Vectorization of Riemann solvers for the single- and multi-layer shallow water equations

Chaulio R. Ferreira
Department of Informatics
Technical University of Munich
Munich, Germany
chaulio.ferreira@tum.de

Kyle T. Mandli
Applied Physics and Applied Mathematics Department
Columbia University
New York, USA
kyle.mandli@columbia.edu

Michael Bader
Department of Informatics
Technical University of Munich
Munich, Germany
bader@in.tum.de

***Abstract*—We discuss vectorization of normal and transverse Riemann solvers for the single- and multi-layer shallow water equations. Our approach is simple and portable, as it is based on auto-vectorization by the compiler, aided by OpenMP 4.0 directives. Despite the high complexity of the solver routines, the Intel Fortran Compiler proved itself able to successfully vectorize loops containing calls to these solvers, after only a few small changes in their code. We evaluate the performance of the vectorized Riemann solvers within the context of GeoClaw, a software designed for simulation of geophysical flows with finite volume methods. Our performance studies consider two platforms with different sets of SIMD instructions: a dual-socket Haswell system with the AVX2 instruction set (256-bit) and an Intel Xeon Phi (Knights Landing) with AVX-512 instructions (512-bit). The experimental results indicate performance improvements of up to 2.1x on the former platform and up to 6.5x on the latter (with double-precision arithmetic). We also show that these speedups can easily compensate for the overhead introduced by the rearrangement of the simulation data structures, which might be necessary to achieve efficient vectorization.**

***Index Terms*—parallel processing, vectorization, Riemann solvers, shallow water equations**

I. Introduction

The width of SIMD instructions supported by modern high performance architectures has increased remarkably over the past few years. While a decade ago processors would typically provide 128-bit vector instructions, nowadays 256-bit instructions are widely supported, and even 512-bit instructions are available on the latest architectures, like Intel's Many Integrated Core (MIC) and some Skylake Xeon models. Since these processors are able to operate on 16 single-precision (or 8 double-precision) values with a single instruction, the benefits of vectorization can no longer be ignored for high performance applications.

In this paper, we evaluate the benefits of vectorization on the numerical routines of GeoClaw, a software designed for simulation of depth-averaged geophysical flows. Specifically, we focus on vectorization of normal and transverse Riemann solvers for the single- and multi-layer variations of the shallow water equations. Since the solution for Riemann problems can be directly applied in finite volume or discontinuous Galerkin methods, the solver routines are essential for computational simulations using these equations. Also, they are usually responsible for the majority of the computational costs in such applications, which makes it especially important to have highly efficient solvers.

Our implementations rely on *auto-vectorization* by the compiler, aided by the `!$OMP SIMD` compiler directive provided by the OpenMP standard 4.0 and later versions. As such, our codes are portable across various platforms with different SIMD instruction sets. Considering that, we performed experiments on two different architectures: a dual-socket Haswell system with the AVX2 instruction set (256-bit) and an Intel Xeon Phi processor (Knights Landing), which supports the AVX-512 instruction set (512-bit). On the former, we experienced performance improvements in our double-precision solvers ranging from 1.3× to 2.1×, while on the latter the observed improvements range from 2.4× to 6.5×.

We discuss some related work in Section II and describe our numerical model and finite volumes approach in Sections III and IV. In Section V, we describe the GeoClaw framework and the changes in the code necessary to achieve vectorization of its Riemann solvers. Finally, we present the results of our performance experiments in Section VI and conclude in Section VII by summarizing our findings.

II. Related work

Vectorization has been successfully applied to Riemann solvers for the single-layer shallow water equations in previous work. For instance, Bader et. al. [1] present vectorized implementations for the f-Wave [2] and augmented Riemann [3] solvers. Although auto-vectorization was possible for their f-Wave solver, *intrinsics* functions were necessary to achieve vectorization of the augmented Riemann solver, because the compiler was not able to auto-vectorize the loop annotated with `#pragma simd`.

Another example is the work described in [4], where the authors present an auto-vectorized implementation for both solvers. However, their approach is based on obtaining multiple instructions in the solver vectorized independently from each other instead of using a single vectorized loop that solves multiple instances of Riemann problems. The disadvantage of their approach is that each vectorized loop introduces some

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
The final authenticated version is available online at `https://ieeexplore.ieee.org/document/8514378`.

overhead, limiting the performance improvement. Thus, it is desired to have as few loops as possible.

In contrast to these work, this paper presents a simpler auto-vectorized implementation of the augmented Riemann solver based on a single loop annotated with the `!$OMP SIMD` directive (available in the OpenMP 4.0 standard and later versions). With this directive, which was still quite recent when [1] was published, the Intel Compiler is able to successfully auto-vectorize a loop that solves multiple Riemann problems using the augmented Riemann solver, despite its complexity.

In addition to these differences in the vectorization approach, in this paper we also propose and evaluate novel vectorized implementations of normal and transverse Riemann solvers for the *multi-layer* shallow water equations (based on the original implementations proposed in [5]). Although other authors [6] have worked on a GPGPU implementation for the multi-layer shallow water equations, this is, to the best of our knowledge, the first paper reporting successful vectorization of Riemann solvers for these equations.

