Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL

Markus Büttner [1] Christoph Alt [2,3] Tobias Kenter [2]
Harald Köstler [3] Christian Plessl [2] Vadym Aizinger [1]

[1]Chair of Scientific Computing, University of Bayreuth

[2]Paderborn Center for Parallel Computing

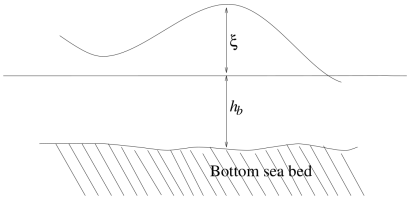
[3]Department of Computer Science, FAU Erlangen

June 3rd, 2024
PASC 2024

Shallow Water Equations

$$\partial_t \xi + \nabla \cdot \mathsf{q} = 0$$

$$\partial_t \mathsf{q} + \nabla \cdot (\mathsf{q}\mathsf{q}^T / H) + \tau_{bf}\mathsf{q} + \begin{pmatrix} 0 & -f_c \\ f_c & 0 \end{pmatrix} \mathsf{q} + gH\nabla\xi = \mathsf{F}$$



- Variables: Water height ξ, horizontal velocity $\mathsf{q} = (U, V)^T$
- Total water depth $H = h_b + \xi$
- Depth-integrated vertical velocity
- Application: Coastal ocean simulations (tides, tsunamis, storm surges)

Discontinuous Galerkin method

UNIVERSITÄT BAYREUTH

- Partition of domain into triangles Ω_e
- Hierarchical basis on each triangle (orthogonal polynomials on triangles)
- Define discrete solution $c_\Delta = (\xi_\Delta, U_\Delta, V_\Delta)$,

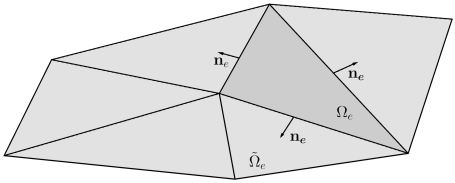
$$\xi_\Delta(t,x) = \sum_{j=1}^{k} c_{1j}(t)\varphi_j(x) \qquad U_\Delta(t,x) = \sum_{j=1}^{k} c_{2j}(t)\varphi_j(x) \qquad V_\Delta(t,x) = \sum_{j=1}^{k} c_{3j}(t)\varphi_j(x)$$

- k dependent on order (1 for piecewise constant, 3 for linear, 6 for quadratic basis)
- Degrees of freedom (DOF) c_{ij} per element ($i = 1,2,3$, $j = 1,\ldots,k$)

Discontinuous Galerkin method

UNIVERSITÄT BAYREUTH

- Multiply with test function φ_Δ, integrate over Ω_e, integration by parts once
- Solution generally has jumps between elements
- Discretization locally mass conservative
- Original UTBEST implementation in Fortran/C [1]



$$\frac{d}{dt}\int_{\Omega_e} \xi_\Delta \varphi_\Delta dx - \int_{\Omega_e} \mathsf{q}_\Delta \cdot \nabla \varphi_\Delta dx + \int_{\partial\Omega_e} \widehat{F}(\xi_\Delta, \mathsf{q}_\Delta, \widetilde{\xi}_\Delta, \widetilde{\mathsf{q}}_\Delta, \mathsf{n}_e)\varphi_\Delta dx = 0$$

with numerical flux $\widehat{F}$ (Lax-Friedrichs); $\widetilde{\xi}_\Delta, \widetilde{\mathsf{q}}_\Delta$ values from neighbour element, n_e outward pointing normal vector

[1] V. Aizinger and C. Dawson, "A discontinuous Galerkin method for two-dimensional flow and transport in shallow water", Advances in Water Resources, 2002

Element-wise implementation of DG

UNIVERSITÄT BAYREUTH

- Projection approach originally implemented for FPGAs [2]
- Compute L^2 projection of solution onto edges
- Use projected solution during edge integration

→ No loop-carried dependencies in element loop, trivial to parallelize

while $t < t_1$ **do**
  Loop over Runge–Kutta stages:
  for all stages of the Runge–Kutta method **do**
    Element loop:
    for all element indices $e \in \{1, \ldots, E\}$ **do**
      calculate element integrals
      calculate edge integrals, update only own element
      calculate $\boldsymbol{c}_\Delta$ for the next Runge-Kutta stage
      perform minimum depth control on $\boldsymbol{c}_\Delta$
      for all edges belonging to Ω_e **do**
        $\boldsymbol{p}_\Delta \leftarrow$ projection of $\boldsymbol{c}_\Delta$ to edge
        store $\boldsymbol{p}_\Delta$ in global array
      end for
    end for
  end for
  $t \leftarrow t + \Delta t$
end while

[2] Kenter et al. "Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA", PASC 2021

Goal: Single-source Performance Portability

UNIVERSITÄT BAYREUTH

- Target platform: CPUs, GPUs and FPGAs
- Existing codes target only CPUs or GPUs or use code generation

Can we run on all three platforms using the same source code?

