Vectorization and GPGPU-Acceleration of an augmented Riemann solver for the shallow water equations

Wolfgang Hölzl
Vectorization and GPGPU-Acceleration of an augmented Riemann solver for the shallow water equations

Vektorisierung und GPGPU-Beschleunigung eines erweiterten Riemann-Lösers für die Flachwasser-Gleichungen

Author: Wolfgang Hölzl
Supervisor: Univ.-Prof. Dr. Michael Bader
Advisor: Dipl.-Math. Alexander Breuer
Advisor: Sebastian Rettenberger, M. Sc.
Date: July 15th, 2013
I assure the single handed composition of this bachelor’s thesis only supported by declared resources.

Munich, July 15th, 2013

Wolfgang Holzl
# Contents

## Abstract

<table>
<thead>
<tr>
<th>I</th>
<th>Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-A</td>
<td>Conservation Laws</td>
<td>1</td>
</tr>
<tr>
<td>I-B</td>
<td>Characteristics</td>
<td>1</td>
</tr>
<tr>
<td>I-C</td>
<td>Shallow Water Equations</td>
<td>1</td>
</tr>
<tr>
<td>I-D</td>
<td>Approximation With Finite Volumes</td>
<td>2</td>
</tr>
<tr>
<td>I-E</td>
<td>Riemann Problems</td>
<td>2</td>
</tr>
<tr>
<td>I-F</td>
<td>Computations In A Single Time Step</td>
<td>3</td>
</tr>
<tr>
<td>I-G</td>
<td>Augmented Riemann Solver</td>
<td>3</td>
</tr>
<tr>
<td>I-H</td>
<td>Riemann Structures</td>
<td>5</td>
</tr>
<tr>
<td>I-I</td>
<td>SWE Teaching Code</td>
<td>6</td>
</tr>
</tbody>
</table>

## Vectorization

<table>
<thead>
<tr>
<th>II</th>
<th>Vectorization</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>II-A</td>
<td>Basics of Vectorization</td>
<td>8</td>
</tr>
<tr>
<td>II-B</td>
<td>When Auto-Vectorization Fails</td>
<td>8</td>
</tr>
<tr>
<td>II-C</td>
<td>Vectorization By Hand</td>
<td>9</td>
</tr>
<tr>
<td>II-D</td>
<td>Variable Vector-Extensions</td>
<td>9</td>
</tr>
<tr>
<td>II-E</td>
<td>Macros Instead of Intrinsics</td>
<td>11</td>
</tr>
<tr>
<td>II-F</td>
<td>Handling Divergent Branches</td>
<td>12</td>
</tr>
<tr>
<td>II-G</td>
<td>Working With Sign Bits</td>
<td>12</td>
</tr>
<tr>
<td>II-H</td>
<td>Additional Changes To The Solver</td>
<td>13</td>
</tr>
<tr>
<td>II-I</td>
<td>Performance</td>
<td>14</td>
</tr>
<tr>
<td>II-J</td>
<td>Known Issues</td>
<td>15</td>
</tr>
</tbody>
</table>

## CUDA

<table>
<thead>
<tr>
<th>III</th>
<th>CUDA</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>III-A</td>
<td>Basics</td>
<td>16</td>
</tr>
<tr>
<td>III-B</td>
<td>Kernels</td>
<td>16</td>
</tr>
<tr>
<td>III-C</td>
<td>Thread Divergence</td>
<td>17</td>
</tr>
<tr>
<td>III-D</td>
<td>Implementation In The SWE Teaching Code</td>
<td>17</td>
</tr>
<tr>
<td>III-E</td>
<td>Performance</td>
<td>18</td>
</tr>
</tbody>
</table>

## Outlook – Vectorization On The Intel MIC Architecture

<table>
<thead>
<tr>
<th>IV</th>
<th>Outlook – Vectorization On The Intel MIC Architecture</th>
<th>20</th>
</tr>
</thead>
</table>

## Conclusion

<table>
<thead>
<tr>
<th>V</th>
<th>Conclusion</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>V-A</td>
<td>Vectorization On IA-32 and AMD 64</td>
<td>21</td>
</tr>
<tr>
<td>V-B</td>
<td>CUDA</td>
<td>21</td>
</tr>
<tr>
<td>V-C</td>
<td>Vectorization On Intel MIC</td>
<td>21</td>
</tr>
<tr>
<td>V-D</td>
<td>Getting The Code</td>
<td>21</td>
</tr>
</tbody>
</table>

## References

| References | 22 |
List of Figures

I.1 Components of the Shallow Water Equations in 1D ................................................. 2
I.2 Approximation to the solution via cell averages ......................................................... 3
I.3 Riemann problems at the edges ..................................................................................... 3
I.4 Approximation to the solution via cell averages—next time step ............................... 4
I.5 Propagation of the tsunami caused by the 2011 Tohoku earthquake ........................ 5
I.6 Propagation of the tsunami caused by the 2010 Chile earthquake ............................... 6
II.1 Performance of the vectorized augmented Riemann solver ........................................ 14
III.1 Grid consisting of thread blocks .................................................................................... 16
III.2 Performance of the CUDA version of the augmented Riemann solver ..................... 19

List of Listings

II-1 Non-vectorized loop ....................................................................................................... 8
II-2 Sketch of a vectorized loop ............................................................................................ 8
II-3 Possible vector-dependent code - simple example ....................................................... 8
II-4 Possible vector-dependent code - more subtle example .............................................. 8
II-5 Pragma-directives for the Intel compiler ........................................................................ 9
II-6 Probably not auto-vectorizable code ............................................................................. 9
II-7 Uncountable loop – impossible to vectorize due to the unknown length ..................... 9
II-8 Loop, vectorized with SSE-Intrinsics, assuming variables are pointers to float ............. 10
II-9 Summary of SIMD_TYPES.hpp .................................................................................... 11
II-10 Final version of the vectorized loop .......................................................................... 11
II-11 Outline of SIMD_DEFINITIONS.hpp ........................................................................ 12
II-12 Diverging branch within scalar code .......................................................................... 12
II-13 Diverging branch handled with masking .................................................................... 12
II-14 Diverging branch handled with blending ................................................................... 12
II-15 Handling of the different branches in computeMiddleStates() ................................. 13
II-16 Blending at the end of member functions to avoid unintentional changes ............... 13
III-1 Example of calling a CUDA kernel .............................................................................. 16
III-2 CUDA kernel for Listing II-1 ...................................................................................... 17
III-3 Computing the absolute position of a thread within a grid ....................................... 17
III-4 Divergent branch from Listing II-12 with CUDA ...................................................... 18
III-5 Switching the solvers within the SWE teaching code ................................................ 18
IV-1 Masked addition with MIC-intrinsics ........................................................................... 20

List of Tables

II.1 Number of components within a single vector, depending on vector extension and precision 10
II.2 Mapping of the Riemann structures to integers ............................................................. 13
II.3 Performance of the vectorized augmented Riemann solver .......................................... 14
III.1 Performance of the CUDA version of the augmented Riemann solver ...................... 19
Abstract

This work presents results of the vectorization and CUDA parallelization of an augmented Riemann solver for the shallow water equations over variable topography with steady states and inundation. The underlying solver was presented by David L. George in [1] and scalar versions are implemented in ClawPack [2] and in the SWE Teaching Code [3], developed at TUM.

The vectorization is done with SSE 4.1 and AVX 1 intrinsics and able to be adapted to wider vector registers in the future. The thesis explains, how this vectorization has been accomplished, with special view on the handling of divergent branches within the control flow. Implementation details are presented, including the macro based approach, which eases the adaption to wider vector units provided by future architectures.

Furthermore, the solver has been implemented with CUDA, to take advantage of the parallelization capacities of modern GPGPUs from NVIDIA.

Performance results are given. They show speedups in the range of the achievable peak as well as the limits of Sandy Bridge CPUs with respect to AVX.

At the end, an outline is given how to adapt the vectorization on the Intel MIC architecture.
I. Introduction

**Tsunami** is Japanese and means literally harbor-wave. It describes a series of water waves that are able to propagate long distances over the ocean.

Tsunamis are typically caused by massive displacements of water, as in most cases generated by earthquakes in subduction zones or volcanic eruptions. Often tsunamis cause natural disasters, such as the 2004 **Indian Ocean tsunami** which claimed the lives of more than 200,000 people. Another historical disaster was the tsunami caused by the 2011 **Tōhoku earthquake**. The meltdown of the Fukushima Daiichi nuclear power plant was caused by the earthquake as well as the following tsunami.

At the moment it is common to lookup precomputed simulations in a database. However, a tsunami can take several hours to arrive at the far-field coast. In the future it might be possible to do live-simulations to predetermine the effects of the wave at the shorelines.

Therefore it is necessary to provide precise data of the oceanic bathymetry and the displacements, and also numerical methods to compute approximations to the real solution.

In this section the mathematical background is briefly described. This includes the formulation of the **Shallow Water Equations** and solution of linear **Riemann problems** as well as the **augmented Riemann solver** developed by David L. George in [1]. **Section II** proposes a SIMD vectorization which finally accelerates the solver by orders of magnitude up to a factor of 5. **Section III** deals with a parallelization on NVIDIA’s CUDA platform. The last two sections answer the question how the underlying code can be adapted to future CPUs.

A. Conservation Laws

The general prototype of a hyperbolic problem is

$$q_t(x, t) + f(q(x, t))_x = \psi(q(x, t)),$$  \hspace{1cm} (I.1)

where $q \in \mathbb{R}^m$ is the vector of unknowns, the function $f(q) : \mathbb{R}^m \to \mathbb{R}^m$ is the **flux** and the right-hand side $\psi(q(x, t)) \in \mathbb{R}^m$ is the **source term**. The **quasi-linear** form is given by

$$q_t(x, t) + A(q) q_x(x, t) = \psi(q(x, t)),$$  \hspace{1cm} (I.2)

where $A(q) \in \mathbb{R}^{m \times m}$ is the Jacobian $f'(q)$.

In order for this system to be hyperbolic, the matrix $A$ must have real eigenvalues and a set of $m$ linearly independent eigenvectors [4].

An important class of homogeneous hyperbolic problems (i.e. $\psi(q) \equiv 0$) are the so-called **conservation laws**. A natural view to conservation laws is an integral form. It models the fact that the system in $[x_1, x_2]$ changes only due to the contribution of the fluxes at the boundaries of the interval:

$$\frac{\partial}{\partial t} \int_{x_1}^{x_2} q(x, t) \, dx = - f(q(x, t)) \bigg|_{x=x_2}^{x_1},$$  \hspace{1cm} (I.3)

If the solution is smooth enough, the conservation law can be expressed in a differential form like Equation (I.2).

