Kirshanthan Sundararajah
Research Statement

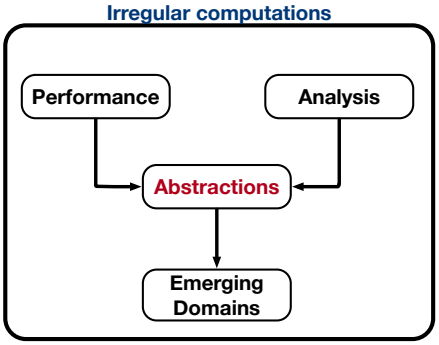
Hardware miniaturization has been at the core of performance gain for many decades. Initially, it was foreseen by physicist Richard Feynman in his 1959 address to the American Physical Society, "*There's Plenty of Room at the Bottom.*" Until recent years, improvements in hardware guaranteed an exponential increase in computing performance – a trend known as Moore's Law. This trend of free performance benefits with every hardware upgrade is dying out as we approach the fundamental physical limitations of transistors. Running out of this *free lunch* seems to be unpromising. Nevertheless, the 2020 Science Journal article by Leiserson *et al.* [1] suggests that "*There's Plenty of Room at the Top*," and performance engineering of software and development of algorithms are ever more critical to deliver faster applications in the post-Moore's law era.

During the Moore's law era, software development was generally focused on ease of implementation rather than efficiency and scalability of execution because hardware upgrades guaranteed performance gain. Concentrating on ease of development led to massive inefficiencies in programs, commonly known as *software bloat*. In the post-Moore's law era, performance transparency — reasoning about the performance of execution merely from code — has been lost. The burden of ensuring performance gain has now fallen in the hands of software developers. A general approach to handle this challenge is **Performance Transparent Abstractions (PTAs)** — *abstractions that allow us to reasonably predict and guarantee performance at the software level*. I believe that we can find principled solutions to this problem by introducing PTAs in programming languages and compilers, and specializing these abstractions to classes of computation, programming models, or architectures is essential for ensuring effectiveness and efficiency of solutions to performance challenges.

PTAs for *regular computations* — programs that operate on regularly structured data such as dense arrays using loops with statically predictable control-flow — is a mature field of research. For a long time, computer systems have been designed for regular computations and these computations appear in many critical applications such as image processing pipelines, PDE solvers, and machine learning workloads. On the other hand, *irregular computations* — programs that operate on irregularly structured data, such as sparse tensors, trees, and graphs using recursion with statically difficult to predictable control-flow — *increasingly* appear in essential applications such as physical simulations, data mining, and graphics rendering. There are fewer research efforts on *principled and general approaches* to design, analyze, and accelerate irregular computations compared to regular computations, partly due to lack of PTAs.

Research Goals

My research goals are centered around filling the "*room at the top*" (*i.e., developing solutions at the software level to guarantee performance improvements.*) in a principled way with a focus on *irregularity*. There are two major factors make my goals challenging. First, irregular computations appear in applications *intermixed* with regular computations. Second, the lack of PTAs to seamlessly handle the irregular computations. Isolated approaches provide ad-hoc solutions, which may solve a specific problem without the capacity for generalizing. Hence, it is paramount to provide general and extensible solutions. I believe that generalization can be achieved by pursuing the following with a focus on irregularity.



Abstractions for Performance: Capturing irregular computations with enough additional information is critical for optimizing their performance. Transformation of computations (*e.g.,* restructuring) is one of the ways to gain performance. Designing abstractions for transformations help us achieve generality and performance.

Abstractions for Analysis: Capturing irregular computations is not sufficient if we cannot analyze various ways for them to be transformed *correctly*. Abstractions for analyzes should be cleverly designed for *composability* — ability to use multiple analyzes in conjunction — and *decidability* — existence of effective procedures to verify desired properties (*e.g.,* preservation of dependences).

Incorporating these abstractions for performance and analysis in software stacks (*i.e.,* programming language, compiler, runtime systems, etc.) of **Emerging Domains** has strong impact on the performance of irregular computations.