III. The Shallow Water Equations

The *shallow water equations* (SWEs) are depth-averaged equations that are suitable for modeling incompressible fluids in problems where the horizontal scales (x and y dimensions) are much larger than the vertical scale (z dimension) and the vertical acceleration is negligible. Simulations are often based on the *single-layer* SWEs. They take the form

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghb_x \\ -ghb_y \end{bmatrix} \tag{1}$$

where $h(x, y, t)$ is the fluid depth; $u(x, y, t)$ and $v(x, y, t)$ are the vertically averaged fluid velocities in the x and y directions, respectively; g is the gravitational constant; and $b(x, y)$ is the bottom surface elevation. In oceanic applications, b is usually relative to mean sea level and corresponds to submarine bathymetry where $b < 0$ and to terrain topography where $b > 0$. Here, the source term $\psi(x, y, t) = [0, -ghb_x, -ghb_y]^T$ models the effect of the varying topography but it can be modified to include other terms, such as drag and Coriolis forces.

These equations are appropriate for modelling various wave propagation phenomena such as tsunamis and dam breaks. However, they lack accuracy for problems where significant vertical variations in the water column can be observed. Consider for instance storm surge simulations [7] where wind stress plays a crucial role and affects more the top of the water column than the bottom. The single-layer SWEs are not able to properly model this effect as the water momentum is averaged along the vertical dimension.

Such vertical variations can be properly modeled using the *multi-layer SWEs*, which provide more realistic representations while keeping the computational costs relatively low. In the case of the mentioned storm surge simulations, one can obtain a more accurate representation of the wind effects by modeling the sea with two layers: a shallow top layer over a deeper bottom layer. This can be accomplished with the *two-layer SWEs*, which have the form shown in (2) on the following page. In these equations, $h_1(x, y, t)$ is the water depth of the first (top) layer, $h_2(x, y, t)$ is the depth of the second (bottom) layer, and so on; also, ρ_1 and ρ_2 are the densities of the fluid contained in each layer.

Although the multi-layer equations can be generalized for an arbitrary number of vertical layers, this paper focuses only on the single-layer and two-layer SWEs discussed above.

IV. Numerical Scheme

For our simulations, we use GeoClaw [8], a software package that implements finite volume methods for depth-averaged equations, allowing the simulation of various geophysical flow phenomena. GeoClaw is based on the more general package Clawpack [9], which provides a framework for solving systems of hyperbolic conservation laws with the general form (in two dimensions):

$$q_t + f(q)_x + g(q)_y = \psi(q, x, y), \tag{3}$$

where $q(x, y, t)$ is the vector of unknowns. This framework allows the implementation of Godunov-type methods, which are based on the solution of so-called *Riemann problems*. This numerical approach has been thoroughly covered in [10], [11] and will be briefly described here.

We use a finite volume discretization that represents the simulation domain as logically rectangular grids (usually latitude-longitude or Cartesian) where the solution variables q are averaged within each cell. With this discretization, Riemann problems appear naturally at the interfaces between cells (i.e., edges). A Riemann problem is an initial-value problem with piecewise constant data and a single discontinuity at some point $x = \bar{x}$:

$$q(x) = \begin{cases} q_\ell & \text{if } x < \bar{x} \\ q_r & \text{if } x > \bar{x} \end{cases} \tag{4}$$

The solutions for these problems are sets of shock and rarefaction waves that propagate at constant speeds over the solution domain and are used to compute the numerical fluxes between neighbor cells and to update the solution for the following time step.

In addition to the *normal* Riemann problems described above, our numerical scheme also solves an extra set of *transverse* Riemann problems. In two-dimensional problems, "transverse" waves (waves that do not move perpendicularly to the edge) actually affect cells other than the two ones adjacent to the crossed edge, but this effect is not properly captured by the normal problems alone. By introducing this additional step, such waves are properly handled and the method's accuracy is improved, as the true two-dimensional problem is being solved. This approach is based on the corner-transport upwind method and is described in Chapters 20–21 of [11]. Also, second-order correction and limiters are applied

$$\begin{bmatrix} \rho_1 h_1 \\ \rho_1 h_1 u_1 \\ \rho_1 h_1 v_1 \\ \rho_2 h_2 \\ \rho_2 h_2 u_2 \\ \rho_2 h_2 v_2 \end{bmatrix}_t + \begin{bmatrix} \rho_1 h_1 u_1 \\ \rho_1 h_1 u_1^2 + \frac{1}{2} g \rho_1 h_1^2 \\ \rho_1 h_1 u_1 v_1 \\ \rho_2 h_2 u_2 \\ \rho_2 h_2 u_2 + \frac{1}{2} g \rho_2 h_2^2 + g \rho_1 h_2 h_1 \\ \rho_2 h_2 u_2 v_2 \end{bmatrix}_x + \begin{bmatrix} \rho_1 h_1 v_1 \\ \rho_1 h_1 u_1 v_1 \\ \rho_1 h_1 v_1^2 + \frac{1}{2} g \rho_1 h_1^2 \\ \rho_2 h_2 v_2 \\ \rho_2 h_2 u_2 v_2 \\ \rho_2 h_2 v_2^2 + \frac{1}{2} g \rho_2 h_2^2 + g \rho_1 h_2 h_1 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -\rho_1 g h_1 (h_2)_x - \rho_1 g h_1 b_x \\ -\rho_1 g h_1 (h_2)_x - \rho_1 g h_1 b_x \\ 0 \\ \rho_1 g h_1 (h_2)_x - \rho_2 g h_2 b_x \\ \rho_1 g h_1 (h_2)_x - \rho_2 g h_2 b_x \end{bmatrix} \quad (2)$$

to the solutions in order to improve the method's accuracy. This is covered in Section 4.1 of [10] and Section 6.2 of [11].

