.1145/3581784.3607033.

[2] **Rohan Yadav**, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. "Composing Distributed Computations Through Task and Kernel Fusion". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 182–197. ISBN: 9798400706981. DOI: 10.1145/3669940.3707216. URL: https://doi.org/10.1145/3669940.3707216.

[3] **Rohan Yadav**, Michael Bauer, David Broman, Michael Garland, Alex Aiken, and Fredrik Kjolstad. "Automatic Tracing in Task-Based Runtime Systems". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 84–99. ISBN: 9798400706981. DOI: 10.1145/3669940.3707237. URL: https://doi.org/10.1145/3669940.3707237.

[4] **Rohan Yadav**, Joseph Guman, Sean Treichler, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. *On the Duality of Task and Actor Programming Models*. 2025. arXiv: 2508.16522 [cs.PL]. URL: https://arxiv.org/abs/2508.16522.

[5] **Rohan Yadav**, Alex Aiken, and Fredrik Kjolstad. "DISTAL: The Distributed Tensor Algebra Compiler". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 286–300. ISBN: 9781450392655. DOI: 10.1145/3519939.3523437. URL: https://doi.org/10.1145/3519939.3523437.

[6] **Rohan Yadav**, Alex Aiken, and Fredrik Kjolstad. "SpDISTAL: Compiling Distributed Sparse Tensor Computations". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022.

[7] **Rohan Yadav**, Michael Garland, Alex Aiken, and Michael Bauer. "Task-Based Tensor Computations on Modern GPUs". In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729262. URL: https://doi.org/10.1145/3729262.

External References

[8] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, et al. *PETSc/TAO Users Manual Revision 3.23*. Tech. rep. Argonne National Laboratory (ANL), Argonne, IL (United States), Mar. 2025. DOI: 10.2172/2565610. URL: https://www.osti.gov/biblio/2565610.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. "ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance". In: *Computer Physics Communications* 97.1 (1996). High-Performance Computing in Science, pp. 1–15. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(96)00017-3. URL: https://www.sciencedirect.com/science/article/pii/0010465596000173.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[11] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[12] Michael Bauer and Michael Garland. "Legate NumPy: accelerated and distributed array computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356175. URL: https://doi.org/10.1145/3295500.3356175.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence With Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[14] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.

[15] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. 2023. arXiv: 2307.08691 [cs.LG]. URL: https://arxiv.org/abs/2307.08691.

[16] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608 [cs.LG]. URL: https://arxiv.org/abs/2407.08608.

[17] Raghu Prabhakar et al. "SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Nov. 2024, pp. 1353–1366. DOI: 10.1109/micro61859.2024.00100. URL: http://dx.doi.org/10.1109/MICRO61859.2024.00100.

[18] Axel Feldmann, Courtney Golden, Yifan Yang, Joel S. Emer, and Daniel Sanchez. "Azul: An Accelerator for Sparse Iterative Solvers Leveraging Distributed On-Chip Memory". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2024, pp. 643–656. DOI: 10.1109/MICRO61859.2024.00054.

[19] Andronicus Rajasukumar, Jiya Su, Yuqing, Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. *UpDown: Programmable fine-grained Events for Scalable Performance on Irregular Applications*. 2024. arXiv: 2407.20773 [cs.AR]. URL: https://arxiv.org/abs/2407.20773.