Abstractions for Irregular Computations

My prior work is centered around this research philosophy for irregular computations, especially computations with control-flow irregularity (*e.g.,* recursion). The core part of my research philosophy relies on designing abstractions, and these abstractions are implemented on compiler pipelines.

Abstractions for Performance. A standard technique for optimizing the locality of programs with regular loop nests is multi-level tiling — splitting loops such that the data accessed by these loops fit in caches — which requires knowledge of the cache sizes for better performance (*i.e., Cache-Aware*). An alternative *Cache-Oblivious* strategy is recursively subdividing

the iteration space (*i.e.,* divide and conquer) to make parts of data fit in caches [2]. My **ASPLOS '17** work [3] introduced *Recursion Twisting*, a transformation that can be applied to computations captured with the pattern of *Nested Recursion* to generalize cache-oblivious multi-level tiling. In nested recursion, one recursive function is nested inside another: consider the scenario of traversing a tree, and for each node of the tree, traversing a second, inner tree, as in tree-join algorithms. In this case, traversing the smaller tree second has better locality, as the smaller tree is more likely to fit in the cache and accessed repeatedly. The key insight of recursion twisting is that after partially traversing the outer tree, it may now be smaller than the inner tree. Hence, swapping the order of recursion (an analog of loop interchange) provides better locality by placing the smaller tree on the inside. Continuing this swapping process will naturally lead to smaller trees fitting in cache, and improves locality in a cache-oblivious manner. The challenge here is ensuring the swapping does not violate any dependences. We applied recursion twisting to *Dual-Tree* implementations of *Generalized N-body Problems* from Curtin *et al.* [4], and demonstrated an in order-of-magnitude performance enhancement.

Trees are pervasive data structures in numerous critical applications and tree traversal is one of the main source of irregular computations. There are associated optimizations for a specific class of tree traversals, but applicability of those optimizations to a different class is not catalogued well. Our **ISPASS '17** work [5] introduced Treelogy, a benchmark suite for tree traversals. Treelogy differs from other benchmark suites by providing an abstraction in the form of an ontology to match tree traversal algorithms to optimizations based on the structural properties of tree traversals.

Although PTAs are built into compilers, compilers themselves can be the source of irregularity. For instance, irregularity in control-flow graph has implications on performance, code size, and testing of the generated code. Our **CGO '22** work developed a compiler transformation, *Control-Flow Melding (CFM)*, for reducing irregularity in control-flow graphs by fusing similar regions. CFM is part of DARM [6], a compiler aimed at reducing divergence in GPGPUs. GPGPUs use the Single-Instruction-Multiple-Thread (SIMT) execution model, where a group of threads execute instructions in lockstep and lose performance when execution diverges at branches. DARM can meld divergent control-flow structures with similar instruction sequences to reduce performance degradation and stands out from classical approaches, since it handles arbitrary control-flow structures.

Abstractions for Analysis. In the world of nested loop based programs, a number of frameworks such as the *Polyhedral Model* are capable of reasoning about the correctness of transformations. Proving that a transformation is safe requires providing tests to ensure the preservation of dependences in the original program. Often complex transformations for these programs are compositions of multiple smaller ones. It is important to build abstractions that help us reason about a composed sequence of transformations since transformations that are individually unsafe may be part of a safe sequence of composed transformations. In my **PLDI '19** work [7], we introduced PolyRec, a framework to perform scheduling transformations for nested recursive iteration spaces. Nested recursion is a broad pattern to capture the mix of recursion and loops, since any loop can be treated as a tail recursion. The PolyRec framework represents the iteration space of perfectly nested recursive program as a multitape automaton (*i.e.,* non-deterministic finite state automaton with multiple input tapes) and the scheduling transformations as multitape transducers. These abstractions pave the way to analyze composed transformations. Verifying the correctness of these transformations requires representing the dependence of the program and an algorithm to check their preservation. We introduced *Witness Tuples* to represent dependences, an abstraction similar to, but more expressive than the distance vector from prior loop transformation frameworks. The crux of PolyRec is a decidable algorithm for checking the preservation of dependence (*i.e.,* dependence check). PolyRec provides composable irregular program transformations covering all the analogues loop transformations except skewing.