B. Characteristics

The function

$$q_t(x, t) = q(x - \bar{u} t, 0)$$  \hspace{1cm} (I.4)

is the solution of the **advection equation**

$$q_t + \bar{u} q_x = 0.$$  \hspace{1cm} (I.5)

This solution is called a **characteristic** and models, that the initial data $q(x, 0)$ propagates along the characteristic curve

$$X(t) := x_0 + \bar{u} t$$

with speed $\bar{u}$ [4].

For linear hyperbolic systems the matrix $A$ is diagonalizable with $m$ linearly independent eigenvectors $v^p$ and eigenvalues $\lambda^p \neq 0$, thus

$$A = R \Lambda R^{-1},$$  \hspace{1cm} (I.6)

where $\Lambda$ is a diagonal matrix. With $w = R^{-1} q$ it follows

$$w_t + \Lambda w_x = 0$$  \hspace{1cm} (I.7)

which reads for the $p^{th}$ equation as

$$w^p_t + \lambda^p w^p_x = 0.$$  \hspace{1cm} (I.8)

The system decomposes to $p$ advection equations, each with the solution

$$w^p(x, t) = w^p(x - \lambda t, 0).$$  \hspace{1cm} (I.9)

Equation (I.9) reflects the fact, that the initial data $w^p(x, 0)$ of the $p^{th}$ component propagates with speed $\lambda^p$ along the $p^{th}$ characteristic curve.

It follows for the solution $q(x, t)$

$$q(x, t) = R w(x, t)$$

$$= \sum_{p=1}^{m} w^p(x, t) v^p$$  \hspace{1cm} (I.10)

This transformation is correct only if the decomposition in Equation (I.6) exists. If the matrix $A$ depends on $x$ or $t$, the eigenvectors and -values are space and time dependent as well and have to be determined depending on $(x, t)$.

C. Shallow Water Equations

The **Shallow Water Equations (SWEs)** are a set of hyperbolic differential equations that can be used to model the propagation of tsunami waves under certain assumptions such as deep water and shallow waves in relation to the wave length [4, p. 254ff]. A short derivation as well as the physical background is given in [5, p. 8ff]. With a source term and in one dimension the SWEs take the form

$$\begin{bmatrix} h \\
        hu \\
        hu^2 + \frac{1}{2}gh^2 
\end{bmatrix}_x = \begin{bmatrix} 0 \\
        -ghb_s \end{bmatrix},$$  \hspace{1cm} (I.11)
where $g$ is the gravitational constant, $h(x, t)$ denotes the depth of the fluid, $u(x, t)$ the vertical averaged horizontal fluid velocity and $b(x)$ the bathymetry [1], as shown in Figure I.1.

The Jacobian of the flux function is given by

$$f'(q) = \begin{bmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{bmatrix}$$

with eigenvectors and -values

$$\{r^\pm, \lambda^\pm\} = \left\{ \left( 1, u \pm \sqrt{gh} \right)^T, u \pm \sqrt{gh} \right\}.$$ (I.13)

In two dimensions the equations take the form

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \begin{bmatrix} hu & hu & hu \\ hu & h^2 + \frac{1}{2} gh^2 & hv \\ hv & hv & h^2 + \frac{1}{2} gh^2 \end{bmatrix} \begin{bmatrix} x \\ t \end{bmatrix} = \begin{bmatrix} 0 \\ gh_{x} \\ -gh_{y} \end{bmatrix}.$$ (I.14)

This section considers the homogeneous case in 1D only. The 2D equations can be reduced to solving 1D problems in both, x and y direction [1].

D. Approximation With Finite Volumes

For the first thoughts, the source term is skipped, such that the equations become homogeneous. With

$$q(x, t) := \begin{bmatrix} h \\ hu \end{bmatrix} \text{ and } f(q(x, t)) := \begin{bmatrix} hu \\ h^2 + \frac{1}{2} gh^2 \end{bmatrix}$$

Equation (I.11) can be reformulated in integral form

$$\frac{\partial}{\partial t} \Omega \int q(x, t) \, dx + f(q(x, t)) \Big|_{x^1}^{x^2} = 0$$

where $\Omega$ denotes the spatial domain $[x^1, x^2]$. Another integral form comes up, when integrating Equation (I.14) over time:

$$\int_{x^1}^{x^2} q(x, t) \, dx + \int_{t_1}^{t_2} f(q(x, t)) \, dt = 0$$

The solution to these equations is approximated via a finite volume approach. Therefore the domain $\Omega$ is discretized in cells with length $\Delta x$ and the time is discretized in steps of length $\Delta t$. The spatial integrals of Equation (I.15) over these cells are approximated for each time step. The $i^{th}$ grid cell is written as

$$C_i := (x_{i-1/2}, x_{i+1/2}).$$

The value of $q$ at time $t^n$ is approximated by the average of the integral over the cell:

$$Q_i^n \approx \frac{1}{\Delta x} \int_{C_i} q(x, t^n) \, dx.$$ (I.16)

A Taylor expansion shows, that for smooth $q$ the error is of magnitude $O((\Delta x)^2)$ (compare [4, p. 64]). Additionally, the usage of cell averages makes it easier to derive numerical methods [4, p. 64].

Equation (I.16) leads with Equation (I.15) to

$$Q_i^{n+1} = Q_i^n - \frac{1}{\Delta t} \int_{t^n}^{t^{n+1}} (f(q(x_{i+1/2}, t)) - f(q(x_{i-1/2}, t))) \, dt.$$ (I.17)

The combination of Equation (I.18) and Equation (I.17) gives

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x} \left( F_{i+1/2} - F_{i-1/2} \right).$$ (I.19)

The numerical solution to this problem is reduced to the computation of $F_{i+1/2}$.

E. Riemann Problems

Consider the hyperbolic problem in Equation (1.1) with initial data, that is piecewise constant with an initial discontinuity:

$$q(x, 0) = \begin{cases} Q_i & x < x_{i+1/2} \\ Q_{i+1} & x > x_{i+1/2} \end{cases}$$ (I.20)

This discontinuity propagates along the characteristics. Thus, the solution to the problem is

$$q(x, t) = \begin{cases} Q_i & x - \lambda t < x_{i+1/2} \\ Q_{i+1} & x - \lambda t > x_{i+1/2} \end{cases}$$ (I.21)

where $\lambda$ denotes the speed of the characteristic. In the case of a linear system of $m$ equations, the initial discontinuity can be decomposed into

$$Q_{i+1} - Q_i = \sum_{p=1}^{m} \alpha_{i+1/2}^p Q_{i+1/2}$$ (I.22)

with linearly independent vectors $Q_{i+1/2}$ and uniquely determined coefficients $\alpha_{i+1/2}^p$. From now on, these vectors are referred to as WAVES.
So, Equation (1.22) can be seen as decomposing the discontinuity into the waves \( W_{1+i/2}' \) each with a constant wave speed \( p_{1+i/2}' \).

Recall that, in the case of the 1D shallow water equations, the data \( Q_i \) and thus the waves \( W_{1+i/2}' \) denote both a vector with two components—the height in the first and the momentum in the second field.

Splitting the sum in right- and the left-going waves, one is able to define the Net-Updates or fluctuations from cell \( C_i \) and \( C_{i+1} \) according to [1]:

\[
\begin{align*}
A^+ \Delta Q^n_{1+i/2} &= \sum_{p: s^n_{1+i/2} > 0} s^n_{1+i/2} W^n_{1+i/2} \\
A^- \Delta Q^n_{1+i/2} &= \sum_{p: s^n_{1+i/2} < 0} s^n_{1+i/2} W^n_{1+i/2}
\end{align*}
\tag{I.23}
\]

The term \( A^- \Delta Q^n_{1+i/2} \) describes the leftward moving waves from the (right) cell interface \( x_{i+1/2} \) into cell \( C_i \). \( A^+ \Delta Q^n_{1+i/2} \) denotes the rightward moving waves from the (now left) cell interface \( x_{i+1/2} \) from cell \( C_i \) into cell \( C_{i+1} \). An illustration of this can be found in Section I-F.

It is convenient to use the \( p^n \) EIGENPAIR \( \{ \lambda^n_{1+i/2}, r^n_{1+i/2} \} \) of the so-called Roe average \( \hat{A}_{1+i/2} \), [1], which satisfies per definition

\[
\lambda^n_{1+i/2}(Q_{i+1} - Q_i) = f(Q_{i+1}) - f(Q_i).
\tag{I.24}
\]

Usage of these eigenpairs produces the exact solution, if the true Riemann solution consists of a single shock wave [1, p.3092]. This leads to the Roe solver.

The jump in the flux can also be decomposed in

\[
f(Q_{i+1}) - f(Q_i) = \sum_{p=1}^{m} \beta^n_{1+i/2} r^n_{1+i/2}
\tag{I.25}
\]

With the Roe-eigenpairs it can be shown that [1, p.3093]

\[
\begin{align*}
A^- \Delta Q^n_{1+i/2} &= \sum_{p: s^n_{1+i/2} > 0} \beta^n_{1+i/2} Q^n_{1+i/2} \\
A^+ \Delta Q^n_{1+i/2} &= \sum_{p: s^n_{1+i/2} < 0} \beta^n_{1+i/2} Q^n_{1+i/2}
\end{align*}
\tag{I.26}
\]

Note that Equation (I.23) and Equation (I.26) are different methods. Equation (I.23) describes the fluctuations as a decomposition of the jump in the solution \( Q_{1+\pm1/2} \). Equation (I.26), however, describes the fluctuations as a decomposition of the jump in the flux \( F_{1+\pm1/2} \).

Equation (I.26) can be used to approximate the values of \( F^n_{1+\pm1/2} \) in Equation (I.19). This results in the well-known F-wave method where the waves carry units of flux to determine the cell update (see [1, p.3093])

\[
Q^{n+1} = Q^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q^n_{1-1/2} + A^- \Delta Q^n_{1+1/2} \right).
\tag{I.27}
\]

The method is first order accurate. It is possible, to extend the method to be second order accurate in the absence of shock waves. This is done by adding correction terms, as described in detail in [1] and [4].

F. Computations In A Single Time Step

This section provides an outline of the computations done in a single time step.

Figure I.2 shows the approximation of the true solution via piecewise constants (finite volumes). Recall that the approximation \( Q_i \) denotes a vector.

![Figure I.2](image1)

Figure I.2. First step: Approximation of the solution at time \( t^n \) via piecewise constants

Riemann problems arise at every edge between two cells. In order to solve these problems, the jump is decomposed into vectors or respectively waves. These waves are sketched as dashed vectors in Figure I.3.

