Kirshanthan Sundararajah
Research Statement

Hardware miniaturization was the core of performance gain. 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 getting free performance with every hardware upgrade is dying out as the fundamental physical limitations of transistors are approached. Even though running out of this free lunch seems to be unpromising, "There's plenty of room at the Top," the 2020 Science Journal article by Leiserson *et al.* [1] suggests that performance engineering of software and development of algorithms are ever more critical and continue to deliver faster applications in the post-Moore's law era.

In the Moore's law era, software development was generally focused on the ease of implementation rather than the efficiency and scalability of execution because the hardware improvement guaranteed performance gain. Focusing on the ease of development led to massive inefficiencies in programs commonly known as software bloat. In the post-Moore's law era, the performance transparency (reasoning about the performance of execution merely from the code) has been lost. Now the burden of ensuring performance gain has fallen on the hands of software developers, and I believe that through introducing performance transparent abstractions in programming languages and compilers, we can find principled solutions to this problem. Specializing these abstractions to a class of computation, programming model, or architecture is essential in making effective and efficient solutions.

Performance transparent abstractions 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 pipeline PDE solvers, and machine learning. On the other hand, we have *irregular computations* – those that operate on irregularly structured data, such as programs operating on sparse matrices, trees, and graphs using recursion with statically difficult to predict control flow – and these computations 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 ones.

Research Goals

My research goals are focused on filling the "room at the top" in a principled way with an eye for *Irregularity*. The involvement of irregularity makes this issue challenging. People often take an isolated approach and provide ad-hoc solutions to these challenges on a case by case basis. This may solve a specific problem, but these solutions often lack generality. The challenges due to irregularity are general and pervasive. Hence it is paramount to provide general and extensible solutions. I believe that generalization can be achieved by accomplishing the following with a focus on irregularity,

Designing Abstractions for Performance: The issue with irregular computations is often capturing them properly to optimize for better performance. Abstractions for capturing irregular computations should be general enough, but should have enough information to optimize performance. Transforming the captured computation is the key to better performance. Abstractions at the right level can help us achieve this goal.

Designing Abstractions for Analysis: Capturing irregular computations is not sufficient if we cannot analyze different ways for them to be transformed correctly. Abstractions for analyzes

should be cleverly designed to be *composable* – facilitating multiple analyzes to be used in conjunction – and *decidable* – existence of effective procedures to check the desired properties.

Designing Software Stacks for New Domains: There are often emerging new domains requiring us to develop software stacks (*e.g.,* programming language, compiler, runtime systems, etc.) for computing. We should design these systems with the ability to easily express and optimize irregular computations.

In my prior work, and in the future I have been guided by these research goals.

Prior Work

My prior work has manifested this research philosophy for a computation class with Control-Flow Irregularity when naturally expressed (*i.e.,* programs with recursion). The abstractions for performance and analysis are implemented as part of a compiler's transformation and analysis pipeline.

Abstractions for Performance One standard technique for optimizing regular nested loop-based programs targeting machines with multi-level caches is multi-level tiling – creating multiple levels of tiled loops, with each tile fitting in a particular level of cache. This requires the knowledge of sizes and number of caches for better performance (*i.e.,* cache-aware). An alternative strategy is being cache-oblivious with divide and conquer implementations – recursively subdivide the iteration space so that some level fits in cache [2]. In my **ASPLOS '17** work [3], we have introduced Recursion Twisting, a transformation that can be applied to computations captured with the abstraction of nested recursion to generalize cache-oblivious multi-level tiling approach to recursive programs.

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 profitable locality benefits, as the smaller tree is more likely to fit in cache. The insight of recursion twisting is that after partially traversing the outer tree, it may now be smaller than the inner tree. Swapping the order of recursion (an analog of loop interchange) will then place the smaller tree on the inside, providing better locality. Continuing this swapping process will naturally lead to smaller trees fitting in cache, giving good cache locality in a cache-oblivious manner. Ensuring that this complicated recursion pattern does not violate any dependences, and does not break the overall computation, requires careful management of data. Application of recursion twisting to *dual-tree* implementations of *Generalized N-body Problems* from Curtin *et al.* [4], results in order-of-magnitude performance enhancement.

