Automatic Vectorization of Stencil Codes with the GGDML Language Extensions

Nabeeh Jumah
Researcher
Informatik Department
Universität Hamburg
Hamburg, Germany
jumah@informatik.uni-hamburg.de

Julian Kunkel
Lecturer
Computer Science Department
University of Reading
reading, UK
j.m.kunkel@reading.ac.uk

Abstract

Partial differential equation (PDE) solvers are important for many applications. PDE solvers execute kernels which apply stencil operations over 2D and 3D grids. As PDE solvers and stencil codes are widely used in performance critical applications, they must be well optimized.

Stencil computations naturally depend on neighboring grid elements. Therefore, data locality must be exploited to optimize the code and to better use the memory bandwidth – at the same time, vector processing capabilities of the processor must be utilized.

In this work, we investigate the effectiveness of using high-level language extensions to exploit SIMD and vectorization features of multi-core processors and vector engines. We write a prototype application using the GGDML high-level language extensions, and translate the high-level code with different configurations to investigate the efficiency of the language extensions and the source-to-source translation process to exploit the vector units of the multi-core processors and the vector engines.

The conducted experiments demonstrate the effectiveness of the language extensions and the translation tool to generate vectorized codes, which makes use of the natural data locality of stencil computations.

Keywords HPC, Earth system modeling, Stencil computation, SIMD

1 Introduction

Stencil computations are essential for many applications including PDE solvers. Earth system modeling is one of the sciences that uses stencil codes extensively to simulate processes on the earth surface. The modern simulation applications are highly demanding for performance. Simulation codes should optimally use the features of the hardware to allow the models to run under specific time constraints.

Among the hardware features of the modern architectures are the vector units and the single instruction multiple data (SIMD) operations. Such capabilities allow the machine to execute a single mathematical operation on a whole vector at once. Besides to the vectorized mathematical operations, such architectures provide data movement instructions that operate on vectors.

General-purpose language compilers can generate vector instructions to handle the data movement and the mathematical operations. However, they require that data is in a suitable layout in memory, and they recognize appropriate patterns. To exploit the architectural capabilities of handling vector operations, the structure of the data in memory should be optimal to allow the vector data movement instructions to be used. Unit stride arrays access, where data are contiguously placed in memory, and accessed with the right loop orders are best fit for vectorization. Therefore, transforming layouts from array-of-structures (AoS) to structure-of-arrays (SoA) and the alignment of these arrays is essential to use vector instructions. Stencil computations are naturally fitting to a SoA data structure as computations are based on access to neighboring grid elements.

In this paper, we discuss using high-level coding and the capability to automatically generate vectorized codes from a high-level source code using source-to-source translation tools. We demonstrate the capabilities on a prototype application that solves the shallow water equations [1] developed with the General Grid Definition and Manipulation Language (GGDML) set of high-level language extensions [7] according to the source-to-source translation process suggested in [6]. A tool translates the GGDML code and applies certain optimization procedures driven by user-provided configuration files.

One important feature of the mentioned translation process is the flexible way to configure the memory layout by expert users, which gives them full control on the placement of the elements in memory. This allows the users to

WPMVP'19 , February 16, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6291-7/19/02…$15.00
https://doi.org/10.1145/3303117.3306160

choose the data layout that enables compilers to use vector operations efficiently.

The **main contribution** of this work is the extension of the translation process by applying user-guided loop transformations that tune the access within loops to match the memory layout in order to achieve optimal vectorization. The idea is to use the right combinations of memory layout and loop structure based on the semantics provided by GGDML from the source code and the provided user configuration.

This paper is structured as follows: First, we discuss related work in Section 2. Next, in Section 3, we discuss the methodology that we use for this work. The experimental results are presented in Section 4. Finally, the work is concluded in Section 5.

2 Related Work

Efforts to exploit vector units and vector instructions evolved from manual optimization to automatic tool-aided solutions, and to vectorize data with multiple dimensions.