While it is possible to compute the exact solution for the Riemann problems, in most cases this is too expensive and not worth the effort for the accuracy, as these exact solutions are used in an inexact discrete numerical model. Instead, our simulations use so-called "approximate Riemann solvers" that deliver faster, yet reliable solutions. For the single-layer SWEs GeoClaw implements the Augmented Riemann solver [3] and for the two-layer SWEs the solver proposed in [5] is used. Although these approximate solvers are considerably fast, they are still usually responsible for the majority of the computational costs in the simulations and are, thus, the focus of our optimization efforts in this paper.

V. Implementation details

GeoClaw provides an efficient framework for implementing the Godunov-type finite volume methods described in the previous section, including shared-memory parallelism (OpenMP) and adaptive mesh refinement (AMR). In this section, we discuss how these strategies and the numerical approach are implemented in GeoClaw. More detailed descriptions can be found in [8], [10]. Here we also describe the changes that were necessary to successfully vectorize the Riemann solvers.

A. Adaptive mesh refinement

AMR is essential for large-scale simulations where high resolution is necessary at specific regions, but not over the entire domain – oceanic simulations are a a typical example. Dynamically adaptive meshes are able to considerably reduce the simulation's computational costs (with respect to both time and space) while maintaining its accuracy [10].

GeoClaw implements a hierarchical patch-based strategy for AMR. The entire simulation domain is comprised of a coarse Level 1 grid, whose cells may be flagged for refinement depending on various user-defined criteria. Every few time steps, a regridding procedure is performed, identifying clusters of spatially-close flagged cells and defining rectangular grid patches that contain these clusters. These patches are then refined according to refinement ratios that are also user-defined, producing a set of Level 2 grids. This process can be recursively repeated up to an arbitrary number of levels, where the refinement ratios of each level can be defined independently. As an example, consider the simulation with three AMR layers illustrated in Fig. 3. More details regarding the AMR algorithms in GeoClaw can be found in [10], [12].

B. Shared-memory parallelism

Parallelism in GeoClaw is also based on the concept of patches – each patch can be processed almost independently, needing only to exchange boundary data with its neighbor patches via an additional ghost layer. The data used for the ghost layer comes either from neighbor patches in the same AMR level or from the coarser grid that contains the patch (in this case, interpolation methods are usually necessary). After exchanging boundary data, patches in the same AMR level can then be trivially processed in parallel.

An important detail in GeoClaw's implementation is that we impose a limit in the dimensions of each patch, such that multiple patches are applied where a single patch might otherwise be enough. In the current implementation, each patch is limited to 60×60 cells. This does not only lead to more parallel work (due to the larger number of patches), but also improves spatial locality by producing smaller patches that are more likely to fit in the cache.

C. Numerical time step

As described in Section IV, a simulation time step is completed by solving the Riemann problems that appear at all edges in the grid and using their solutions to update the cell quantities accordingly. The Riemann problems are identified and their input data are organized by two separate loops for the vertical and horizontal edges, which process one-dimensional "slices" of a patch, each direction at a time. For each slice, we extract the data from two adjacent rows (or columns) of cells, and organize them into temporary arrays that represent a list of Riemann problems and are given as input to the Riemann solvers. After solving the Riemann problems, their solution is then used to compute (and accumulate) the numerical fluxes going in and out of each cell, which are afterwards used to update all cells, finalizing the time step.

The fact that our implementation organizes the Riemann problems into lists (arrays) that are passed to the solvers is very relevant when considering vectorization for the solvers' routines. Since each solver already has to deal with multiple problems that are independent from each other, no major changes in these algorithms have been necessary to obtain such a scheme, which is usually necessary for vectorization.

D. Changes in the data layout

The arrays used in GeoClaw and Clawpack's default implementation are organized in an Arrays of Structs (AoS) fashion that is not suitable for vectorization. In other words,

all components of a cell's data (e.g., h, hu and hv for the single-layer model) are contiguous in memory while the same component of neighbor cells are not. However, vectorization requires unit stride access to the same component of different cells because we loop through multiple Riemann problems that have these components as input data. Thus, it would be necessary to change the data layout to Structs of Arrays (SoA), in order to sucessfully apply vectorization.

Unfortunately, modifying the data layout of the entire framework was not practical, not only because of its complexity, but also because GeoClaw is part of the Clawpack distribution, which includes various other applications and concerns several simultaneous projects that would also be affected by such changes. Instead, we decided to use the appropriate data layout only in the routines that compute the numerical fluxes (where we wish to apply vectorization), leaving other parts of the code unchanged.

In practice, the data in each patch are organized in two-dimensional matrices with an AoS fashion but we rearrange them on-the-fly when extracting one-dimensional slices of data. After this step, the temporary arrays given as input to the Riemann solvers have a SoA layout, and are thus suitable for vectorization.