![Figure I.3](image2)

Figure I.3. Second step: solving Riemann problems at the edges

Note that this figure shows already the augmented Riemann solver described in the following. The f-wave and the Roe solver work only with two waves instead of three. They do not consider the water height.

The leftgoing fluctuations \( A^- \Delta Q^n_{1+1/2} \) at the cell interface \( x_{i+1/2} \) are colored in magenta. The rightgoing fluctuations \( A^+ \Delta Q^n_{1+1/2} \) are colored in yellow.

After solving the Riemann problems, the net-updates can be computed to get an approximation at the next time step. An example for this approximation is shown in Figure I.4.

G. Augmented Riemann Solver

The f-wave method described above has a fundamental drawback [1]. Since the depth is not included in the decomposition, it is not guaranteed to preserve DEPTH-NEGATIVITY, which is important in the context of inundation. If inundation is to be simulated, a drying out cell might lose more water due to the flux, than there is water in the cell. This results in a negative water height.
The \( f \)-wave method is a CONSERVATIVE METHOD with any set of vectors \( r_{i+1,j}^p \) since it decomposes the flux. A conservative method is also obtained, if Equation (I.23) is based on the Roe solver. Recall that Equation (I.23) decomposes the jump in the solution rather than the jump in the flux.

Here conservative means, that changes in a cell are only due to the contribution of the flux at the cell edges.

However, the Roe solver fails with preserving depth-non-negativity [1]. Other solvers of the so-called HLL-type are able to handle the depth-non-negativity. They are not based on a decomposition into eigenvectors \( r_{i+1,j}^p \) like Equation (I.10). On the contrary, HLL-type solvers, such as the HLLE solver, require other modifications in order to work properly [1, p. 3094].

The augmented solver, which is subject of this thesis uses a combination of the \( f \)-wave and the HLLE method. This combination makes the solver conservative and preserves depth-non-negativity at the same time.

The solver relies on a decomposition with more than \( m \) waves. Recall that \( m = 2 \) in the 1D-case.

1) The Homogeneous Case

First consider

\[
\left[ \begin{array}{c}
\Delta Q_{i+1} - Q_i \\
f(Q_{i+1}) - f(Q_i)
\end{array} \right] = \sum_{p=1}^{2m} \alpha_{i+1}^p P_{i+1/2}^p
\]

which decomposes the jumps in the solution and in the flux. However, the second component of the solution and the first component of the flux are identical—the momentum \( hu \).

With \( Q_i = (H_i, Hu_i)^T \) as approximation for \( q = (h, hu)^T \) and \( \phi(q) = (hu^2 + 1/2gh) \), (compare Equation (I.11) and [1]) the decomposition becomes

\[
\left[ \begin{array}{c}
H_{i+1} - H_i \\
Hu_{i+1} - Hu_i \\
\phi(Q_{i+1}) - \phi(Q_i)
\end{array} \right] = \sum_{p=1}^{3} \alpha_{i+1}^p W_{i+1/2}^p
\]

Note that this describes a \((3 \times 1)\)-vector. The vectors \( W_{i+1/2}^p \) are prechosen and linearly independent.

To preserve conservation the updating fluctuations \( A^- \Delta Q_{i+1/2}^n \) are chosen to be based on the flux only (compare Equation (I.25)), thus

\[
\begin{bmatrix}
0 & 1 & 1 \\
0 & 1 & 1
\end{bmatrix} \alpha_{i+1/2}^p W_{i+1/2}^p
\]

and

\[
\begin{align*}
A^- \Delta Q_{i+1/2}^n &= \sum_{p:s_i+1/2 < 0}^P \xi_{i+1/2}^p \\
A^+ \Delta Q_{i+1/2}^n &= \sum_{p:s_i+1/2 > 0}^P \xi_{i+1/2}^p
\end{align*}
\]

This leads again to a \( f \)-wave-like method where the water height is included in the decomposition.

The first and the third vectors \( w_{i+1/2}^p \) and their speeds \( s_{i+1/2}^p \) are chosen to be

\[
\begin{align*}
\left\{ w_{i+1/2}^1, s_{i+1/2}^1 \right\} &= \left\{ (1, \xi_{i+1/2}^- (\xi_{i+1/2}^-)^2)^T, \xi_{i+1/2}^- \right\} \\
\left\{ w_{i+1/2}^3, s_{i+1/2}^3 \right\} &= \left\{ (1, \xi_{i+1/2}^+ (\xi_{i+1/2}^+)^2)^T, \xi_{i+1/2}^+ \right\}
\end{align*}
\]

with

\[
\begin{align*}
\bar{\xi}_{i+1/2}^- &= \min \left\{ \lambda^- (Q_i), \lambda_{i+1/2}^- \right\} \\
\bar{\xi}_{i+1/2}^+ &= \max \left\{ \lambda^+ (Q_{i+1}), \lambda_{i+1/2}^+ \right\}
\end{align*}
\]

where \( \lambda_{i+1/2}^\pm \) are the eigenvalues of the Roe averaged Jacobian \( A_{i+1/2} \) [1].

\( \lambda^\pm (Q_i) \) denote the eigenvalues of the SWE system matrix (see Equation (I.13)).

The speeds in Equation (I.33) are called Einfeldt speeds [1].

A possible choice for the second pair, which is also used in the implementation of the solver, is

\[
\left\{ w_{i+1/2}^2, s_{i+1/2}^2 \right\} := \left\{ (0, 0, 1)^T, \frac{1}{2} (\bar{\xi}_{i+1/2}^+ + \bar{\xi}_{i+1/2}^-) \right\}
\]

If Equation (I.34) is used and if only shock waves occur, the augmented solver corresponds to the Roe solver [1, p. 3096]. In [1, p. 3101ff] other choices are presented which prevent wrong solutions if strong rarefaction waves are present. The wave \( \bar{\xi}_{i+1/2}^\pm \) is called CORRECTOR WAVE [1].

It can be shown, that the choice of Equation (I.31) and Equation (I.34) leads to the height-update being equivalent to the HLLE-solver [1, p. 3096]. The HLLE solver as well as the solver in this thesis use the speeds in Equation (I.33) to define two discontinuities. They compute a MIDDLE-STATE. The HLLE-middle-state consists of the HLLE-MIDDLE-HEIGHT and the corresponding momentum to achieve NUMERICAL CONSERVATION [1, p. 3096]. The HLLE-middle-height is given by

\[
\bar{H}_{i+1/2}^\pm := \frac{Hu_{i+1} + s_{i+1/2}^\pm H_{i+1} - s_{i+1/2}^\pm H_i}{s_{i+1/2}^+ - s_{i+1/2}^-}
\]

Since \( \bar{H}_{i+1/2}^\pm \geq 0 \), the augmented solver preserves depth-non-negativity, and is conservative at the same time.
2) The Source Term

In the presence of a source term the computations become even more difficult and are outlined in the following.

It can be shown, that—analogue to Equation (I.29)—the inclusion of the source term leads to

\[
\begin{bmatrix}
  H_{i+1} - H_i \\
  HU_{i+1} - HU_i \\
  \phi (Q_{i+1}) - \phi (Q_i) \\
  B_{i+1} - B_i
\end{bmatrix}
= \sum_{p=1}^{4} \alpha_{i+1/2}^P w_{i+1/2}^P.
\] (I.36)

with \(B_i\) as the approximation of the bottom surface in the cell \(C_i\). The fluctuations are computed analogue to Equation (I.30) by choosing the second and third component.

Note that Equation (I.36) can be seen as a solver for a non-linear homogeneous problem as described by Equation (I.1) that uses four waves. It can be shown, that the shallow water equations in Equation (I.11) can be written as such a system as

\[
\dot{q}_t + W(\dot{q}) \dot{q}_x = 0
\] (I.37)

with

\[
\dot{q} := \begin{bmatrix}
  h \\
  h\dot{u} \\
  \phi
\end{bmatrix}
\] (I.38)

and

\[
W(\dot{q}) := \begin{bmatrix}
  0 & 1 & 0 & 0 \\
  -u^2 + gh & 2u & 0 & gh \\
  0 & -u^2 + gh & 2u & 2ugh
\end{bmatrix}.
\] (I.39)

Three eigenvectors are identical to the vectors \(w_{i+1/2}^P\) with \(p = 1, \ldots, 3\), which were introduced previously. The fourth component equals to \(0\) [1]. The matrix has an eigenvalue \(\lambda^0 = 0\), too. This introduces a stationary or steady-state wave because

\[
\dot{q}_t + W(\dot{q}) \dot{q}_x = 0
\] (I.40)

can have solutions with \(\dot{q}_x \neq 0\) and \(\dot{q}_t = 0\). It follows that \(\dot{q}\) is proportional to the eigenvector \(r^0(\dot{q})\).

The vector \(w_{i+1/2}^0\) is then chosen as an approximation to \(r^0(\dot{q})\). For the details see [1].

The stationary wave corresponds to a jump discontinuity in the Riemann solution that acts as a source term. The wave is not included in the updates \(A^\pm \Delta Q^i_{i+1/2}\) since it does not move to the neighboring cells. For that reason it is important to preserve the steady-state wave. Thus, the decomposition can be written as

\[
\begin{bmatrix}
  H_{i+1} - H_i \\
  HU_{i+1} - HU_i \\
  \phi (Q_{i+1}) - \phi (Q_i) \\
  B_{i+1} - B_i
\end{bmatrix}
= \sum_{p=1}^{4} \alpha_{i+1/2}^P w_{i+1/2}^P.
\] (I.41)

It can be shown, that this decomposition corresponds to subtracting an approximation of the source term from the momentum flux \(\phi(\dot{q})\).

It can further be shown, that the decision to use four waves in the decomposition as well as the choice of \(w_{i+1/2}^P\) make the solver able to preserve smooth steady-state waves over variable bathymetry even on coarser grids. It also preserves depth-non-negativity in the presence of a source term.

The proofs are a major topic of [1] and not further discussed in this thesis.

H. Riemann Structures

The approximate solution of the Riemann problem consists of four shock waves, where the first wave \(Z^1_{i+1/2}\) corresponds to the first field of the shallow water equations in Equation (I.11). The third wave \(Z^3_{i+1/2}\) is related to the second field.

It is possible to determine the structure of the exact solution of the Riemann problem via a function evaluation [1]. The structure belongs to one of four possible families. Either both fields have a shock wave, both have a rarefaction wave, or one field has a shock and the

![Figure I.5. Propagation of the tsunami caused by the 2011 Tōhoku earthquake on March 11th, 2011 — the earthquake and the tsunami caused 15,886 deaths [6] and resulted in the Fukushima Daiichi nuclear disaster. The bathymetry data stems from [7]; for the initial displacement data see [8]](image)
other a rarefaction wave. Finally, if one state is initially dry, only a rarefaction wave occurs [1].