- Reuse as much code as possible
- Some hardware-specific abstractions might be necessary
- SYCL: open standard for single-source programming for heterogeneous computing
- oneAPI supports CPUs, GPUs and Intel FPGAs out of the box
- AdaptiveCpp supports CPUs and GPUs

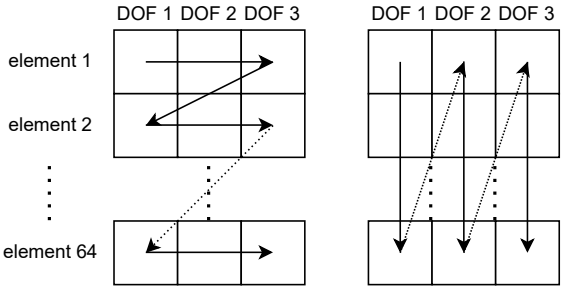
Parallelization

UNIVERSITÄT BAYREUTH

- `parallel_for` obvious choice for parallelization on CPUs and GPUs: Each work item corresponds to one element
- Double-buffering for projected solutions
- `single_task` for pipeline design on FPGAs:
  - All inner loops (e.g. over quadrature points, degrees of freedom) must be unrolled
  - Element loop pipelined: each clock cycle one element starts processing
  - Multiple time-steps can be computed per kernel launch
  - Dependency analysis of FPGA compiler ensures correctness

Memory layout: degrees of freedom

- Degrees of freedom for ξ_Δ, U_Δ and V_Δ stored in contiguous array
- Memory layout depends on target architecture
- Array of Structs-like (left, original CPU implementation, FPGAs)
- Struct of Arrays-like (right, CPUs and GPUs with SYCL)
- Switched by compile-time flag



Custom FPGA-Caches

UNIVERSITÄT BAYREUTH

- Major limitation: off-chip memory bandwidth (~69 GB/s measured in stream benchmark)
- Previously shown (Kenter et al., 2021): High performance possible for small meshes, but mesh size fixed during compilation

→ Design with high performance for small meshes and no limitation on mesh size?

Custom FPGA-Caches

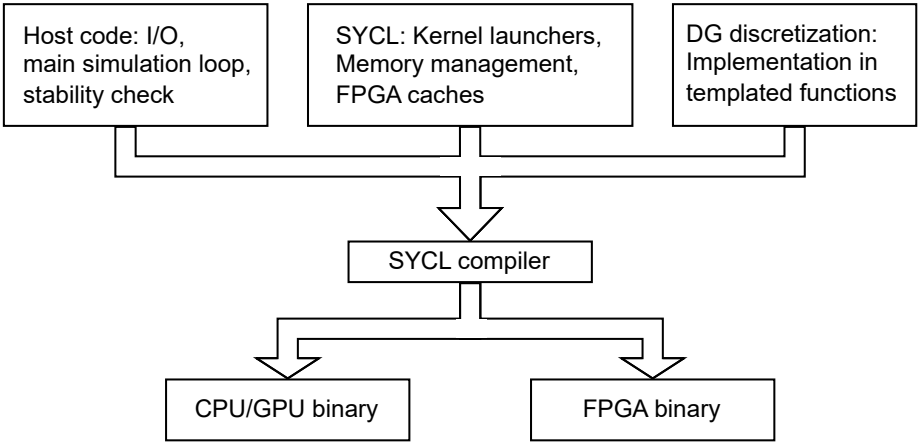
UNIVERSITÄT BAYREUTH

- Major limitation: off-chip memory bandwidth (~69 GB/s measured in stream benchmark)
- Previously shown (Kenter et al., 2021): High performance possible for small meshes, but mesh size fixed during compilation

→ Design with high performance for small meshes and no limitation on mesh size?