Although this additional data rearrangement introduces an overhead in the algorithm due to strided accesses to the data, it should also be noticed that strided accesses were already performed previously, when extracting slices of data either in the x or y direction. Also, the performance results presented in Section VI show that the benefits from vectorization easily outweigh this overhead.

E. Vectorization of the Riemann solvers

To make auto-vectorization possible, we organized the code in the Riemann solvers' subroutines and added compiler directives to them. All computations necessary to solve a single instance of a Riemann problem were encapsulated into a separate subroutine, and the loop code was simplified to the bare minimum – loading the problem data from the arrays, calling the solver subroutine, and storing the computed solutions into the appropriate arrays – see the example shown in Fig. 1. Note that, while this example is based specifically on the normal Riemann solver for the single-layer equations, the same code structure was also used for the transverse solver, as well as for the multi-layer solvers.

With respect to the SoA vs. AoS discussion in the previous section, notice that the arrays containing the input data for the Riemann problems are organized as SoA (see lines 5–12), which results in unit stride accesses between the same component for consecutive Riemann problems. In the original implementation they were organized as AoS, which required strided accesses and inhibited vectorization. The same can be said about the output arrays (lines 19–20), whose dimensions have been reordered such that now the left-most index is used to identify the Riemann problems. Because Fortran uses column-major ordering, this ordering also results in contiguous acesses for the same output component of different Riemann

```
1  !$OMP SIMD PRIVATE(hL,hR,huL,huR,hvL,hvR,bL,bR,sw,fw)
2  DO i=1,num_problems ! iterate through all problems
3
4    ! copy input parameters from input arrays:
5    hL  = hL_list(i)
6    hR  = hR_list(i)
7    huL = huL_list(i)
8    huR = huR_list(i)
9    hvL = hvL_list(i)
10   hvR = hvR_list(i)
11   bL  = bL_list(i)
12   bR  = bR_list(i)
13
14   !DIR$ FORCEINLINE
15   CALL solve_problem(hL,hR,huL,huR,hvL,hvR,bL,bR,&
16          sw,fw) ! output: sw(3) and fw(3,3)
17
18   ! store solution in output arrays:
19   sw_list(i,:)   = sw(:)
20   fw_list(i,:,:) = fw(:,:)
21 END DO
```

Fig. 1. Example of how the main loop in the solvers' subroutines has been organized to allow auto-vectorization assisted by compiler directives. This code snippet has been slightly modified from the actual implementation to make it more comprehensible. This example was based on the normal Riemann solver for the single-layer SWEs – the code same structure was also used for the transverse solver, as well as for the multi-layer solvers.

problems – e.g., `sw(i,1)` is contiguous to `sw(i+1,1)`. Thus, this new ordering used on the output arrays is also suitable for vectorization.

Also, notice that we use the OpenMP 4.0 directive `!$OMP SIMD`, and not the alternative `!DIR$ SIMD`, since the latter was unable to vectorize this loop. It was also necessary to declare the iteration-local variables (used as input and output parameters for the subroutine `solve_problem`) as `PRIVATE`, otherwise the compiler would have refused to vectorize the loop. This was one of the main motivations for encapsulating the code into a subroutine, as this minimizes the number of variables that need to be declared here.

We use `!DIR$ FORCEINLINE` to make sure that the subroutine gets inlined into the loop and, thus, also vectorized. Instead of inlining, an alternative option would be to declare it as "SIMD-enabled" with the compiler directive `!$OMP DECLARE SIMD`. However, in our experiments this approach resulted in considerably slower performance, thus this alternative was discarded.

To make the solver subroutines auto-vectorizable as well, a few minor changes in their code were also necessary. In particular, it was necessary to remove early exits from the subroutine – consider the example in Fig. 2. In the original code, Riemann problems identified as "trivial" (both cells completely dry) were simply skipped, as their solution is always zero. However, skipping iterations is not possible with vectorization, as the SIMD model requires that the computation for sets of contiguous arrays positions advance simultaneously. In our implementation, these problems are replaced on-the-fly with artificial problems that are then solved as usual. Afterwards, the real solution for these problems (zero) is attributed to their respective output variables. Because this approach uses simple pre- and post-processing of the input and output variables, it delivers better performance than introducing an extra if-

```fortran
! Identify completely dry problems
if (hL <= dry_tol .and. hR <= dry_tol) then
  ! The original code would have returned here,
  ! but that is not allowed in a vectorized code.
  ! Instead, we solve an artificial problem:
  hL = 0.0
  hR = 0.0
endif

! ...
!   Use the Riemann solver to compute the
!   solution arrays sw(3) and fw(3,3)
! ...

! For dry-dry problems, solution is always zero!
if (hL < dry_tol .and. hR < dry_tol) then
    sw(:) = 0.0
    fw(:,:) = 0.0
endif
```

Fig. 2. Changes in the solvers' code necessary to remove early exits and allow their vectorization. Previously, all "dry-dry" problems (where the column of water is smaller than the threshold `dry_tol` in both cells) were skipped, but that is not allowed when using vectorization. In the new vectorizable code, we solve these problems normally, and their real solution (zero) is attributed afterwards. Notice that this last step is necessary because the solutions computed by the solvers are usually incorrect, because they assume that $h \neq 0$ in both cells.

then-else branch that would force the compiler to use masked operations and assignments.