Consider

$$\varphi(h, H_i) := \begin{cases} 2 \left( \sqrt{g H_i} - \sqrt{g h} \right) & h \leq H_i \\ (h - H_i) \left( \frac{\sqrt{g h}}{\sqrt{g h_i}} \right) & h > H_i \end{cases} \quad (I.42)$$

for \( h > 0 \) and

$$\Phi_{i+1/2}(h) := \varphi(h, H_{i+1}) + \varphi(h, H_i) + U_{i+1} - U_i \quad (I.43)$$

For the Riemann structure follows

$$\Phi_{i+1/2}(H_{\text{max}}) \leq 0 \quad \text{two shocks}$$

$$\Phi_{i+1/2}(H_{\text{min}}) < 0 < \Phi_{i+1/2}(H_{\text{max}}) \quad \text{shock, rarefaction}$$

$$0 \leq \Phi_{i+1/2}(H_{\text{min}}) \quad \text{two rarefactions} \quad (I.44)$$

with \( H_{\text{min}} := \min(H_i, H_{i+1}) \) and \( H_{\text{max}} := \max(H_i, H_{i+1}) \).

\( \Phi(h, H_i) \) is a monotonically increasing function, so it is sufficient to just compare the minimum or the maximum in the first and the third case.

Additionally, the middle-height \( H_{i+1/2}^* \) is the root of \( \Phi(h) \):

$$\Phi(H_{i+1/2}^*) = 0 \quad (I.45)$$

In the case of two rarefaction waves the root can be given directly as

$$H_{i+1/2}^* = \max \left\{ 0, \frac{U_i - U_{i+1} + 2 \left( \sqrt{g H_i} + \sqrt{g H_{i+1}} \right)^2}{16g} \right\}. \quad (I.46)$$

In the case of a single rarefaction the middle-height is set to 0. In the presence of a shock, the root of \( \Phi(h) \) must be determined numerically, e.g., via the Newton-Raphson method.

Once the middle-height is determined, the middle-state speeds can be computed as

$$\lambda^{-} \left( H_{i+1/2}^* \right) = U_i + 2 \sqrt{g H_i} - 3 \sqrt{g H_{i+1/2}^*} \quad (I.47)$$

$$\lambda^{+} \left( H_{i+1/2}^* \right) = U_{i+1} - 2 \sqrt{g H_{i+1}} + 3 \sqrt{g H_{i+1/2}^*}.$$  

Finally the speeds \( s^\pm_{i+1/2} \) can be determined. This leads directly to the vectors \( w^\pm_{i+1/2} \) (see [1]). Subsequently the linear system in Equation (I.41) can be solved to obtain the coefficients \( \alpha^\pm_{i+1/2} \).

The wave decomposition in Equation (I.41) enables the computation of the net-updates in Equation (I.26). For the details of the last steps see [1].

The solver is able to handle inundation. So it is necessary, to care about the wet-dry-state of the two cells. For details of the wet-dry-handling see [1, p. 3103].

### I. SWE Teaching Code

The SWE Teaching Code is developed at Technische Universität München and freely available at [3]. The code implements a finite-volume discretization and allows simulations of the shallow water equations over variable topography. The finite volume discretization is implemented as a two-dimensional array which holds the values for the cell averages.

A simulation consists of two major parts. First the net-updates are computed by solving Riemann problems at all edges between adjacent cells. For this task, several solvers are implemented, such as an f-wave method or the augmented Riemann solver described above. The solver class itself is an abstract class and defines for instance member variables for height and momentum and the pure virtual method `computeNetUpdates()`, which is concretely implemented by the different solvers.

The function `computeNetUpdates()` of the augmented Riemann solver first determines the wet-dry-state. If one of the cells is wet, the wave decomposition in Equation (I.41) is solved. This includes the determination of the Riemann structure according to Equation (I.44) and the computation of the middle states. The function `computeMiddleStates()` implements the Newton-Raphson method to find the root of Equation (I.45) in the case of a shock.

The function `updateUnknowns()` represents the second major part of the simulation. It computes the new
approximations of the cell averages in the next time step according to Equation (I.27).

The teaching code allows different ways of parallelization:

- **CUDA-parallelization of the f-wave method**—the CUDA-version of the augmented Riemann solver is subject of this thesis
- **OpenMP-parallelization of the loops**
- **MPI-parallelization based on a domain-decomposition**

This thesis introduces another level of parallelization based on vector registers. This is described in the next section.

The CUDA implementation of the solver is described in **Section III**.

Figure I.5 shows the propagation of the tsunami caused by the **2011 Tōhoku earthquake** and Figure I.6 the propagation of the tsunami caused by the **2010 Chile earthquake**. For the bathymetry grid data see [7]. For the initial displacement data see [8] (Tōhoku) and [9] (Chile).

The results are obtained by the SWE teaching code and the vectorized Riemann solver.
II. Vectorization

This section describes the vectorization of the Riemann solver. First the basics and the different kinds of vector extensions of modern CPUs are explained. Then an introduction is given into vectorization by hand via so-called **intrinsics**. Afterwards, the handling of divergent branches in the control flow is described—divergent branches are complicated to handle with this kind of **low-level parallelism**. At the end, achieved performance is presented.

A. Basics of Vectorization

One way to introduce core level parallelism into an application is the use of vector registers. This leads to a **SIMD** parallelization, which is easier to develop and to debug than other parallelization techniques, since there is no need for locking or any kind of synchronization.

SIMD stands for **S**ingle **I**nstruction **M**ultiple **D**ata and means that parallelization is introduced by doing one instruction on multiple data entries at the same time. That technique works only for scenarios, where the same instructions are applied over and over again to a row of continuous data, like for-loops that are traversing arrays (e.g. Listing II-1).

**Scalar code** loads two floating point numbers into multi-purpose registers (like RAX, RBX, ...) and stores the result back to memory.

**Vectorized code**, on the contrary, loads multiple floating point numbers in a row into special vector registers, computes multiple results within one instruction and then stores the result back to memory—again in a row.

```c
for (int i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
```

Listing II-1. Non-vectorized loop

Vectorization is automatically done by the compiler, if it detects the scenario described in Listing II-1. Therefore, the compiler splits the loop into packages of length L—say the loop gets **blocked**. These packages are processed simultaneously. Furthermore the compiler generates a **remainder handler** for left-over entries that do not fit into a vector.

The compiler will reformulate the non-vectorized loop in Listing II-1 similar to Listing II-2, if it detects a vectorization candidate.

The code in Listing II-2 introduces some overhead in the beginning (until line 6) for calculating a loop boundary which is a multiple of the length of the vector. The calculation of the boundary exploits the rules of integer division, where the quotient is rounded to the next smaller integer value. The function **VECTOR_ADD()** acts as a placeholder. This function is assumed to add L floating point elements at once. For a concrete implementation see Listing II-8.

```c
// usually, the vector length is 2, 4, 8
const int L = VECTOR_LENGTH;

// we need the greatest multiple of L,
// that is smaller than or equal to n
const int end_n_vector = (n / L) * L;
int i = 0;

// do L instructions at once
for (; i < end_n_vector; i += L)
    VECTOR_ADD(a, b, c, i, L);

// remainder handler for left over elements
for (; i < n; ++i)
    a[i] = b[i] + c[i];
```

Listing II-2. Sketch of a vectorized loop

Usually, the overhead for calculating the loop boundaries is negligible, compared to the achieved speed-up.

B. When Auto-Vectorization Fails

The compiler does not always apply auto-vectorization, for example if it is not able to guarantee, that there are no **vector-dependencies**, i.e. dependencies between elements that would fit in one vector. A simple example for this issue is taken from Intel’s documentation [10] and is shown in Listing II-3. A more subtle example is shown in Listing II-4.

```c
// simple vector dependency
for (int i = 0; i < n; ++i)
    a[i] = a[i + k] * c;
```

Listing II-3. Possible vector-dependent code - simple example

```c
// vector dependencies?
// what if called f(a, a, a)?
void f(double* a, double* b, double* c)
{
    for (int i = 1; i < n - 1; ++i)
        a[i] = b[i - 1] + c[i + 1];
}
```

Listing II-4. Possible vector-dependent code - more subtle example

Obviously, the first example may lead to incorrect code, if k < L. The second one needs a further explanation. What looks safely parallelizable could be called as f(a, a, a).

This issue is called **pointer-aliasing** and it depends on the programming language and on the compiler options, whether the demand for vectorization is rejected or not. In programming language Fortran, for example, pointers are generally treated not to alias. This could lead to incorrect code if the programmer is not aware of it.

In programming language C, pointers are usually treated to alias which prevents the compiler from
auto-vectorization. In this case, the programmer can work with pragmas (see below) or, since C99 with the restrict-keyword. If the pointers in Listing II-4 were declared with restrict, the compiler would treat them not to alias.

Finally, in programming language C++ there is no restrict-keyword. There is only the compiler flag -fstrict-aliasing telling the compiler to ignore aliasing for all pointers in the whole translation unit.

Other ways to make the compiler vectorize the code are special PRAGMAS. They tell the compiler to ignore vector dependencies or request a stronger demand on vectorization. Unfortunately, these pragmas are not standard and depend on the vendor of the compiler. Listing II-5 shows two ways to achieve vectorization with the Intel compiler. Again, the last one is taken directly from Intel’s documentation, see [10].

```c
// this yields the same as Listing II-2
#pragma simd
for (int i = 0; i < n; ++i)
  a[i] = b[i] + c[i];

// ...

// enables vectorization, but may lead to
// wrong code, if k < VECTOR_LENGTH
#pragma ivdep
for (int i = 0; i < m; ++i)
  a[i] = a[i + k] + c;
```

Listing II-5.Pragma-directives for the Intel compiler

The first pragma, #pragma simd, enforces vectorization of the loop. Using the pragma is not necessary here, but it might help with more complex loop bodies where the compiler’s auto-vectorizer is likely to fail. See [11] for more details.

The second one, #pragma ivdep, tells the compiler to ignore assumed vector dependencies due to pointer aliasing. Vectorization is refused, if and only if the compiler can prove a dependency [10]. Hence, this pragma is safer with respect to generating incorrect code.

In the SWE-Teaching-Code the updateUnknowns() function is vectorized with pragmas.

C. Vectorization By Hand

However, the compiler’s auto-vectorizer might fail—at least if there are complicated branches inside the loop, as shown in the—very artificial—example in Listing II-6.