Abstractions for Analysis In the world of nested loop based programs, a number of frameworks, the most popular example being the polyhedral model, possess the capability to reason about the correctness of the transformations. Proving that a transformation is safe means providing tests to ensure that the dependences in the original program are not violated. Often complex transformations for these programs are compositions of multiple smaller transformations. It is important to build abstractions that help us to reason about a composed sequence of transformations, because a safe transformation can be composed of multiple transformations that are individually unsafe for a given program. In my **PLDI '19** work [5], we have introduced *PolyRec*, a framework to perform scheduling transformations for nested recursive iteration spaces. Nested recursion is a broad abstraction to capture the mix of recursion and loops and recursion since any loop can be treated as tail recursion.

The *PolyRec* framework represents the iteration space of perfectly nested recursive programs with the abstraction of multitape automaton (*i.e.,* non-deterministic finite state automata with multiple input tapes) and the scheduling transformations with 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 test check their preservation. We introduced witness tuples, an abstraction similar to, but more expressive than, the distance vector from the prior loop transformation frameworks, to represent dependences. The crux of *PolyRec* is a decidable algorithm for checking the preservation of dependence (dependence check). The abstractions for the representation of iteration space, transformation and dependence properly fit into the dependence check because the design of the abstractions affects the design of the algorithm and vice versa.

Software Stacks for New Domains Implementation of many different computations often requires total or partial redesign when these computations are performed for a new domain to either fit the constraints of the domain or utilize specialized features of the domain (*e.g.,* a new processor with energy-efficient instructions). This need for redesign often results in alteration of the parts of the software stack (*i.e.,* libraries, programming languages, and runtimes). In recent years, Secure Multi-Party Computations (Secure MPC) – Protocols that enable multiple distrusting parties to jointly perform computations without revealing any private data to non intended parties (*e.g.,* Group of people wants to find the wealthiest person among them without revealing each one's wealth) – has become a separate domain where people write many practical applications such as secure online auctions, privacy-preserving machine learning, privacy preserving genomics etc. Secure MPC application setups are inherently complicated and require cryptographic expertise to write even a simple computation. Secure MPCs are orders of magnitude slower than performing the same computation without any cryptographic steps (*i.e.,* classical computation). This necessitates optimizing Secure MPCs to have acceptable runtimes and it requires knowledge of circuit optimization since Secure MPCs are often expressed in the form of circuits. When Secure MPCs are irregular, it adds another layer of complexity in expressing and optimizing these computations. In my **GPCE '21** work [6], we have 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 based on the Lightweight Modular Staging (LMS) metaprogramming and compiler platform. Haccle Rich Representation for Program OperatiON (Harpoon) language is an expressive subset of Scala for writing MPC programs. It is an imperative and monomorphic language, featuring standard control flow operations: loops, function calls, conditionals, and recursions. The language is designed to be expressive enough that programmers could easily write Harpoon code directly, while being constrained enough to ensure that Harpoon programs can be implemented via translation to secure low-level computation. Harpoon programs are lowered to HACCLE Intermediate Representation (HIR), an extensible circuit-like intermediate representation tailored to abstract cryptographic primitives used in Secure MPC.

Collaborations In the past, I have collaborated on different projects with an angle for irregularity that align with my research philosophy. In our **ISPASS '17** work [7], we have introduced *Treelogy*, a benchmark suite for tree traversals. Treelogy differs from other benchmark suites by providing an ontology to match algorithms and state-of-the-art optimizations based on the structural properties of tree traversals. In our **OOPSLA '17** work *TreeFuser* [8], we have explored automatically fusing general recursive tree traversals – transforming multiple recursive functions that traverse the same tree into one function – for reducing number of traversals, increasing opportunities for optimization and decreasing cache pressure and other overheads. The core of TreeFuser is our dependence graph of general recursive tree traversals and the dependence graph serves as the correct level of abstraction to analyze traversal calls that are safe to fuse and generate fused code. In our **PLDI '19** work *Grafter* [9], we have expanded the general recursive traversal fusion to 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. Our **CGO '22** work *DARM* [10] was focused on using fusion for reducing control-flow divergence in GPGPU programs. GPGPUs use the Single-Instruction-Multiple-Thread (SIMT) execution model where a group of threads execute instructions in lockstep. When threads in a group encounter a branching instruction, the execution diverge and it degrades performance. DARM is a compiler analysis and transformation framework that can meld divergent control-flow structures with similar instruction sequences to reduce performance degradation. DARM stands out due to its ability to handle arbitrary control-flow.