In loop transformation frameworks such as the *Unimodular Model*, loop skewing is a powerful scheduling transformation that reveals many parallelization opportunities for regular loop nests when composed with loop interchange, and loop reversal. My **OOPSLA '22** work [8] UniRec is focused on generalizing the PolyRec framework to include scheduling transformations for nested recursive iteration spaces that ultimately expose parallelism. Although PolyRec handles interchange and reversal for nested recursive iteration spaces, its representation is not powerful enough to address skewing. UniRec introduces the notion of skewing to recursive iteration spaces and improves PolyRec's representation to handle the composition of skewing with other transformations. The key idea of UniRec is extending the multitape string representation of iteration space to perform integer arithmetic and ordering, while transformations considered in PolyRec only requires the later. Finally, we prove that we can use PolyRec's correctness checking algorithm in UniRec with minimal changes. UniRec fully subsumes the unimodular framework since nested recursion is more general than nested loops.

Resolving dependences is an essential prerequisite to analyze, verify and perform scheduling transformations. Our **OOPSLA '17** work TreeFuser [9] explored automatically fusing general recursive traversals on the same tree for reducing number of traversals, increasing opportunities for other optimization, and decreasing cache pressure and other overheads. The core of TreeFuser is our *Dependence Graph* of general recursive tree traversals. The dependence graph serves as a good abstraction to analyze traversal calls that are safe to fuse. In our **PLDI '19** work Grafter [10], we have expanded the general recursive traversal fusion to handle heterogeneous trees — different nodes of the tree have different types. Grafter is a framework for fusing traversals of heterogeneous trees that is automatic, sound, and fine-grained.

Emerging Domains. Implementations of many different computations often require total or partial redesign when they are performed in an emerging domain to either fit the constraints or utilize specialized features of the domain. In general,

this results in alteration of the parts of software stack. In recent years, *Secure Multi-Party Computations (Secure MPC)* has become a domain of its own where many practical applications are written such as secure online auctions, privacy-preserving machine learning, etc. Secure MPC application setups require cryptographic expertise to express computations and orders of magnitude slower than classical computations. This necessitates optimizing Secure MPCs to have acceptable runtimes. When Secure MPCs are irregular, it adds another layer of complexity in expressing and optimizing these computations. My **GPCE '21** work [11] introduced HACCLE, an ecosystem to build Secure MPC applications. HACCLE is a compilation framework to build and execute MPC applications written in *Harpoon*, an embedded domain-specific language (eDSL) in Scala. The language is designed to be expressive enough for programming imperatively, while being constrained to ensure correct translation to low-level secure computing primitives.

Future Work

My prior work was mainly focused on schedules for irregular computations. In the future, I would like to explore the axis of generality in optimizing irregular computations by incorporating abstractions for both schedules and data. As for **Broader Impacts** of my research agenda, thinking about schedules and data in a unified manner enables the automation of many complex optimizations, and benefit systems developers by improving the performance of their programs. Moreover, irregular computations appear in numerous application domains that are at the frontiers of science such as data mining, machine learning, and computational genomics. Unification of schedules and data will allow programmers of these applications to focus on simple implementations while being able to deliver provably correct optimizations.

Abstractions for Performance. The space of data layouts offers an avenue for performance gain of programs. Scheduling transformations are locally applicable, which makes the reasoning relatively less complicated than choosing the best data layout. Choice of data layout requires a global analysis of the program and clever data abstractions. Even though the study of schedules for regular programs is mature, finding the best layout is still a growing area of research. There has been a few limited attempts for irregular programs to choose best data layout for programs such as tree traversals. The interplay between schedules and data layouts (since schedules influence data layouts and vice versa) is an interesting future direction that I wish to pursue and it requires composable abstractions for both schedule and data layouts.