Data layout and loop transformations: A data layout transformation technique to solve stream alignment conflict in stencil computations is presented in [4]. Again data layout and other loop transformations were used to optimize a set of high order finite difference kernels in [12] to support vectorization.

Autotuning and DSLs: Besides to the manual optimization of the stencil codes to exploit the vectorization capabilities of the underlying hardware, other techniques were explored including autotuning and domain-specific languages (DSLs). An autotuning technique was proposed in [3] to optimize stencil computations. In this work, the authors developed a set of optimization strategies, and an autotuning environment to select a strategy with the right parameters to minimize execution time. Autotuning was used in [8] to generate optimized codes based on sequential Fortran 95 codes.

PATUS [2] is a code generation and autotuning framework to develop stencil computations that used autotuning together with a DSL. A C-like DSL is used to describe stencils, and the framework generates the code using optimization strategies. Another DSL, the SDSL (Stencil domain-specific language), was proposed in [5] to describe stencil computations. A compiler is also presented to generate code which goes through an optimization process that includes vectorization.

Specialized DSLs: A more complicated technique that improves vectorization besides to caching improvements was presented in [10]. Vector folding is a technique to vectorize data in 2D and 3D stencils. The technique constructs vectors from multi-dimensional data blocks instead of the conventional vectorization over a single dimension. The technique was used within YASK [11] to improve the use of the memory bandwidth and the vector units of the Xeon and the Xeon Phi processors. YASK provides a DSL to describe stencil computations; it then applies the vector folding technique automatically and generates optimized codes for specific processor families.

Our work differs from related work. While we use DSL concepts to describe stencil computations, we use a set of language extensions, which represent an addition to a general-purpose language. We use the GGDML set of language extensions described in [7]. Instead of using DSL compilers, which handle static grammars of DSLs, we use a flexible source-to-source translation technique described in [6] which can be configured and extended by experts and users. An important feature of GGDML and this translation technique is the adaptability and extensibility of the language according to the domain or the application needs, and the flexibility of the translation tool to handle the modifiable language extensions. With the tool, we are able to generate optimized codes for different architectures and configurations.

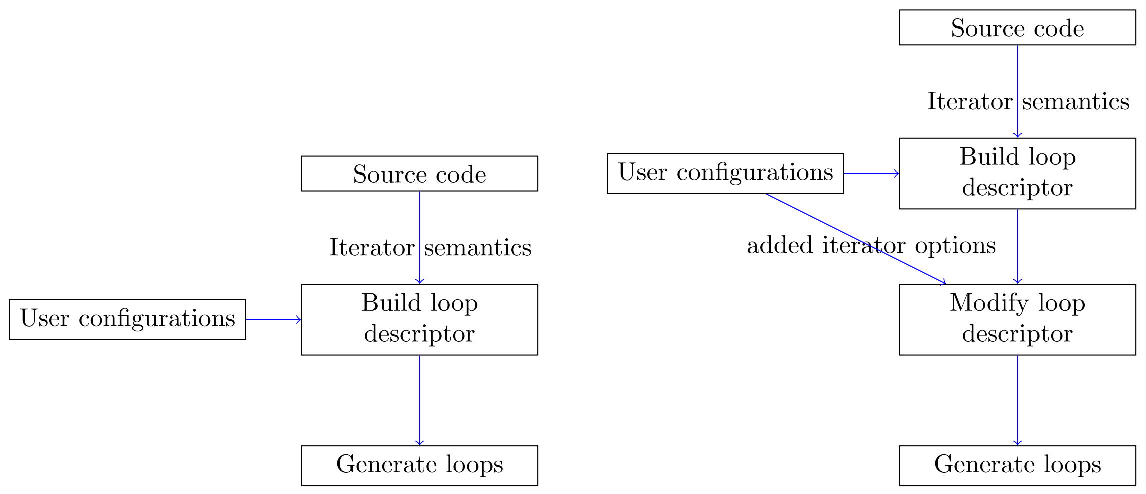
3 Methodology

The approach pursued in this paper is as follows: Based on the GGDML iterator semantics, we extend the translation process by transforming the internal data structure that describes the loop structure allowing to generate variants of loop structure and order with memory layout transformations (see Figure 1).