Ongoing Work

My current work is focused on generalizing the *PolyRec* framework to include scheduling transformations for nested recursive iteration spaces that ultimately target to expose parallelism. In loop transformation frameworks such as the unimodular framework, loop skewing is a powerful scheduling transformation that reveals many parallelization opportunities for regular loop nests when composed with loop interchange, and loop reversal. Although *PolyRec* handles interchange and reversal for nested recursive iteration spaces, its representation is insufficient to address skewing. In this work, we introduce the notion of skewing to recursive iteration spaces and improve *PolyRec*'s representation to handle the composition of skewing with other transformations. Finally, we prove that with minimal changes, we can use *PolyRec*'s correctness checking algorithm.

Future Work

My future work lays out ideas to permeate my research philosophy in new areas both tightly and loosely connected to my prior work. In the future, I would like to explore the axis of generality for transformations on irregular programs by incorporating both schedules and data layouts.

Abstractions for Performance My prior work was mainly focused on scheduling transformations for irregular programs. Another profitable avenue for program transformations lies in the space of the data layout (*e.g.,* In a naive matrix multiplication code for $C = A * B$, column-major linearization of the matrix B improves the performance assuming the default data layout is row-major). Scheduling transformations are locally applicable in general, which makes the reasoning relatively less complicated. On the other hand, *data layout transformations* require whole program analysis and restructuring. Difficulty in analysis and application hinders the wide adoption of data layout transformations. This often leads to restricting the space of programs and separating the application of data layout transformation from other transformations (*e.g.,* scheduling transformations).

Even though the application of scheduling transformations for regular programs is mature, data layout transformations are applied to particular classes of programs such as convolutions. There were few restrictive attempts for irregular programs to incorporate data layout transformations for programs such as tree traversals. Indeed, the interplay between scheduling transformations and data layout transformations (since schedules influence data layouts and vice versa) is far in the future. By building abstractions to connect schedules and data accesses, we can make the data layout transformations composable with scheduling transformations. This may lead to a viable strategy for cost-benefit analysis of various schedules and data layouts and eventually reduces programmers' concern about the implementation of recursive abstract data types.

Abstractions for Analysis In my prior work of constructing abstractions for analysis, I was focused on verifying the soundness of scheduling transformations 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 the dependences. Mature loop transformation frameworks such as the polyhedral model often come with a *General Dependence Analyzer* for programs

that operate over multidimensional arrays with linear accesses (*e.g.,* omega test [11]). Having a general dependence analyzer helps to significantly automate the transformation pipeline with less programmer involvement.

The past works on dependence analysis for irregular programs are mostly designed for a specific scheduling transformation since these transformations were not meant to be composable. Therefore these dependence analyses are not general enough. Advent of composable scheduling transformation frameworks for irregular programs brings the need for designing Generalized Dependence Analyzers. The idea behind generalized dependence analysis is abstraction of access to data structures and mapping these accesses to iterations since they are going to be used for transformations.

Software Stacks for New Domains Conventionally, computing over tensors is considered as regular computation since these tensors were mostly dense (*i.e.,* insignificant number of empty elements) and linear memory access patterns that come with the loops that traverse over them. But pervasive sparse tensor computation – computing over tensors with significant number of empty elements – has brought irregularity into this domain.

Irregularity in tensor computation can be categorized as data irregularity – computation with significant number of empty elements – and control-flow irregularity – computation is recursive when naturally expressed. Although it is a growing field, there has been numerous prior work on transformations for data irregular computation. The advent of various viable recursive decompositions of very large tensors and embedding of recursive data structures such as trees in tensors gives rise to control-flow irregularity in tensor computations. Since there are increasingly different data layout formats and transpositions of tensors, research on unified frameworks for searching and reasoning about schedules and data layouts specifically for irregular tensor computations would be beneficial in future.

Conclusion

I believe that given the ubiquitous nature of irregular computations, one way to fill the "room at the top" is focusing our efforts on abstractions for analysis and acceleration of programs and designing software stacks for emerging domains to accommodate irregularity. I am uniquely suited to lead impactful research in the area of compilers and programming language especially when the computations are irregular. I have a research philosophy with three thrusts that is proven effective in the past and concrete future directions for each thrust. 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction, 2021.

[11] 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.