The functions f(), g() and h() may be as complicated as the loop body itself. Issues arise as some vector components can take the first branch, some the second and some the third of the if-statement. Additionally, in the first branch some components can enforce to execute the break-statement, while others do not. The same happens in the second branch with the continue-statement.

```c
for (int i = 1; i < n - 1; ++i) {
  if (a[i] < b[i]) {
    a[i] = f(a[i], b[i - 1], c[i + 1]);
    if (a[i - 1] <= c[i - 1])
      break;
  } else if (a[i - 1] >= b[i + 1]) {
    c[i] = g(a[i], b[i - 1], b[i]);
    if (c[i] > b[i - 1])
      continue;
  } else {
    b[i] = h(a[i - 1], a[i], c[i + 1]);
  }
}
```

Listing II-6. Probably not auto-vectorizable code

But the compiler’s auto-vectorizer can also fail for simple examples like the uncountable loop in the—obfuscated and unsafe—version of STRING-COPY in Listing II-7. This example is directly taken from the GCC documentation [12].

```c
void strcpy (const char * src, char * dest)
{
  while (*dest++ = *src);
}
```

Listing II-7. Uncountable loop – impossible to vectorize due to the unknown length

In the augmented Riemann solver that is vectorized in this thesis, complicated branches occur, too. Depending on the wet-dry-state of the Riemann problem, different code paths have to be executed. Further on, depending on the Riemann structure, different Newton problems must be solved—maybe with a different number of iterations.

Here, the compiler’s auto-vectorizer sinks without trace. In this case, the code must be vectorized by hand, meaning the transformation from Listing II-1 to Listing II-2—shown for a simple for-loop—must be done by the programmer. The following steps are necessary:

- reformulation of the solver, to work with vector instead of scalar instructions
- loop-blocking to make the loop do L iterations in one step, where L is the length of the vector registers, such that all instructions in one step can be done by vector instructions
- introduction of a remainder handler, to compute the entities, which do not completely fit into a vector

While the last two tasks are straightforward as shown in Listing II-2, the first task denotes the major subject of this thesis and is explained in detail in the following.

D. Variable Vector-Extensions

The first question coming up with vectorization concerns the vector extensions. There are two notable exten-
sions, which differ basically in the width of the vectors. They differ also in some other details—such as three-operant instructions—which are not relevant for the optimizations performed in this thesis.

The first type of extensions is called SSE, the Streaming SIMD Extensions. These vector extensions were introduced by Intel in 1999 and have been extended until today up to version SSE 4.2. Each new version is backwards-compatible and introduces new instructions, while keeping the vector width constant at 16 bytes.

Up to SSE 3, the instructions were the same for Intel and AMD processors. After that version the development forked leading to incompatible instructions sets between the two vendors’ processors. The SSE version used for the vectorization in this thesis is Intel’s SSE 4.1, which was introduced with the Penryn-version of Intel’s Core architecture in 2008 and is also fully supported by AMD’s Bulldozer architecture introduced in 2011.

The second extensions are the Advanced Vector Extensions, or AVX. AVX 1 is available the first time with Intel’s Sandy Bridge processors in 2011 as well as with AMD’s Bulldozers. In contrary to SSE, the vector width has been extended to 32 Bytes.

The first version of AVX concentrates on floating point operations, whereas the second version AVX 2 completes AVX for integer operations. Due to the lack of AVX 2 capable hardware, this thesis uses AVX 1 for vectorization. Since some integer instructions are needed (such as comparison and shifting operations; see Section II-G), which first came up with AVX 2, these missing instructions were temporarily created in software. They can be replaced with native instructions, when running with AVX 2 capable processors.

When working with pragmas, the extensions to be used are indicated via compiler flags. These flags inform the compiler about the underlying machine architecture. For the GCC and Intel compiler, this is done via the –m options, such as –msse4 or –mavx.

Vectorizing by hand means to rewrite the floating point and integer operations in the scalar code with intrinsics. Intrinsics represent functions, that work with vector variables and correspond to the underlying machine instructions. The function VECTOR ADD () in Listing II-2 acts as such an intrinsic. In contrast to writing assembler code, intrinsics enable the programmer to work with variables of some vector type. The compiler cares about the machine instructions and the use of registers.

Another decision has to be taken concerning the precision. There are different instructions for single and double precision. This is because double precision numbers need twice as much bits as single precision, as shown in Table II.1.

Assuming the variables in Listing II-1 are pointers to single precision numbers, the code can be transformed to code as shown in Listing II-8.

First, the vector operands must be loaded via LOAD-intrinsics, that—in this example—loads four consecutive single precision values at the given position in memory into the vector variables. Vice versa, the result must be stored back.

Usually one would prefer AVX 1 as it results in the highest performance boost since it handles twice as many components as SSE 4.1. On the contrary, once the code is based completely on AVX 1, it must be rewritten when a better extension, working with larger vectors, is available.

In order to let the user decide between both, the floating point precision and the set of vector extensions, the code for this thesis was not written directly in intrinsics, but with macros that are mapped to intrinsics depending on the corresponding compiler flags. By doing so, it is possible to provide code using either the SSE 4.1 or the AVX 1 instruction set. Thus the code can easily be adapted to future extensions, that use for instance vectors with 512 bits or more, without writing and testing the frame for intrinsics again.

However, future extensions are supposed to behave in the same way with respect to masking and blendig (see Section II-F). As a counter example, this is not true for the vector instructions on the Intel MIC architecture. This issue is further discussed in Section IV.

<table>
<thead>
<tr>
<th>Bits per Register</th>
<th>SSE 4.1</th>
<th>AVX 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes per Register</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Number of registers</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

**Table II.1**

Number of components within a single vector, depending on vector extension and precision

const int end_n_vector = (n / 3) * 3;

```c
for (int i = 0; i < end_n_vector; i += 4) {
    // vector operand must be loaded first
    const __m128 op1 = _mm_load_ps(b + i);
    const __m128 op2 = _mm_load_ps(c + i);
    // results must be stored back
    _mm_store_ps(a + i, _mm_add_ps(op1, op2));
}
```

Listing II-8. Loop, vectorized with SSE-Intrinsics, assuming variables are pointers to float

For the GCC and Intel compiler, this is done via the _mavx_ extension and precision forked leading to incompatible instructions sets between the two vendors’ processors. The SSE version used for the vectorization in this thesis is Intel’s SSE 4.1, which was introduced with the Penryn-version of Intel’s Core architecture in 2008 and is also fully supported by AMD’s Bulldozer architecture introduced in 2011.

The second extensions are the Advanced Vector Extensions, or AVX. AVX 1 is available the first time with Intel’s Sandy Bridge processors in 2011 as well as with AMD’s Bulldozers. In contrary to SSE, the vector width has been extended to 32 Bytes.

The first version of AVX concentrates on floating point operations, whereas the second version AVX 2 completes AVX for integer operations. Due to the lack of AVX 2 capable hardware, this thesis uses AVX 1 for vectorization. Since some integer instructions are needed (such as comparison and shifting operations; see Section II-G), which first came up with AVX 2, these missing instructions were temporarily created in software. They can be replaced with native instructions, when running with AVX 2 capable processors.

When working with pragmas, the extensions to be used are indicated via compiler flags. These flags inform the compiler about the underlying machine architecture. For the GCC and Intel compiler, this is done via the _m_ options, such as _msse4_ or _mavx_.

Vectorizing by hand means to rewrite the floating point and integer operations in the scalar code with intrinsics. Intrinsics represent functions, that work with vector variables and correspond to the underlying machine instructions. The function VECTOR ADD () in Listing II-2 acts as such an intrinsic. In contrast to writing assembler code, intrinsics enable the programmer to work with variables of some vector type. The compiler cares about the machine instructions and the use of registers.

Another decision has to be taken concerning the precision. There are different instructions for single and double precision. This is because double precision numbers need twice as much bits as single precision, as shown in Table II.1.

Assuming the variables in Listing II-1 are pointers to single precision numbers, the code can be transformed to code as shown in Listing II-8.

First, the vector operands must be loaded via LOAD-intrinsics, that—in this example—loads four consecutive single precision values at the given position in memory into the vector variables. Vice versa, the result must be stored back.

Usually one would prefer AVX 1 as it results in the highest performance boost since it handles twice as many components as SSE 4.1. On the contrary, once the code is based completely on AVX 1, it must be rewritten when a better extension, working with larger vectors, is available.

In order to let the user decide between both, the floating point precision and the set of vector extensions, the code for this thesis was not written directly in intrinsics, but with macros that are mapped to intrinsics depending on the corresponding compiler flags. By doing so, it is possible to provide code using either the SSE 4.1 or the AVX 1 instruction set. Thus the code can easily be adapted to future extensions, that use for instance vectors with 512 bits or more, without writing and testing the frame for intrinsics again.

However, future extensions are supposed to behave in the same way with respect to masking and blendig (see Section II-F). As a counter example, this is not true for the vector instructions on the Intel MIC architecture. This issue is further discussed in Section IV.
Instead of the C++ built-in types `float` and `double`, a typedef named `real` is used which represents a precision, chosen by the user, again via a `#define` macro. Additionally, a corresponding unsigned integer type named `integer` is defined. In the case of single precision, `integer` becomes an `uint32_t`, which is itself a typedef and corresponds to `unsigned int` in most architectures. If double precision is used, `integer` becomes an alias for `uint64_t`, which corresponds to `unsigned long`.

For the vectorization of the augmented Riemann solver, two files have been created. These files are explained in the following.

The file `SIMD_TYPES.hpp` controls the typedefs as well as the internal macros for the vectorization. When compiled with `-msse4`, the macro `SSE4` is defined and additionally all macros for the previous versions of SSE (both, GCC and Intel compiler). These macros are also defined with the compile flag `-mavx`, but in this case additionally the macro `AVX` is present. The header `SIMD_TYPES.hpp` reacts to these macros and sets the corresponding typedefs for floating point vectors and scalars as well as for their integer pendants.

The main functionality of that file is outlined in Listing II-9. The first part arranges the precision. By default it is set to single precision and can be switched to double via defining the macro `FLOAT64`. The next part takes care of the vector extension. Here it is important, to first check for AVX, since the SSE 4.1-indicator-macro is also defined, as explained above. In addition to the typedefs, the corresponding headers are included, too. However, this is not shown in Listing II-9.