This approach is then evaluated:

- We develop a prototype application to solve the shallow water equations [1] using the GGDML language extensions.
- Initial configuration files to generate codes for the architectures of multi-core processors and vector engines are prepared.
- We added the necessary information to the initial configuration files to apply the new transformations.
- Clones of the configurations are created to explore alternative array stride and data placement options and check the impact on vectorization. We explore three different alternatives:
  - Contiguous data placement and access
  - Array data separated by a short distance (4 bytes between two consecutive array elements)
  - Scattered data elements, where loops access data in a bad order
- Code is translated based on the different configurations (different data placement and access and different architectures).
- The different versions were run and profiled with Likwid on the Broadwell and Ftrace on the Aurora.

The generation of the scattered array accesses is enabled by the capabilities of the translation technique. The same



(a) Original loop optimization **(b)** Extended loop optimization

Figure 1. Extending loop optimization to generate vectorized code

is true for the arrays with separated elements, where array elements are separated by some distance according to some formula (e.g. separated by a constant distance of one element or four bytes in the case we test in this paper). This capability is useful for example to allow users to evaluate SoA performance, while still using AoS code.

3.1 GGDML

GGDML is a set of language extensions that was developed to provide performance portability for earth system modeling applications. It was developed to use scientific abstractions for the code development, and to use the semantics of those abstractions to allow the translation tools to optimize the code through the translation process.

The use of GGDML is demonstrated in the example code snippet in Listing 1, which demonstrates field declaration besides to a simple iterator.

Listing 1. Example GGDML based code

```
float EDGE 2D f_U;
float EDGE 2D f_UT;
...
foreach e in grid
{
  f_U[e]=f_U[e]+f_UT[e]*dt;
}
```

Two fields are declared in this example on the edges of the a two-dimensional grid. An iterator updates the values of one field according to a mathematical expression. The code tells that the expression is applied over the edges of the grid, but not the actual memory allocation and access.

Another code example (Listing 2) shows the use of user-defined language extensions to represent relationships between grid components. Those extensions simplify the access to the other related components. In the example, the tendency of the surface level (f_HT) is computed at the centers of the grid cells. It is computed based on the X and Y components of the flux (f_F and f_G respectively), which reside on the edges of the grid cells. The keywords *east_edge*(), *west_edge*(), *north_edge*(), and *south_edge*() allow referring to the edges of the cell *c*. Edge references are abstractions of the grid concepts, where no details about where the data is actually stored in memory.

Listing 2. Grid component relationships in GGDML

```
float CELL 2D f_HT;
float EDGE 2D f_F;
float EDGE 2D f_G;
...
foreach c in grid
{
     float df=(f_F[c.east_edge()]-
            f_F[c.west_edge()])/dx;

     float dg=(f_G[c.north_edge()]-
            f_G[c.south_edge()])/dy;

     f_HT[c]=df+dg;
}
```

The iterator in this example traverses the cells of the grid. It uses the extensions to access the edges of the cell, and reads the flux values to eventually compute the surface level tendency. The demonstrated keywords to access the edges from the cell are defined by the users through configuration files.

User-provided configuration files are used to translate GGDML codes into a usable code. Different optimization procedures are applied during code translation. Configuration files include different sections to guide the application of different optimization procedures.

3.2 Code Translation

The high-level code, e.g., the iterator, is translated into optimized loops in general-purpose language. Parallelization, blocking, domain decomposition and other optimizations are applied during the translation process. The translation tools can decide which optimizations to apply based on the contents of configeration files.

We implement the translation technique using a Python script. Such an implementation allows to simply ship the translation tools with the code repositories and reduces tool maintenance efforts. The script parses the source code and generates an AST. Using the user-provided configuration files, the translation tool applies the different transformations to the AST.

The users control the parallelization strategy on the node to optimally use the underlying hardware, e.g. cores or warps and streaming multiprocessors. Parallelization on multiple nodes is also controlled by users, where users configure the communication of the different halo patterns. Domain decomposition can be either done automatically by the tool, or can be guided by the user.

