SIMD Code Generation for Stencils on Brick Decompositions

Tuowen Zhao, Mary Hall
School of Computing
University of Utah
{ztuowen,mhall}@cs.utah.edu

Protonu Basu, Samuel Williams, Hans Johansen
Computational Research Division
Lawrence Berkeley National Laboratory
{pbasu,swwilliams,hjohansen}@lbl.gov

Abstract

We present a stencil library and associated compiler code generation framework designed to maximize performance on higher-order stencil computations through the use of two main technologies: a fine-grained *brick* data layout designed to exploit the inherent multidimensional spatial locality endemic to stencil computations, and a *vector scatter* associative reordering transformation that reduces vector loads and alignment operations and exposes opportunities for the backend compiler to reduce computation. For a range of stencil computations, we compare the generated code expressed in the brick library to the standard tiled code. We attain up to a 7.2× speedup on the most complex stencils when running on an Intel Knights Landing (Xeon Phi) processor.

CCS Concepts • **Software and its engineering** → **Source code generation**; • **Computing methodologies** → *Parallel programming languages*;

Keywords Compiler Optimization, Stencil, SIMDization

ACM Reference Format:
Tuowen Zhao, Mary Hall and Protonu Basu, Samuel Williams, Hans Johansen. 2018. SIMD Code Generation for Stencils on Brick Decompositions. In *Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 2 pages.
https://doi.org/10.1145/3178487.3178537

1 Introduction

The solutions to partial differential equations using the finite difference or finite volume methods result in *stencil* computations wherein space is discretized and the derivative at each point in space is calculated as a weighted sum of neighboring point values. A stencil's *order of accuracy* is the exponent on the relationship between grid spacing (array size) and error — both small grid spacings (large arrays) and high order can result in low error. Low-order stencils are typically bound by memory bandwidth and thus underutilize the compute capability of modern architectures. Much prior work is based on lower order stencils and has thus focused on techniques to reduce data movement [3, 7, 9].

PPoPP '18, February 24–28, 2018, Vienna, Austria
2018. ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178537

Recognizing that processor architectures are becoming more compute-intensive, computational scientists are increasingly turning to high-order schemes that can attain equal error with substantially less data movement (smaller arrays). Although, the larger radius of high-order stencils results in more computation per point (higher arithmetic intensity), it also places higher pressure on the caches and TLBs, can challenge SIMD code generation, and, in extreme cases, can be limited by the compute capability of modern architectures [2, 4, 8].

Our approach develops a novel brick data decomposition and SIMD code generator for stencils, making the following contributions to optimizing this important class of computation: (1) it presents vector scatter, an associative reordering technique that generalizes scatter to target wide SIMD instruction sets; (2) it utilizes brick decompositions to improve memory hierarchy and TLB utilization, accelerate data copies, and improve inter-domain ghost zone exchange performance; (3) it presents the first node-level code generation for wide SIMD operations on bricks; and, (4) it improves performance up to 7.2× on the most complex stencils running on a Intel Knights Landing (Xeon Phi) processor thereby consistently delivering high performance on both compute- and memory-intensive stencils.

2 Brick Data Layout and Code Generation

A *brick* is a high-level data layout that decomposes a grid domain into small, fixed-size multi-dimensional subdomains stored contiguously in memory [1, 6, 10]. Although bricks have been established as being well-suited for high-order stencils, high-performance SIMD code generation has been a challenge due to the complexity of their nearest-neighbor memory access patterns. Neighbor accesses that cross brick boundaries introduce overhead from additional index calculations and vector shifts.

To address the code generation problem, we developed *vector scatter*, a compiler code generation technique that reuses vector register data within and across stencils and optimizes alignment operations on the data. As in prior work [2, 4, 8] on optimizing high-order stencils, we use *associative reordering*, recognizing that the weighted sums in a stencil computation are associative operations that can be safely reordered. Consequently, computation order can be optimized to reduce memory loads and reduce register pressure.