Abstractions for Analysis. In my prior work, I focused on verifying the soundness of schedules given the dependences of an irregular program. Additional information of the properties of data structures used in a program can help us to automatically resolve these dependences. Loop transformation frameworks often come with a *General Dependence Analyzer* for programs that operate over arrays with linear accesses (*e.g.,* omega test [12]) and it helps to significantly automate the compilation pipeline. Past work on dependence analysis for irregular programs are mostly designed for specific transformations (*i.e.,* not general enough) since they were not meant to be *composable*. The composable scheduling transformations for irregular programs brings the need for designing generalized dependence analyzers and we are currently working on formulating one for programs operate over pointer-based data structures such as trees and lists.

Emerging Domains. The domain of *Sparse Tensor Computation* is getting traction in recent years. In the past, tensor computation was geared towards dense tensors — tensors with majority of the entries are non-zeros — which is handled by work centered on improving regular computation. The advent of sparse tensors — tensors with majority of the entries are zeros — showing up often in many essential applications such as recommendation systems, scientific simulations, and big-graph processing, sparse tensor computation has emerged as a separate domain. Sparse tensors are stored in compressed formats as opposed to dense tensors that are stored in regular arrays with random access to its elements. Computations over sparse tensor storage is irregular and tedious for manually programming due to the *data-irregularity*. Recently, there were many efforts to tackle the code generation problem of sparse computations such as Tensor Algebra Compiler (TACO) [13], which comes with class of primitive scheduling transformations akin to the ones from loop transformation frameworks to improve the performance. In our preliminary work *SparseLNR*, we extended the *iteration graph* abstraction of TACO to handle *imperfectly nested loops* and it resulted in **ICS'22** paper in which we introduced *Fusion/Distribution* transformations to irregular loop nests over sparse tensors. *SparseLNR* has been named **Best Paper** at ICS '22. Although sparse tensor computations is a growing domain, there has been numerous prior work on schedules and data formats. The emergence of various recursive decompositions for very large tensors, and embedding of recursive data structures such as trees in tensors give rise to control-flow irregularity in sparse tensor computations. Therefore, many applications that use sparse tensors would find the research on abstractions for reasoning about schedules and data layouts together impactful.

Conclusion

Given the ubiquitous nature of irregular computations, I believe that one way to fill the "*room at the top*" is focusing our efforts on abstractions for performance and analysis, while accommodating irregularity in the software stacks of emerging domains. My past work on generalizing the scheduling transformations for irregular computations makes me uniquely suited to lead impactful research in the area of compilers and programming languages. Exploring sparse tensor domain and data abstractions for irregular computations gives me concrete future directions while expanding my research agenda. I believe that computing in post-Moore's law era would be heavily influenced by compiler researchers and I hope to be a part of it.

References

[1] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[2] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), Jan 2012.

[3] Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. Locality transformations for nested recursive iteration spaces. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 281–295, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Ryan Curtin, William March, Parikshit Ram, David Anderson, Alexander Gray, and Charles Isbell. Tree-independent dual-tree algorithms. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1435–1443, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[5] Nikhil Hegde, Jianqiao Liu, Kirshanthan Sundararajah, and Milind Kulkarni. Treelogy: A benchmark suite for tree traversals. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 227–238, 2017.

[6] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pages 28–40. IEEE Press, 2022.

[7] Kirshanthan Sundararajah and Milind Kulkarni. Composable, sound transformations of nested recursion and loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 902–917, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. Unirec: A unimodular-like framework for recursions and loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2), Oct 2022.

[9] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. Treefuser: A framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[10] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. Sound, fine-grained traversal fusion for heterogeneous trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 830–844, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoit Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. Haccle: Metaprogramming for secure multi-party computation. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2021, pages 130–143, New York, NY, USA, 2021. Association for Computing Machinery.

[12] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. Association for Computing Machinery.

[13] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.