F. Problems with the compiler

Unfortunately, the executable generated by the Intel Fortran Compilers (versions 16, 17 and 18) for codes like the example shown in Fig. 1 actually produced incorrect simulation results. After some experiments, we noticed that the compilers were not able to correctly handle the output arrays `sw` and `fw`, because they are declared as `PRIVATE` for each loop iteration. We also noticed that, when we replaced these arrays with scalar variables (e.g., using three variables `sw1`, `sw2`, and `sw3` instead of the array `sw(3)`), the program would perform correctly. Thus, we used this workaround in our implementations. Note that, for simplicity and clarity, we chose to show the implementation with `PRIVATE` arrays in the example in Fig. 1, instead of the implementation we actually used where these are replaced with many scalar variables.

Another alternative to avoid these problems would have been to use SIMD-enabled subroutines instead of inlining, as discussed above. This approach was correctly handled by the compiler but, again, it was discarded due to its lower performance.

The issues described above have been investigated by Intel engineers, who reported that the problem was identified and will be fixed in the next major compiler version (Intel Compiler 2019).

VI. Performance Results

A. Experimental platforms

We conducted experiments on the CoolMUC2 and CoolMUC3 cluster systems hosted at the Leibniz Supercomputing Center (LRZ). The former cluster contains nodes with dual-socket Haswell systems, while the latter provides nodes with Xeon Phi "Knights Landing" (KNL) processors. An overview of the system configuration of the nodes in each system is presented in Table I. With respect to this paper, the most important difference between these architectures is the width of the SIMD vector instructions they provide: the Haswell processors use the AVX2 instruction set (256-bit), while the KNL processor uses the AVX-512 instruction set (512-bit). As such, the benefits from vectorization are expected to be more noticeable on the KNL machine.

TABLE I
Specifications of the experimental platforms.

System overview	Haswells	Knights Landing
Architecture	Intel® Xeon®	Intel® Xeon Phi™
Model	E5-2697v3	7210F
Cores	2x14	64
Clock rate	2.60 GHz	1.30 GHz
SIMD vector width	256-bit	512-bit
Memory	64 GB	96 GB
Peak bandwidth	136 GB/s	102 GB/s
Peak performance (DP)	582 GFlop/s	2 662 GFlops/s
Measured perform. (DP)	152 GFlop/s	699 GFlops/s

In Table I, we also list both the theoretical peak performance and the measured performance for double precision operations in floating-point operations per second (Flop/s). The former was calculated based on the specifications of each system, while the latter was obtained with a simple benchmark proposed in Chapter 2 of [13]. This benchmark measures the Flop/s rate in a trivial loop that repeatedly computes $a[i] \leftarrow a[i] * k + b[i]$ on all elements of small (cache-fitting) arrays a and b. It was designed with the goal of obtaining the highest attainable performance in architectures like the ones we use in the paper (which support multi-threading, vectorization and fused multiply-add operations, e.g.), and its results serve as a good estimate for the maximum performance that can be achieved in practice.

In all experiments reported here we used the Intel Fortran compiler 17.0.4 and double precision arithmetic. On both systems we use all available cores, i.e., 28 on the Haswells and 64 on the KNL. On the KNL we also experiment with different number of threads per core (from 1 to 4). However, here we only list the results with 4 threads per core (i.e., 256 in total), because this configuration achieved the best performance in most experiments.

B. Simulation scenarios

1) Chile 2010 scenario: To test our single-layer SWEs implementation, we simulate a real tsunami event that took place in the Pacific Ocean in February 2010 and reached the coast of Chile and Peru – see Fig 3. We used topography data obtained from the ETOPO2 data set available at [14] together with initial water displacement based on simulations performed with the Okada model [15] and USGS earthquake data [16]. The interested reader is referred to [10] for more details and numerical considerations regarding the simulation of the Chile 2010 Tsunami.

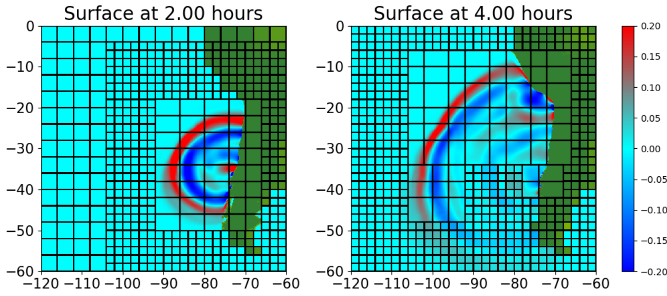

Fig. 3. Simulation of the Chile 2010 tsunami with GeoClaw. In this simulation, we used a Level 1 grid with 15×15 cells and refinement factors of 2×2 and 8×8 for the second and third levels, respectively. Note that the internal edges of the third level grids are omitted for clarity.

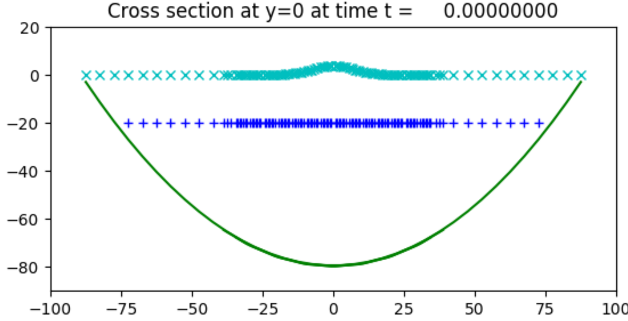

Fig. 4. Cross-cut of the "bowl-radial" scenario at $t = 0$. The green line depicts the bowl-shaped bathymetry and the cross and plus signs show the elevation of the first and second layers of water, respectively. Note that the cell concentration is much higher in the region close to the hump due to AMR.