The last part of the header is crucial. It handles the case, that no vectorization should be done. The vector length is set to one, reducing computation to scalars. The vectorized loop in Listing II-2 is not needed in this case, since it is up to the remainder handler to care about the scalar case. So, the vector code can be blended out, enabling compilation for architectures which lack in the corresponding vector extensions. Otherwise the compiler would refuse to compile code with instructions present, that do not exist on the target machine.

The final version of the artificial example in Listing II-1 then becomes Listing II-10.

```c
const int L = VECTOR_LENGTH;
const int end_n_vector = (n / L) * L;
int i = 0;
#if !defined NOVEC
for (; i < end_n_vector; i += L)
  // vector code goes here
#endif
for (; i < n; ++i)
a[i] = b[i] + c[i];
```

Listing II-10. Final version of the vectorized loop

### E. Macros Instead of Intrinsics

As mentioned in Section II-D, the vector instructions are mapped to macros. Thus, it is possible to provide a common, macro-based implementation and to choose the vector extensions and the precisions at compile time. More over debugging becomes easier. If the code works for one extension, no changes for new instruction sets are required, as long as the new instructions provide compatible functionality.

The macros are defined in the header file `SIMD_DEFINITIONS.hpp`. The general idea of that file is demonstrated in Listing II-11.

The instructions which are not available with AVX 1, such as comparing integers for equality, are defined as functions, that provide the same functionality as the missing intrinsics should do. This introduces a slight performance impact which can be solved with AVX 2.

From now on, vector instructions are denoted as macros, i.e. functions with capitalized names, colored in cyan.
F. Handling Divergent Branches

The previous sections Section II-D and Section II-E introduce the initial setup. The loop calling compute-NetUpdates() is blocked by the length of the vectors, to compute multiple Riemann problems at once. The intrinsics are hidden behind macros.

The following sections focus on the solver in the file AugRie.hpp, which must now handle vectors of size L. Here the challenge arises, that, depending on the wet-dry-state of the cells, different code paths have to be executed. This issue occurs when determining the Riemann structures of the waves, as well as when computing the middle state.

In scalar code, multiple Riemann problems are solved sequentially. Within a single problem, the control flow takes different branches.

In vectorized code, multiple Riemann problems are solved in parallel and each problem can take its own branch. Thus, the control flow might take different branches at the same time.

The example in Listing II-12 contains an assignment to an array c, depending on an if-condition. The value of the array element is only modified, if the if-condition is true.

```cpp
for (int i = 0; i < n; ++i)
  if (a[i] <= b[i])
    c[i] = d[i];
```

Listing II-12. Diverging branch within scalar code

In a vectorized version some elements of the same vector may fulfill the condition and take the branch, while others do not. There are two different ways to handle this issue, namely MASKING and BLENDING, which are described in detail in the following. Both require the execution of the code within the if-clause with the whole vector, even if only one element takes the branch.

In the following, the loop body in Listing II-12 is vectorized with both approaches to illustrate the differences.

1) Masking

The results of compare-instructions in SSE and AVX are masks—vectors, where the single components have all bits set to 1, if the condition is fulfilled. Otherwise all bits of the corresponding components become 0.

Selecting the corresponding vector components is achieved via computing the logical AND bit by bit with the mask. Thus the result of the AND-operation is either the selected component or zero.

The solution to resolve a simple branch as shown in Listing II-12 using the masking approach is to take the sum of two vectors. The elements from the first vector are chosen via the mask and the elements from the second by the mask’s bitwise complement.

This is demonstrated in Listing II-13.

```cpp
const real_vector mask = CMP_LT(a, b);
c = ADD(AND(d, mask), AND(c, NOT(mask));
```

Listing II-13. Diverging branch handled with masking

This approach may become rather complicated and error prone when vectorizing if-else-blocks with more than one condition or even switch-statements with several cases.

2) Blending

Another way to handle branches is to use blend instructions. They work on the masks's sign bits to select components of the two arguments. This method is more transparent as it reflects an if-else-block within one instruction, as shown in Listing II-14.

```cpp
const real_vector mask = CMP_LT(a, b);
c = BLEND(c, d, mask);
```

Listing II-14. Diverging branch handled with blending

The blending approach performs slightly faster than the masking approach. On Sandy Bridge for example, blending takes 1 cycle in the best and 2 cycles in the worst case [13, page C-8]. The masking approach needs two bitwise ANDs each taking 1 cycle [13, p. C-16] and one addition, taking 1 cycle in the best and 3 in the worst case [13, p. C-16]. So, in some situations, blending is up to 4 cycles faster than masking.

In the code for this thesis, blending was preferred due to better readability. In addition it is sufficient to only provide proper sign bits instead of full masks. This leads to advantages as described below.

G. Working With Sign Bits

The code within an if-clause often contains more than just an assignment. Take the function compute-MiddleStates() as an example.

Here, first the Riemann structure of the underlying waves has to be determined. Then a corresponding nonlinear equation has to be solved. In the case, that the
two waves are both rarefaction waves, the solution can be given directly. If one of the cells is dry, the solution consists of a single rarefaction wave.

But, whenever a shock wave occurs, a non-linear system must be solved via the Newton-Raphson method. Apparently, the Newton branches must only be taken, if it is required by at least one component of the vector.

In order to detect whether a branch must be taken or not, one must detect whether a component within the vector fullfills the corresponding condition. For the result of a comparison (a mask as described above) it is sufficient to check just one bit.

This is managed by the instruction MOVEMASK. This instruction maps a vector to an integer, where the \( k^{th} \) bit of the integer is set to the sign bit of the \( k^{th} \) vector component.

If MOVEMASK returns zero, all conditions are false, if it returns \( (2^L - 1) \) (L bits set to 1) all conditions are true.

Then the question arises, how to derive the masks in a fast way. In former scalar based code, a member variable

<table>
<thead>
<tr>
<th>enum-value</th>
<th>integer-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShockShock</td>
<td>1000 0000 ...</td>
</tr>
<tr>
<td>RarefactionRarefaction</td>
<td>0100 0000 ...</td>
</tr>
<tr>
<td>ShockRarefaction</td>
<td>0010 0000 ...</td>
</tr>
<tr>
<td>DrySingleRarefaction</td>
<td>0001 0000 ...</td>
</tr>
<tr>
<td>SingleRarefactionDry</td>
<td>0000 1000 ...</td>
</tr>
</tbody>
</table>

Table II.2
Mapping of enum-values to integers

named riemannStructure was assigned a value of an enumeration—respective an integer—indicating shock or rarefaction.

This approach has been adapted to the vector code by using integers, where—read from left to right—only a single bit is set (see Table II.2). Note that—per construction—the member vector holding the Riemann structures then automatically acts as a blend mask for the ShockShock-case. It is sufficient to apply a MOVEMASK on the member vector in order to check whether this is the case.

This approach can be applied to all other possible cases (see Table II.2) by shifting the single vector components bit-wise to the left. Then we obtain, bit by bit, blend masks for the remaining cases. This makes it easy to handle the different code paths in compute-MiddleStates(), while keeping the code clean and understandable. The technique is outlined in Listing II-15.

```c
void f {const real_vector blend_mask)
{
    real_vector local_var;
    ...
    // do computations on local_var
    ...
    // blend the local results into the member
    // according to the blend mask given
    this->var = BLEND{
        this->var, // keep the old values
        local_var, blend_mask
    };
}
```

Listing II-15. Blending at the end of member functions to avoid unintentional changes

H. Additional Changes To The Solver

With the techniques described so far, the scalar solver can be completely rewritten with vector instructions. At first glance it is just a replacement of the arithmetic computations by vector instructions. Every member variable gets a vector pendant and every member function a second version, which does exactly the same, only based on vectors.

The more complex changes are imposed by functions changing member variables—such as the middle-state height and speed or the Riemann structure—at different points in the control flow, because other functions need these values to be set properly. Therefore every function changing any member variable must be extended with a blend mask as an argument. That blend mask is to avoid unintentional changes to components of the member vectors.

For example, if only one component needs the computation of the middle-state, the computations are done with all components on local variables and afterwards the members are updated according to the blend mask. The general procedure is shown in Listing II-16.
The last change to the solver concerns the arguments of the computeNetUpdates() -function. Originally the function took scalar values passed by value. The arguments have to be changed to pass pointers to the arrays holding the water height, the momentum and the bathymetry. The pointers are necessary to get the values from memory into vectors and vice versa by LOAD- and STORE-instructions.

I. Performance

The performance of the amendments has been measured on a dual socket Intel Xeon E5-2670 with 2.6 GHz and with turbo mode disabled. For this measurements, no parallelization but a single core has been used to allow a better comparison between the scalar and the vectorized code.

The theoretical peak performance of one core is then given by

\[ 2.6 \cdot 10^9 \frac{\text{Cycles}}{\text{s}} \cdot \frac{8 \text{ Operations}}{\text{Cycle}} = 20.8 \text{ GFLOPs}. \]  

(II.1)

Here the term \( \text{Operations} \) stands for one of the following FLOATING-POINT OPERATIONS:

- Addition or Subtraction
- Multiplication or Division
- Square-Root
- Minimum and Maximum

Since the theoretical peak in Equation (II.1) assumes eight operations per clock-cycle, the value is only correct for single precision computations, that are done with AVX instructions. With SSE4 the value decreases to the half, i.e. to 10.4 GFLOPs. These values will be referred as the PRACTICAL PEAK.

Peak performance is achieved, if one of the instructions as listed above is computed in every cycle. This is the case in dense matrix-matrix multiplications.

However, the augmented Riemann solver has lots of integer, logical and blend instructions. Neither these instructions nor the load and store instructions are counted by the performance measurements.

Measurements show, that these non-floating-point-operations take nearly the half of all instructions within the solver. Thus, a result in the range of 50% to 55% of the practical performance is the maximum that can be achieved.

The propagation of the Tōhoku tsunami, as seen in Figure I.5, is taken as a reference. The simulation has been computed on a \((1000 \times 1000)\)-grid for 8 hours. Table II.3 and Figure II.1 show the results.

The scalar code reaches already 30% of the achievable performance. This is also obtained by the SSE version and leads to a very good speedup of 4.2, which is slightly greater than theoretically possible. This effect is most likely affiliated to cache effects.

However, the results for the AVX vectorization look even worse. Although—with respect to the SSE results—a factor of 8 is expected, a factor of 5 is obtained.

This is astonishing, since the same code has been executed in the SSE version, only with other intrinsics behind the macros. AVX1 lacks of two instructions: the check for two integer vectors to be equal and the shift-left operation. These instructions are implemented in software rather than in hardware. However, that little performance impact of these two instructions can not explain the performance loss of the whole solver.