Blocking is another example optimization procedure that can be applied by the tool. Users can control the blocking factor through a section within the configuration files that guides the tool to apply the necessary AST transformations to handle blocking.

Listing 3 is a sample from a configuration file demonstrating how the user provides different options to drive the code transformation process.

Listing 3. Example configuration file contents

```
...
RANGE OF YD= 0 TO GRIDY
RANGE OF XD= 0 TO GRIDX
...
CBLOCKING:
XD=10000
ENDCBLOCKING
...
INDEXOPERATORS:
  edge_right(): XD=$XD+1
...
```

The sample shows a section where the user defines the ranges of the two dimensions of the grid. It also shows how the user guides the blocking process by guiding the tool to block the X dimension with block size of 10000. Another section in this sample shows how the language extensions are specified to define the relationship between grid components, and how the indices are transformed based on those extensions.

Besides to the mentioned transformations, the flexibility of the memory layout transformations allows the users to control the placement of data in memory. To support an alternative layout, firstly, a configuration with the suitable transformation must be generated, secondly, the code is linked to the implementation of the alternative data structures. This allowed us in our experiments to generate codes with different striding options.

3.3 Matching Access to Loop Structure

The translation of the code in Listing 1 generates the code in Listing 4 based on a prepared configuration file. The array access indices are translated based on a specified data layout, where the *XD_index* (inner) loop accesses contiguous array elements.

Listing 4. Resulting vectorizable code

```
#pragma omp for
for (size_t YD_index = (0);
     YD_index < (local_Y_Eregion);
     YD_index++) {
#pragma omp simd
  for (size_t XD_index = blk_start;
       XD_index < blk_end; XD_index++) {
    f_U[(YD_index)][(XD_index)] =
      f_U[(YD_index)][(XD_index)] +
      f_UT[(YD_index)][(XD_index)]*dt;
  }
}
```

The code in Listing 4 is vectorizable, because the loop access patterns are matching the data layout to support the use of the vector operations instructions. However, changing the access patterns, e.g., interchanging the *XD_index* and *YD_index* loops or changing the data layout without changing the loop body accordingly (see the changed loop body in Listing 5), will limit the vectorization.

Listing 5. Resulting bad-performing operation not matching the selected loop order

```
  f_U[(XD_index)][(YD_index)] =
    f_U[(XD_index)][(YD_index)] +
    f_UT[(XD_index)][(YD_index)]*dt;
```

The main extension that we suggest to the translation technique described in [6] is to allow the user to guarantee the matching of the data layout and the access patterns within the loops to the data to optimally exploit vectorization.

All the accesses to the different fields in all stencil operations within a kernel are transformed through the memory layout transformation procedures, which decide the transformations for each access. Besides to those transformations, the structure and the order of the loops is transformed through the suggested transformation procedure, which allows the index transformations to use caches efficiently.

4 Evaluation

In this section, we describe the experiments and the results.

4.1 Test Application

The test application solves the shallow water equations on a 2D regular grid with cyclic boundary conditions. The application uses an explicit time stepping scheme in which eight fields are updated once in each time step. Each field update is executed within a single kernel.

4.2 Test Systems

The multi-core processor experiments are run on dual socket Broadwell nodes. The processors are Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz. We used the Intel C compiler (ICC 17.0.5 20170817).

The vector engine experiments were run on a machine from NEC with the SX-Aurora TSUBASA vector engine using the NCC (1.3.0) C compiler.

The used Broadwell processor contains 18 cores (36 threads), which share a 45 MB SmartCache for L3 caching. The maximum memory bandwidth of this processor is 76.8 GB/s. It supports the Intel(R) AVX2 instruction set extensions. The AVX2 works with registers of length 256 bits. The vector operations are applied with those vector lengths.

The SX-Aurora vector engine contains 8 cores. Those cores share a 16 MB last level cache. The maximum memory bandwidth of this vector engine is 1.2 TB/s. Each register on the SX-Aurora holds 256 entries (64 bits). Each core contains three FMA pipes each of which can handle 32 double precision floating point operations per cycle.