For the performance results presented in the following we used three levels of AMR: a Level 1 grid composed of 100×100 cells and refinement factors of 6×6 and 8×8 for Levels 2 and 3 respectively. This gives a cell resolution of $\frac{1}{80}$ degrees (roughly 1.4km) on the finest level. Our results consider a simulation of the first six hours after the earthquake that generated the tsunami. In each simulation, roughly 14.1 billion cell updates were computed.

2) Parabolic bowl-shaped ocean : For the experiments with the multi-layer SWEs, we used an artificial 2D scenario where waves generated by a circular hump of water propagate over a parabolic bowl-shaped bathymetry. Consider Fig. 4, where we show a vertical section of this scenario to illustrate its initial conditions. In these simulations we also used three levels of AMR, and approximately 4.1 billion cell updates were computed.

C. Solver performance

Since our goal is to evaluate the effectiveness of vectorization, initially we consider only the performance of each solver individually; the overall simulation performance will be addressed later. Thus, in the following results we used the execution times spent by each solver to compute their performance – given as *Riemann problems solved per second*.

In Fig. 5 we present the performance of the normal and transverse Riemann solvers used in our single- and multi-layer

TABLE II
PERFORMANCE OF THE VECTORIZED SOLVERS.

Equations	Machine	Solver	Speedup	GFlop/s
Single-layer	Haswells	Normal	2.1×	66.6
		Transverse	1.7×	53.4
	KNL	Normal	6.5×	150.7
		Transverse	5.2×	121.0
Multi-layer	Haswells	Normal	1.4×	33.9
		Transverse	1.3×	36.4
	KNL	Normal	2.4×	31.2
		Transverse	2.8×	46.4

implementations. The benefits from vectorization are clear, especially on the KNL (as expected), where speedups ranging from 2.3× to 6.5× were achieved. Although the Haswells also experience considerable improvements (1.3× to 2.1×), now they are outperformed by the KNL in three of the four solvers.

The speedups obtained are listed in Table II, where we also list estimates of the floating-point operations per second rate (GFlop/s) for each solver and each machine, obtained after measuring the total number of operations performed by the solvers in our simulations with the PAPI interface [17]. Comparing these values with the maximum performance measured in each machine (see Table I), we observe that the single-layer solvers achieve roughly 35–44% of the attainable performance on the Haswells and 17–22% on the KNL. Since the solvers are much more complex than the benchmark used (including if-then-else branches, division and square-root operations, etc.), these numbers indicate good utilization of the computational resources.

A similar analysis can be made for the multi-layer solvers, which achieve 22–24% and 4–7% of the attainable performance on the Haswells and the KNL, respectively. It is evident that these solvers benefit less from vectorization than the single-layer ones and perform considerably slower. This can be attributed to the even higher complexity of the multi-layer solvers (in comparison with the single-layer ones), where large if-then-else branches are necessary to deal with the four times higher number of dry/wet combinations (e.g., two-layer problems with one completely dry layer are actually handled by single-layer solvers). Because such branches can diminish the benefits from vectorization substantially, the smaller speedups and worse performance obtained are not surprising. Also, these results indicate that the vectorized multi-layer solvers might be improved by minimizing the number and size of the branches in their code. However, it is not clear whether such modifications are possible and investigating this is left as a suggestion for future work.

D. Simulation performance

Now, we consider the influence of vectorization to the simulation as a whole. In Fig. 6 we plot the execution (wall) times taken for the entire simulations using the serial and the vectorized solvers. We divide the execution time into four components: "Normal solver", "Transverse solver", "Other numerical routines" and "Grid management". The first