Another explanation is, that the Sandy Bridge architecture is able to load 16 bytes of data from the L1 cache as well as to store 16 bytes of data back to the L1 cache within a single cycle. If nothing is to be stored, the chip is actually able to load 32 bytes [13, p.3-41].

Thus with SSE, the CPU is able to load up to two operands from the L1 cache within one cycle. With SSE, the chip is able to store the content of one register, back to the L1 cache, within one cycle, too.

However, with AVX the CPU needs at least two cycles, to load two operands from the cache into registers and, again, it needs at least two cycles to store one register back to the cache.

Usually, the number of registers is always too small and, due to the fact that the augmented solver needs lots of temporary variables, it is realistic to identify the L1 cache bandwidth as the bottleneck. Thus, future experiments on Haswell CPUs are necessary.

Haswell CPUs extend the bandwidth from \([32/16]\) to \([64/32]\). This results in the same cache characteristics for AVX1 as described above for SSE.
Further on, Haswell supports AVX2 and eliminates the performance impact due to the software implementation of missing instructions.

Transferring the results of the SSE vectorization on Sandy Bridge to AVX on Haswell, a speedup in the magnitude of 8 is likely to be achieved.

J. Known Issues

Some problems came up during the development of the vectorized code. These problems are briefly described in the following.

1) Deviations From The Reference

When computing the wave decomposition, it is necessary to solve a linear system to obtain the coefficients \( a_{p+1/2} \) (recall Equation (I.41)). The system matrix consists of the waves \( w_{p+1/2} \). However, there exist situations, where this matrix becomes very ill-conditioned.

The code for solving the linear system contains a line of the form

```c
const real result = a + b + c;
```

which is evaluated by the compiler as

```c
const real result = (a + b) + c;
```

(note the parentheses).

The vectorized code, however, was first written as

```c
const real_vector result = ADD(a, ADD(b, c));
```

which corresponds to an evaluation like

```c
const real result = a + (b + c);
```

Tests have shown, that there are Riemann problems within the Töboku simulation, where the results between these two choices of placing brackets differ by up to 20%.

Reformulating the vectorized line to correspond with the scalar line makes the deviations decrease to the range of \( 10^{-5} \). This last discrepancy can probably be affiliated to the fact, that in the vectorized code implicit parantheses have been introduced at other places, too.

The behaviour can be explained with the limitations of single precision. Single precision numbers have an accuracy of about \( 10^{-7} \). The water height is in the range of \( O(1000 \text{ m}) \), such that the last significant digit represents water heights in the range of millimeters. Typically, one digit gets lost due to rounding issues. So, the highest possible accuracy is in the centimeter-range.

However, discrepancies of centimeters can be very significant for tsunamis, especially near shorelines.

This issue needs further investigation and can probably be solved by using double precision.

2) Intel Compiler Issues

All performance measurements in this thesis were compiled with the GNU C++ compiler. The reason is, that the Intel compiler (version 13) aborts compilation of the vectorized code due to an “Internal Error”. According to Intel’s documentation, this is rather an issue with the compiler, than with the code.
III. CUDA

"CUDA™" is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).” [14]

The acronym stands for Compute Unified Device Architecture. The technology is implemented on GPUs from NVIDIA.

CUDA allows the programmer to use directly the GPU’s instruction set and memory in Fortran, C or C++ code. Wrappers for other languages such as Java or Python exist, too.

The programmer provides so-called kernels. These are executed in parallel on the GPU. The details are described in Section III-B.

CUDA introduces some extensions to the C language, which can be found in [15]. It is necessary to compile the CUDA code with NVCC, a compiler provided by NVIDIA. This compiler translates the kernel to binary code which runs on the GPU. The binary GPU code then gets linked with the CPU code when the executable is created.

The programming with CUDA is briefly described in the following. A general tutorial for how to write CUDA code and how to get peak performance can be found in [15].

A. Basics

GPUs consist of many computation cores, so-called streaming multiprocessors. These multiprocessors are assigned blocks of threads. On a multiprocessor, groups of 32 threads are executed in parallel. These groups are called warps. The aim is to execute many threads slowly in parallel rather than few threads fast.

GPU threads are different from CPU threads. They are more specialized with respect to the underlying hardware and are faster to create and to schedule.

A major difference between GPU and CPU is a smaller cache of the GPU. Further on, the GPU works with its own memory. The bandwidth between the GPU and its memory is usually much higher than the bandwidth between a CPU and the main memory. In order to distinguish, the GPU is referred to as the device, and the CPU as the host.

The parallelization is similar to the SIMD technique described in Section II. All threads within a single warp execute the same instruction. This is similar to the usage of vectors. The challenges with divergent branches arise again (compare Section II-F). This issue is described in Section III-C.

In order to work on the GPU, the data must reside in the GPU’s memory. There are two functions cudaMalloc and cudaFree to allocate and free the GPU’s memory. They behave in the same way as their C pendants malloc and free. In the examples of this sections, data is considered to be allocated in the GPU’s memory.

B. Kernels

Kernels are plain C functions that are called via the syntax kernel<<<grid_dim, block_dim>>>(...). The <<<...>>> part is one of the extensions to the C language and interpreted by the NVIDIA compiler. The variables grid_dim and block_dim indicate how the kernel is executed in parallel on the device.

These variables are either of type int or dim3. The latter type represents a three-dimensional variable. They enable the developer to introduce a logical structure of thread blocks and threads.

Threads can be grouped in blocks with a one-, two- or three-dimensional structure. These (equal shaped) blocks of threads can again be grouped in a one-, two- or three-dimensional grid. Listing III-1 shows a simple example for a kernel call.

Each thread is given a private variable with the name threadIdx of type dim3. This variable identifies the thread within its block. In order to determine the block within the grid, each thread has a corresponding

```
int main ()
{
  dim3 grid_dim(4, 2);
  dim3 block_dim(24, 16);
  kernel<<<grid_dim, block_dim>>>();
}
```

Listing III-1. Calling a kernel with 8 blocks, organized as a $4 \times 2$ grid. Each block has a shape of $24 \times 16$ and thus 384 threads. In total, the kernel is executed with 3072 threads in parallel.

Figure III.1. Grid consisting of $(2 \times 2)$ blocks each with $(3 \times 2)$ threads. The thread ids are relative to the block.
variable `blockIdx`, too. This makes it easier for the programmer to map threads to the computational domain, for instance.

Threads within a common block are expected to be executed on the same multiprocessor. On current GPUs the maximum number of threads within a single block is restricted to 1024 [15]. If more threads are needed, more blocks must be created.

Usually the problem determines the number (and the shape) of blocks. However, the number of blocks is allowed to exceed the number of available multiprocessors. Figure III.1 shows an example of a grid and its thread blocks. The figure is adapted from NVIDIA’s documentation [15].

Further on, the kernel function must be declared with the keyword `__global__`. This keyword indicates that the function is called on the host but executed on the device.

Listing III-2 shows the code of Listing II-1 with CUDA. The example code is taken from the NVIDIA documentation in [15]. It is slightly adapted, such that the variable names match to the ones in Listing II-1. Additionally, some comments and variables for the grid and the blocks have been added to make it more transparent.

```c
// Kernel definition
__global__
void VecAdd (float* a, float* b, float* c)
{
    int i = threadIdx.x;
    a[i] = b[i] + c[i];
}

int main ()
{
    ...
    // create a grid with 1 thread block
dim3 grid_dim(1, 0, 0);

    // create a thread block with n threads
dim3 block_dim(n, 0, 0);

    // Kernel invocation with n threads
VecAdd<<<grid_dim, block_dim>>>(a, b, c);
    ...
}
```

Listing III-2. Kernel for vector addition from Listing II-1; the example is taken from [15]

Note that there is no loop anymore. The kernel is executed by one block, consisting of `n` threads (this works only for `n \leq 1024`) and each thread computes exactly one summation. The thread determines the element to compute by the thread id. This id is obtained by the (three-dimensional) variable `threadIdx`, with components `x`, `y` and `z`.

This results in a three-dimensional coordinate relative to the thread block. In order to get the absolute coordinate within the grid, the variable `blockIdx` can be used. The number of threads per block in each dimension is stored in the `dim3` variable `blockDim`. This variable enables the computation of a unique (three-dimensional) thread-id within the grid, as demonstrated in Listing III-3.

```c
// determine the block
int block_i = blockIdx.x * blockDim.x;
int block_j = blockIdx.y * blockDim.y;
int block_k = blockIdx.z * blockDim.z;

// determine absolute position within the grid
// by the block determined above
// and relative position inside the block
int i = block_i + threadIdx.x;
int j = block_j + threadIdx.y;
int k = block_k + threadIdx.z;

dim3 unique_id(i, j, k);
```

Listing III-3. Computing the absolute position of a thread within a grid

C. Thread Divergence

As already mentioned, threads are executed in warps, i.e. groups of 32 threads on a multiprocessor and all threads within a warp execute the same instruction at a time.

If the control flow diverges into several paths, the warp becomes serialized [15]. The warp then executes all these paths in a sequential-like way and all threads, that do not take a specific branch, are disabled by hardware while the warp executes this branch.

Ideally all 32 threads take the same branch, resulting in fully parallelization, i.e. a speedup of 32.

However, consider code, where 31 threads take the first branch and only one thread, say thread X, takes a second branch. Then the first branch is parallelized by 31 threads with thread X disabled. After the 31 threads have finished their branch, they get disabled when thread X is enabled and executed in the second branch.

In the worst case all threads take different paths. Then parallelism is lost completely and all threads in the warp are executed sequentially.

This issue is called **thread divergence**. It can only happen to threads within a common warp.

With CUDA, the task for the programmer is different from the branch handling with vectorization as explained in Section II-F. Divergence handling is done by the hardware.

Assuming `n \leq 1024`, a CUDA version of the branch in Listing II-12 can be formulated as in Listing III-4.

D. Implementation In The SWE Teaching Code

The SWE teaching code contains already a CUDA parallelization for the `f-wave` solver, but not for the augmented Riemann solver. This section describes the CUDA parallelization of the augmented Riemann solver.
When using CUDA, all computations are done on the GPU. The solver and therefore the function computeNetUpdates(), is implemented as a kernel.

The changes to the code, to make the augmented solver work with CUDA, are rather moderate in contrast to the changes introduced by the SIMD vectorization.

The first step is to separate the augmented solver’s code from the abstract class structure and transform it into standard C functions. As C is not object-oriented, member variables have to be eliminated.

Functions using these member variables get their argument list extended as follows. Functions reading member variables get the values of these variables passed as an argument. Functions writing to member variables get additional output arguments passed by reference. Here, the variable is created by the caller and set by the callee.