Name	Description	Name	Description
A1	7-pt, 2^{nd} order	A2	13-pt, 4^{th} order
A3	19-pt, 6^{th} order	A4	25-pt, 8^{th} order
C1	27-pt, 2^{nd} order	C2	125-pt, 4^{th} order
iso	8^{th} order, isotropic finite difference [10]	CNS	8^{th} order, compressible Navier Stokes [5]

Table 1. Stencils used in experiments.

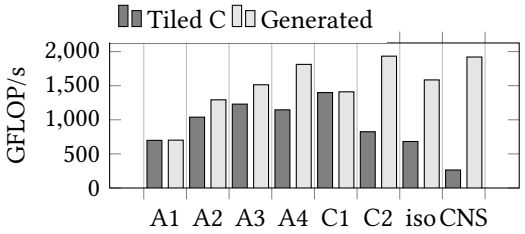

Figure 1. Resultant stencil performance on a KNL node.

The main goal of vector scatter is to eliminate as many vector load and alignment operations as possible. Vector scatter operates on the expression DAG of the source code and its variants. A nice side effect of vector scatter is *array common subexpression elimination*, which eliminates redundant computation of compute-limited stencils.

Ultimately, application developers write their code using the brick library in C. The coding style is similar to programming with arrays, with one extra dimension to denote the brick index. All accesses beyond the dimension of a brick are automatically translated to accessing the corresponding neighboring brick. Computation within a brick will be transformed by the compiler while the application developer retains control over how computation on the set of bricks is parallelized. This approach can fully vectorize stencil codes with AVX-512 using the Intel C Compiler 2018.

3 Experimental Results

Table 1 describes the stencils used in the evaluation. It includes six synthetic ones to capture the memory-compute ratio of different kind of stencils due to their shape and radius; and two real-world stencils `iso`, where one stencil's result is used in another and, `CNS`, where fifteen different stencils operate on a dozen separate fields. For comparison, we use as a baseline a high-quality tiled implementation annotated with pragmas for vectorization, unroll and jam, and nontemporal stores.

We evaluate the code on the Intel Xeon Phi 7250 Knights Landing in *quadflat* mode. A 512^3 domain is decomposed into rectangular subdomains which are further decomposed into individual bricks. Each subdomain is assigned to one KNL tile (2 cores / 8 threads) and is further decomposed into planes of bricks. We use nested OpenMP to map subdomains to one of the 68 tiles (dynamic OpenMP scheduling) and to map the brick planes within a subdomain to individual threads on a KNL tile (static OpenMP scheduling). The baseline code also uses this strategy. Figure 1 shows the generated code now delivers increasing performance with increasing complexity (arithmetic intensity) attaining a 7.2× speedup for `CNS`. In addition, we are 1.34× faster than YASK (1183.6 GFLOP/s) [10] on the `iso` stencil.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

References

[1] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. 2009. 3D Seismic Imaging Through Reverse-time Migration on Homogeneous and Heterogeneous Multi-core Processors. *Sci. Program.* 17, 1-2 (Jan. 2009), 185–198.

[2] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 313–323.

[3] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.* 51, 1 (2009), 129–159.

[4] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th international conference on Supercomputing*. ACM, 65–77.

[5] Matthew Emmett, Weiqun Zhang, and John B Bell. 2014. High-order algorithms for compressible reacting flow with complex chemistry. *Combustion Theory and Modelling* 18, 3 (2014), 361–387.

[6] Jagan Jayaraj. 2013. *A strategy for high performance in computational fluid dynamics.* Ph.D. Dissertation. University of Minnesota.

[7] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*.

[8] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[9] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In *International Computer Software and Applications Conference*.

[10] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK-yet Another Stencil Kernel: A Framework for HPC Stencil Code-generation and Tuning. In *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC (WOLFHPC '16)*. IEEE Press, 30–39.