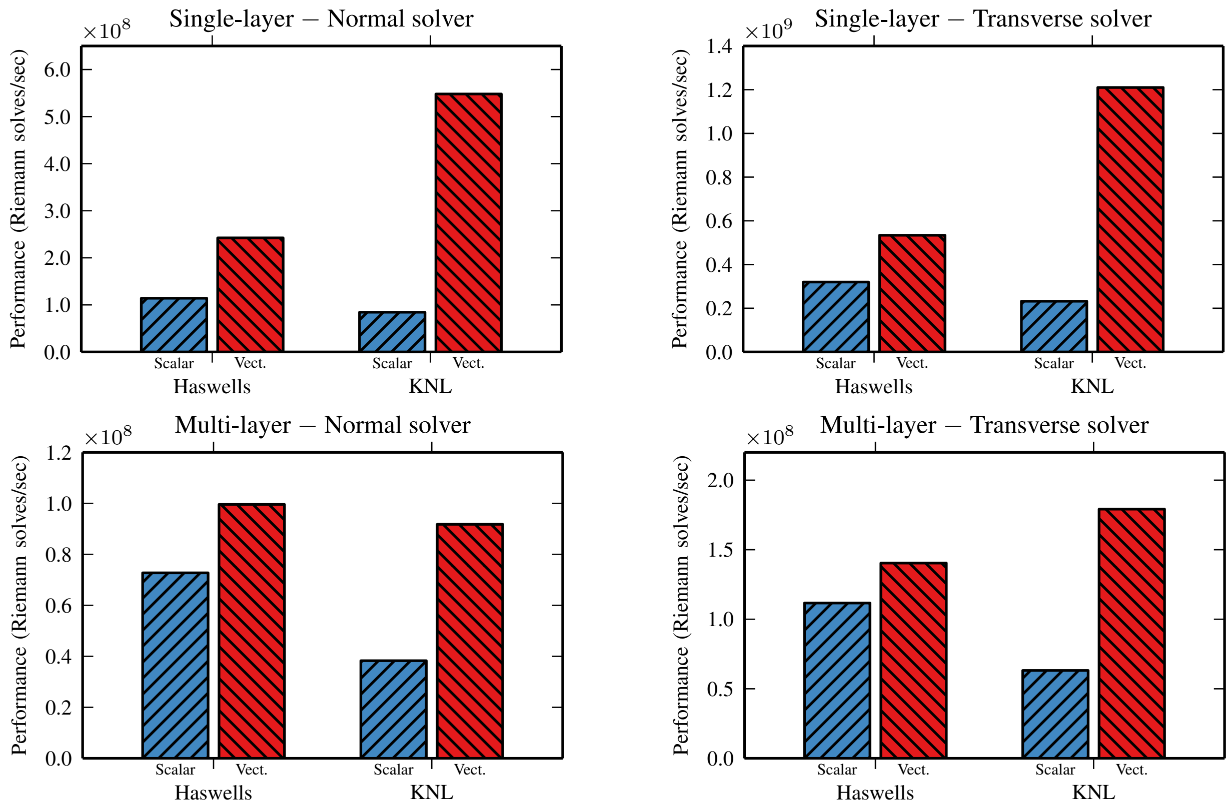

Fig. 5. Performance of the single- and multi-layer Riemann solvers, before and after applying vectorization.

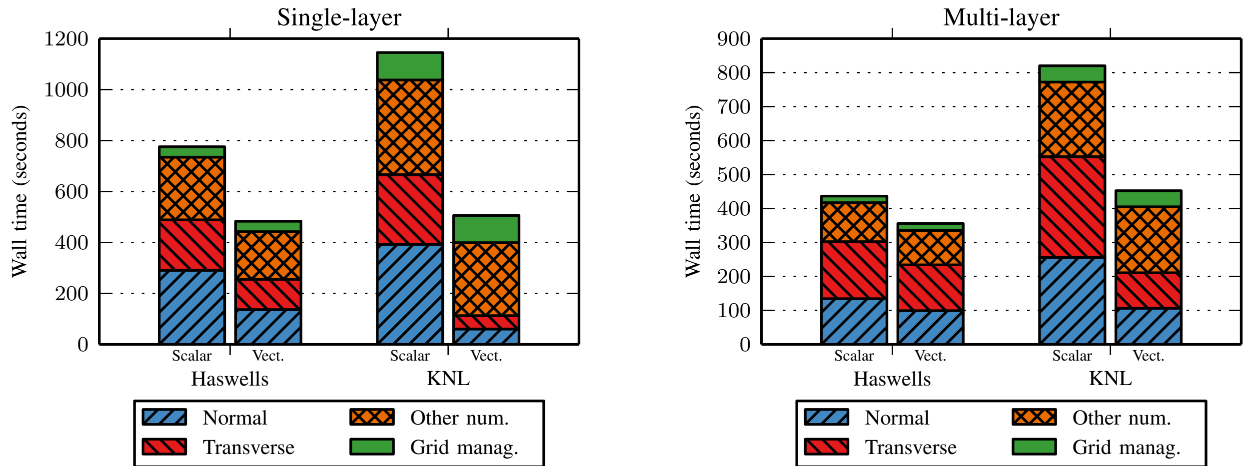

Fig. 6. Wall time split into components for the entire simulations, before and after vectorization of the Riemann solvers.

two correspond to the Riemann solvers discussed previously. "Other numerical routines" includes all execution time necessary to update the solution values, except for the two Riemann solvers. These routines are basically composed of the steps that precede and succeed the Riemann solvers. Initially, they are responsible for fetching the cell data and reorganizing them as one-dimensional slices to be used as input for the solvers. Afterwards, they apply second-order corrections and limiters to the Riemann solutions and use them to update the cell quantities. Lastly, "Grid management", comprises the operations necessary for AMR, such as refining/coarsening cells, merging contiguous patches, etc.

The overall speedups are also much higher on the KNL (2.3× and 1.8× for the single- and multi-layer equations, respectively) than on the Haswells (1.6× and 1.2×). However, the execution times for the vectorized codes are roughly the same on both architectures, especially for the single-layer equations. This happens despite the fact that the vectorized single-layer solvers perform considerably faster on the KNL. This is due to the other two components, which are executed

much faster on the Haswells than on the KNL. Particularly, the "Other numerical routines" component now consumes most of the execution time on the KNL. As described above, this component is the one responsible for fetching the grid data from the main memory. Compared to the Riemann solvers, the number of floating-point operations performed in this step is much smaller. This suggests that the simulation code has now become memory-bound, as often happens after vectorizing the most compute-intensive routines of an application.