To avoid function calls with a lot of variables and frequently copying arguments, all function calls are replaced with the actually function body as far as possible. This concerns the functions for computing the wet-dry-state, the wave decomposition, the Riemann structure and the solution of the linear system.

In the former C++ code, these functions are called only once within the computation of a single net-update. The reason for their existence is encapsulation. Thus, this denotes a step backwards with respect to software engineering. However the manually inlining eliminates the need for some member variables such as the wet-dry-state and the Riemann structure.

After transforming the solver to C code, the kernel function has to be integrated into the simulation code. This is done via the compile flag -DCUDA_AUGRIE. This flag defines a macro that switches between the f-wave and the augmented Riemann solver at compile time. Thus, the changes in the existing simulation code to switch between the two solvers, are just a few lines, as demonstrated in Listing III-5.

```c
// this is the kernel, which is executed
// in parallel
void computeNetUpdates_kernel(...) {
  ...
  // switch the solver depending
  // on the macro CUDA_AUGRIE
  # ifdef CUDA_AUGRIE
    AugRie(...);
  # else
    fWave(...);
  # endif
  ...
}
```

Listing III-5. Switching the solvers within the SWE teaching code

It looks like that the CUDA parallelization is much easier than the SIMD vectorization. However, this is because the general CUDA interface exists already in the SWE teaching code. This keeps the changes as simple as described above. In general, achieving peak performance in CUDA needs more than just providing a kernel that is executed in parallel.

One task is to identify components within an application that can be done in parallel. Another task is, for instance, a synchronization strategy if needed. For obvious reasons, it is important to avoid thread divergence, too. A general optimization guide can be found in [15].

Most of these tasks are already handled in the SWE teaching code.

E. Performance

The performance measurements of the CUDA parallelization are done on a NVIDIA Tesla M2090 GPU. The GPU provides 512 CUDA cores (multiprocessors), 64 GB memory, and, according to NVIDIA, a theoretical single precision peak performance of 1.331 TFLOPs [16].

The CUDA performance is compared to that of a Sandy Bridge node consisting of 2 dual socket Intel Xeon E5-2670 with 8 cores each. In the CPU code, the computation of the net-updates has been parallelized with OpenMP. Moreover, Hyperthreading is used to execute 32 threads on the Sandy Bridge node. The theoretically peak performance of the CPU configuration is 665.6 GFLOPs. Thus, the GPU is almost two times faster than the Sandy Bridge node.

However, comparing CPUs and GPUs for floating point operations leads to wrong conclusions. Among other issues, this is because NVIDIA GPUs provide a fused multiply-add operation (FMA), which is able to do a multiplication and an addition in one cycle. This leads to two FLOPs within one instruction and doubles the theoretical peak performance.
In contrast, the peak performance of 1.331 TFLOPs is achieved only if the code can take advantages of the fused multiply-add. This is the case when multiplying two dense matrices but not with the augmented Riemann solver. Only a few lines of code can be accelerated by the FMA. Thus, the theoretical peak is rather in the range of 665.5 GFLOPs.

The FLOP-rate is not used in order to allow reasonable comparisons. Measurements such as the number of net- or element-updates per second enable a more suitable comparison. Net-updates represent the number of Riemann problems that are solved.

Note that this number does not reflect directly the number of element-updates. Every element, i.e. every cell of the finite volume discretization, uses the results of four Riemann problems—above, below, to the left and to the right. However, every Riemann problem contributes to two adjacent cells. So the number of element updates is nearly the half of the net-updates.

Note that it is not exactly the half, since \( n \) cells need \((n + 1)\) Riemann problems to be solved.

In order to avoid such calculations, the results are measured in net-updates per second or \( \text{NUps} \). Again the Tōhoku tsunami was used as reference scenario this time on a grid with \((4800 \times 4800)\) cells. Again, the propagation was simulated for eight hours.

The CUDA implementation uses thread blocks of the sizes \((16 \times 16)\) and \((24 \times 24)\) and thus only 256 or 576, instead of 1024 possible threads. The reason for that are technical issues. With more threads the application abnormally ends due to a “lack of resources”. This issue needs further investigation and goes beyond the scope of this thesis.

Under ideal circumstances, the \((16 \times 16)\) thread block can only achieve half of the peak performance. The \((24 \times 24)\) block makes more usage of parallelization, but some threads are always idle. The restriction, that the thread blocks must be quadratic are due to implementation details in the SWÉ teaching code.

\[
\begin{array}{|c|c|c|c|c|}
\hline
& CUDA & CUDA & AVX 1 & SSE 4.1 & \text{scalar} \\
\hline
\text{NUps} & 447M & 436M & 368M & 312M & 105M \\
\hline
\end{array}
\]

Table III.1

Million net-updates per second (NUps) for the CUDA implementation compared with one Sandy Bridge node with 32 OpenMP threads.

Table III.1 and Figure III.2 show the results. First of all it can be observed that the speedup of the vectorization decreases on a Sandy Bridge node with 16 cores. This can be affiliated to cache effects. If the simulation is run on a single core, this single core has 20 MB L3 cache for its own. However, in the case of 32 threads, the L3 cache is shared between all these threads.

The measurements provide a poor performance of the current CUDA implementation compared to the vectorized Sandy Bridge implementation. A factor of 2 is expected, assuming a peak of 1.3 TFLOPs. However, a factor in the range of 1.2 is obtained. This drawback can not be explained by missing FMA acceleration only. Recall that the AVX implementation is expected to be even faster on a Haswell CPU.

Moreover, using 576 threads per block is slightly slower than using 256.

Thread divergence might be a reason for the poor performance of the CUDA code compared with the vectorized CPU code. However, comparisons of the scalar versions of the augmented Riemann solver with the f-wave solver (with respect to the net-updates per second) show, that the f-wave solver is double as fast as the augmented Riemann solver. Comparisons of the CUDA versions of both solvers show the same behaviour. So, the poor CUDA performance might be caused by problems with the CUDA parallelization in general.

Nevertheless, an appropriate answer for this has not been found yet and is subject to further investigation.
IV. Outlook – Vectorization On The Intel MIC Architecture

MIC stands for Many Integrated Core. MIC processors consist of many Pentium-like cores with small clock rates in the range of 1 GHz.

In contrast to latest IA-32 or AMD64 CPUs, MIC processors neither support SSE nor AVX. Instead, they have their own set of vector registers and instructions. The registers contain 64 byte (512 bit), thus covering 16 single precision or 8 double precision numbers.

Handling of diverging branches is fundamentally different. The results of comparisons are no longer vectors. Recall that in SSE and AVX, if the condition between the \( k^{th} \) components are fulfilled, the \( k^{th} \) component of the resulting vector has all bits set to 1.

Compare instructions on MIC, however, return either a short integer (16 bit for single precision) or a char (8 bit for double precision). If the condition between the \( k^{th} \) components is fulfilled, the \( k^{th} \) bit is set to 1.

Every instruction has a normal version, working on all components and a masked version. The masked version is given a mask and a source vector as parameter. Then, the operation is only computed for these components, where the corresponding mask bit is set. All other components in the result vector are copied from the source vector.

Listing IV-1 shows an example, where only the components with even indices are summed up. All components with odd indices are copied from the source vector. The type \texttt{__mmask8} is a typedef for \texttt{unsigned char}. The type \texttt{__m512d} corresponds to a vector with 512 bits, containing 8 double precision numbers.

```c
__m512d masked_sum(__m512d a, __m512d b)
{
    // create a mask with 0101 0101
    __mmask8 mask = 0x55;

    // create source with all components
    // set to zero
    __m512d source = _mm512_setzero_pd();

    return _mm512_mask_add_pd(
       source, 
       mask, 
       a, 
       b
    );
}
```

Listing IV-1. Masked addition with MIC-intrinsics. Only components with even indices are computed, the others are copied from the source vector. The variables are assumed to be double precision.

This eases handling of divergent branches a lot. The blend-shift-blend-approach described in Section II-G is not necessary anymore. Instead, member variables for all different wet-dry-states as well as the different Riemann structures have to be introduced. These member variables are supposed to act as masks, as seen in Listing IV-1. The test, whether at least one condition is fulfilled degenerates to testing the mask to be equal zero.

Due to this differences in the branch handling, it is not possible to extend the macro based implementation described in Section II-E to work with MIC-intrinsics by just mapping intrinsics to macros. Instead, the solver has to be rewritten completely with MIC-intrinsics.

However, the rewrite can be done with macros again. That enables adaption of the code to future extensions of the MIC architecture. Thus it is possible, to provide all future-relevant vector extensions—SSE, AVX and vector instructions on MIC—with only two implementations.
V. Conclusion

A. Vectorization On IA-32 and AMD 64

The vectorization described in this thesis accelerated the augmented Riemann solvers by orders of magnitude compared to the scalar based version.

With SSE 4.1 a speedup of 4.2 is achieved, while the theoretical maximum is a factor of 4. The discrepancy is probably caused by cache effects. The implementation achieves up to 31% of the theoretical peak performance, with respect to the number of floating point operations. Nearly half of the solver’s code consists of integer operations and branches. So a portion of 31% of the floating point peak performance is pretty good.

The AVX 1 implementation, however, achieves only 19% of the theoretical peak. This results in a speedup of 5.2, while a factor of 8 is feasible. Most likely this effect can be affiliated to the limited L1 cache bandwidth. Further experiments with Haswell CPUs are necessary to find the concrete bottleneck. If the assumption concerning the limited cache bandwidth holds, a speedup of 8 is very likely on the Haswell architecture.

Further on, the mapping of the underlying machine instructions to macros enables the usage of wider vector registers in the future, while minimizing the need for testing.

B. CUDA

The performance of the CUDA implementation is not as good as expected. An NVIDIA Tesla M2090 performs only slightly faster than a Sandy Bridge node with two dual-socket Intel Xeon E5-2670.

Under ideal circumstances, the GPU is two times faster than the CPU node. However, the augmented Riemann solver achieves only a speedup of 1.2.

Further investigation is required to find out whether the factor of 1.2 can be beaten.

C. Vectorization On Intel MIC

The vector units on Intel MIC behave different from the vector units on IA-32 or AMD 64 CPUs. So, it is not possible to port the vectorized version of the solver to MIC without a rewrite.

This rewrite should be done with the same macro approach as the vectorization in this thesis. Then it is possible to provide a version for each relevant vector extension with two concrete implementations. One for the IA-32 or AMD 64 architecture and one for Intel MIC. This minimizes maintaining, testing and debugging of the code, while being well-prepared for vector extensions in the future.

D. Getting The Code

The code written for this thesis is Open Source and part of the SWE teaching code. All files are freely available at [3].
References