4.3 Multi-core Processors

First, we generated the code for the multi-core processors, in three code versions: scattered access, constant short distance between array elements, and contiguous array elements. We profiled the three code version with Likwid[9]. The measurements for the different kernels are shown in Table 1.

The first eight rows show the measurements for the different kernels. The runtime and the measured GFLOPS performance of the AVX instructions are shown in the three code versions. The last row summarizes the runtime and the measured GFLOPS of the whole application (total application-level measured GFLOPS and not only AVX).

For the code with contiguous array elements, if we multiply the time by the GFLOPS we find that the first eight kernels executed around $4.9 \cdot 10^{12}$ FLOP on the AVX vector units. This is close to the measured total GFLOPS of the application. In the code with constant short distance separating its elements, some kernels were vectorized. Performance of this code is nearly half that of the contiguous array version. However, for the code with scattered access, the executed operations on the AVX vector units are 0, and the performance was only 12% of the performance that the unit stride code achieves.

In fact, the vectorized arithmetic operations are not the only factor of those results, but also the vectorized data movement and memory access patterns. The memory access to the non-contiguous arrays degrades the role of the caches, and hence the use of the memory bandwidth. This explains the ratio of the performance of the array with constant short distance between elements to the performance of the contiguous array code. In the code with elements separated by constant short distance, we use 4 bytes to store a single precision value and there are 4 bytes separating the values, the ratio of the needed data is 4:8. Thus, the efficiency of using the memory bandwidth is half that of contiguous array, and performance is also the same ratio. For stencil computations it is well known that the optimal use of the memory bandwidth is critical to achieve an optimal performance. This is because stencil computations are memory bound.

Given the arithmetic intensity of 0.45 FLOP/Byte of the application, and the measured memory throughputs around 68 GB/s[1] (theoretical bandwidth 76.8 GB/s), the code with contiguous arrays is nearly optimal as it achieves 80% of the theoretical memory bandwidth. The code generation generated vectorizable code applying a single pattern across several operations.

4.4 Vector Engines

We generated again three code versions of the same source code for the Aurora vector engine: scattered access, constant short distance between array elements, and contiguous array elements. Ftrace was used for the measurements, in which we record performance metrics for the different kernels. Results are shown in Table 2.

The first eight rows show the measurements for the different kernels. The runtime and the measured GFLOPS performance are shown for both the three code versions. The last row summarizes the runtime and the measured GFLOPS of the whole application.

The arithmetic operations of all codes are executed by the vector units. However, the efficiency of using the vector units differs, where the contiguous array code is nearly twice the performance of the code with constant short distance between array elements, and four times faster than the code with scattered access. Again, the vectorization of arithmetic operations is not the only factor of this result, but also memory access patterns. As with multi-core processors, the

[1]According to Likwid's stream_sp_mem_avx benchmark

Kernel	Scattered		Constant short distance		Contiguous	
	Time (s)	AVX GFLOPS	Time (s)	AVX GFLOPS	Time (s)	AVX GFLOPS
flux1	250	0	52	0	27	11
flux2	248	0	54	0	27	11
compute_U_tendency	431	0	80	21	41	41
update_U	158	0	39	0	20	10
compute_V_tendency	432	0	94	18	47	37
update_V	158	0	40	0	20	10
compute_H_tendency	251	0	55	0	28	11
update_H	158	0	40	0	20	10
Application Level	2,103	3	466	13	244	25

Table 1. Performance measurements on Broadwell

Kernel	Scattered		Constant short distance		Contiguous	
	Time (s)	GFLOPS	Time (s)	GFLOPS	Time (s)	GFLOPS
flux1	5.37	56	3.96	76	1.30	230
flux2	5.36	56	4.08	74	1.51	199
compute_U_tendency	20.67	92	8.26	230	5.29	359
update_U	3.82	52	2.44	82	1.21	166
compute_V_tendency	20.66	97	9.12	220	5.22	384
update_V	3.82	52	2.43	82	1.21	165
compute_H_tendency	6.88	73	4.26	117	1.52	330
update_H	3.82	52	2.44	82	1.20	167
Application level	70.40	80	37.17	161	18.63	322

Table 2. Performance measurements on the NEC Aurora

memory access to non-contiguous arrays degrades the role of the caches, and hence the use of the memory bandwidth.