It should be noted, however, that the execution times for this component did not increase on the vectorized versions, despite the overhead introduced by rearranging the arrays from AoS to SoA during every time step (as discussed in Section V-D). In fact, there is a small time reduction due to the auto-vectorization of some instructions in this component (made possible by the new data layout). This clearly shows that the benefits of vectorization easily outweigh this overhead, especially when also considering the performance improvements of the Riemann solvers. Nevertheless, it is still expected that modifying the data layout for the entire application might be beneficial for its performance.

Clearly, the "Grid management" component is also responsible for a considerable fraction of the execution time on the KNL. This happens mainly because this component does not scale well on hundreds of threads, as is necessary for high performance with the KNL. As such, future efforts on optimizing these applications for the KNL (or MIC architectures in general) should consider ways to improve scalability and parallel efficiency for this component.

VII. Conclusions

In this paper, we proposed and evaluated vectorized implementations of normal and transverse Riemann solvers for the single- and multi-layer shallow water equations. Compared to previous related work, our approach is much simpler and more portable, as it uses auto-vectorization guided by an OpenMP compiler directive that proved itself able to successfully vectorize highly complex loops.

We experienced substantial speedups on the solver routines, especially on the modern KNL processor, which provides a set of 512-bit SIMD instructions. As expected, smaller but still considerable improvements were also reported for the Haswell architecture, able to perform 256-bit instructions. However, although the solvers perform faster on the KNL, the overall execution time is still comparable on both machines.

A component analysis of the execution times showed that the benefits of vectorization clearly compensate for the overhead introduced to make the simulation data structures suitable for SIMD instructions. It also revealed that, while the compute-intensive Riemann solvers used to be the most time-consuming routines, now most of the time is spent on other more memory-intensive components of the simulation code. These results suggest that future optimizations attempts of GeoClaw should focus on these components, instead of the Riemann solvers that have already been addressed in our work.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant no. DMS-1720288. We acknowledge computing resources from the Leibniz Supercomputing Center and from Columbia University's Shared Research Computing Facility project, which is supported by NIH Research Facility Improvement Grant 1G20RR030893-01, and associated funds from the NYSTAR Contract C090171. The authors thank the TUM Graduate School for its financial support for a three-month research stay of Chaulio R. Ferreira at Columbia University. Chaulio R. Ferreira also appreciates the support of CNPq, the Brazilian Council of Technological and Scientific Development (grant no. 234439/2014-9).

References

[1] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger, "Vectorization of an Augmented Riemann Solver for the Shallow Water Equations," in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 2014, pp. 193–201.

[2] D. S. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith, "A wave propagation method for conservation laws and balance laws with spatially varying flux functions," *SIAM Journal on Scientific Computing*, vol. 24, no. 3, pp. 955–978, 2002.

[3] D. L. George, "Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation," *Journal of Computational Physics*, vol. 227, no. 6, pp. 3089–3113, 2008.

[4] C. R. Ferreira and M. Bader, "Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors," in *Proceedings of the Platform for Advanced Scientific Computing Conference*. ACM, 2017, p. 12.

[5] K. T. Mandli, "A Numerical Method for the Two Layer Shallow Water Equations with Dry States," *Ocean Modelling*, vol. 72, pp. 80–91, 2013.

[6] M. Lastra, M. J. C. Díaz, C. Ureña, and M. de la Asunción, "Efficient multilayer shallow-water simulation system based on gpus," *Mathematics and Computers in Simulation*, 2017.

[7] K. T. Mandli and C. N. Dawson, "Adaptive mesh refinement for storm surge," *Ocean Modelling*, vol. 75, pp. 36–50, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500314000031

[8] M. J. Berger, D. L. George, R. J. LeVeque, and K. T. Mandli, "The GeoClaw software for depth-averaged flows with adaptive refinement," *Advances in Water Resources*, vol. 34, no. 9, pp. 1195–1206, 2011.

[9] K. T. Mandli, A. J. Ahmadia, M. Berger, D. Calhoun, D. L. George, Y. Hadjimichael, D. I. Ketcheson, G. I. Lemoine, and R. J. LeVeque, "Clawpack: building an open source ecosystem for solving hyperbolic PDEs," *PeerJ Computer Science*, vol. 2, p. e68, 2016.

[10] R. J. LeVeque, D. L. George, and M. J. Berger, "Tsunami modelling with adaptively refined finite volume methods," *Acta Numerica*, vol. 20, pp. 211–289, 2011.

[11] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002. [Online]. Available: http://www.clawpack.org/book.html

[12] A. Schwarzschild and K. T. Mandli, "An implementation of adaptive mesh refinement for shallow water equations," *arXiv preprint arXiv:1803.01450*, 2018.

[13] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[14] National Centers for Environmental Information, "ETOPO2v2 2-minute Worldwide Bathymetry/Topography Grids," www.ngdc.noaa.gov/mgg/global/etopo2.html, accessed 2018-03-15.

[15] Y. Okada, "Surface deformation due to shear and tensile faults in a half-space," *Bulletin of the seismological society of America*, vol. 75, no. 4, pp. 1135–1154, 1985.

[16] U.S. Geological Survey, "USGS Earthquake Hazards Program," https://earthquake.usgs.gov/, accessed 2018-03-15.

[17] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.