As mentioned before, the arithmetic intensities of the application level are 0.45 FLOP/Byte. The theoretical memory bandwidth of the used vector engine is 1.2 TB/s. Based on the numbers, the code with contiguous array elements is nearly optimal as it runs with a high percentage (80%) of the theoretical memory bandwidth.

5 Summary

In this paper, we presented a strategy to generate and optimize high level codes in order to match data layout and memory access patterns for yielding vectorizable code. We also investigated the performance impact of changing the array stride and data layout and access in the generated codes.

For the evaluation, we used an application that solves the shallow water equations, which is a typical model that uses stencil operations to solve partial differential equations. The application is written in higher-level code using the GGDML language extensions. A single source code was used for the different experiments on the different architectures. To make this possible we prepared configuration files to generate code for multi-core processors (Broadwell) and for vector engines (NEC Aurora).

The experiments included generating codes with different array strides/layouts and access patterns. The source code consists of eight kernels. We ran the experiments with different configurations changing array strides on two architectures. Performance of the application were recorded by the Likwid and Ftrace on Broadwell and Aurora, respectively. The experimental results were theoretically discussed to explain the measurements.

Results show the impact of generating unit stride code, which provides twice the performance of codes with constant short distance separating array elements, and about eight times the performance of scattered access on the Broadwell processors. On the Aurora vector engine, the unit stride code achieves twice the performance of the code with constant short distance between array elements, and four times the performance of the code with scattered access. The use of the GGDML language extensions allowed to productively write scientific codes in a single source code, and the use of the translation technique provided performance-portability to get nearly optimal code on the two platforms. A key benefit compared to manual code optimization is to reuse and apply efficient patterns across multiple kernels.

5.1 Future Work

To optimize generated code further, we will look into inter-kernel optimizations across time steps of an application. We will still use the high-level semantics of the GGDML language extensions, but the translation technique can be extended to automatically explore those temporal kernel merges besides to spatial kernel merging. The complexities that arise within such transformed loops should be analyzed with respect to vectorization and other considerations.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) (GZ: LU 1353/11-1). We also thank the 'Regionales Rechenzentrum Erlangen' (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) for the access to their Broadwell nodes, and NEC Deutschland for the access to their Aurora nodes. We also thank Prof. John Thuburn – University of Exeter, for his help to develop the code of the shallow water equations.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Vincenzo Casulli. 1990. Semi-implicit finite difference methods for the two-dimensional shallow water equations. *J. Comput. Phys.* 86, 1 (1990), 56–74.

[2] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International.* IEEE, 676–687.

[3] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* IEEE Press, 4.

[4] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. 2011. Data layout transformation for stencil computations on short-vector simd architectures. In *International Conference on Compiler Construction.* Springer, 225–245.

[5] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing.* ACM, 13–24.

[6] Nabeeh Jumah and Julian Kunkel. 2018. Performance Portability of Earth System Models with User-Controlled GGDML code Translation. In *High Performance Computing (Lecture Notes in Computer Science).* Springer. https://doi.org/10.1007/978-3-030-02465-9$_5$0

[7] Nabeeh Jumah, Julian M Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Thomas Meurdesoif. 2017. GGDML: icosahedral models language extensions. *Journal of Computer Science Technology Updates* 4, 1 (2017), 1–10.

[8] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, 1–12.

[9] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on.* IEEE, 207–216.

[10] Charles Yount. 2015. Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on.* IEEE, 865–870.

[11] 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 *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2016 Sixth International Workshop on.* IEEE, 30–39.

[12] Gerhard Zumbusch. 2012. Vectorized higher order finite difference kernels. In *International Workshop on Applied Parallel Computing.* Springer, 343–357.