- Implement custom caches as C++ classes
- Fixed part of mesh is loaded into on-chip buffers
- Cache size depends on approximation order

Code design


Host code: I/O,
main simulation loop,
stability check
SYCL: Kernel launchers,
Memory management,
FPGA caches
DG discretization:
Implementation in
templated functions
SYCL compiler
CPU/GPU binary
FPGA binary

Summary: Target specific optimizations

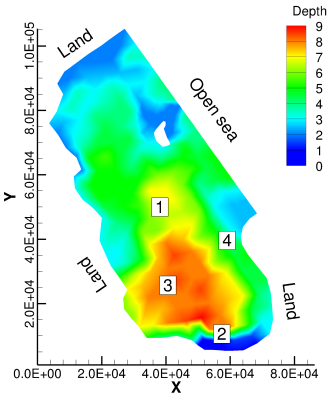
UNIVERSITÄT BAYREUTH

	GPU	CPU	FPGA
kernel type		parallel for	single task
unrolled loops		18 ×	18 × + 24 ×
DOF layout		blockwise Struct of Arrays	Array of Structs
caches		hardware feature	C++ classes

Explicit unrolled loops needed on Intel Ponte Vecchio GPUs with oneAPI compiler, not for other GPUs or compilers

Results: Accuracy, performance, portability

Test domain



Bight of Abaco, Bahamas

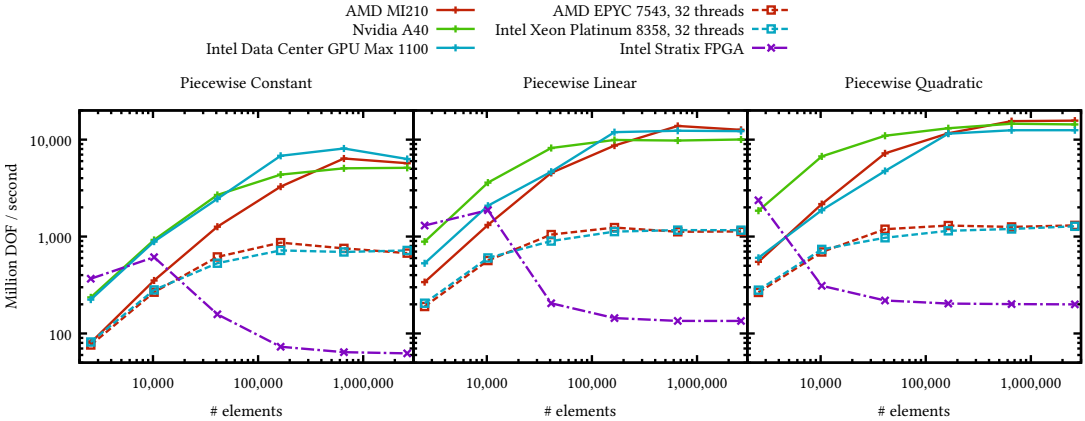
- Mesh sizes between 2,000 and 2,600,000 elements
- Boundary conditions:
  - No normal flow at land boundaries
  - Prescribed water elevation at open sea boundary
- Single precision
- Reference solutions generated by sequential version of UTBEST

UNIVERSITÄT BAYREUTH

Intel Xeon Platinum 8358 CPU and AMD MI 210 GPU
maximum difference for water elevation and velocity

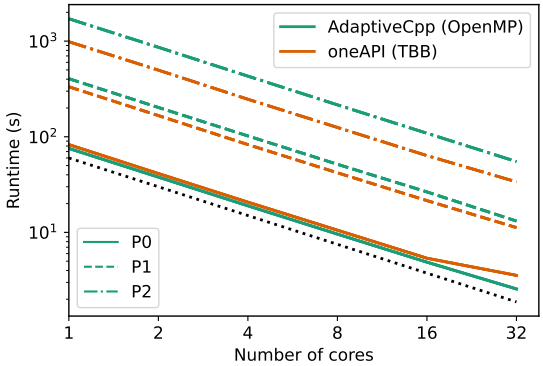
Compiler	Variable	P0	P1	P2
oneAPI CPU	$\Delta\xi$, m	$1.3 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$
	$\lvert\Delta q\rvert$, m^2/s	$7.2 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$	$1.3 \cdot 10^{-4}$
oneAPI CPU	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.7 \cdot 10^{-7}$	$5.4 \cdot 10^{-6}$
`-fp-model=precise`	$\lvert\Delta q\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$8.7 \cdot 10^{-5}$
oneAPI GPU	$\Delta\xi$, m	$2.0 \cdot 10^{-7}$	$3.4 \cdot 10^{-7}$	$1.2 \cdot 10^{-5}$
	$\lvert\Delta q\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$6.5 \cdot 10^{-5}$
oneAPI FPGA	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.4 \cdot 10^{-7}$	$1.5 \cdot 10^{-5}$
	$\lvert\Delta q\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$8.6 \cdot 10^{-6}$	$6.1 \cdot 10^{-5}$
ACPP CPU	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.3 \cdot 10^{-7}$	$1.2 \cdot 10^{-5}$
	$\lvert\Delta q\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$8.9 \cdot 10^{-6}$	$4.5 \cdot 10^{-5}$
ACPP GPU	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$4.1 \cdot 10^{-7}$	$8.4 \cdot 10^{-6}$
	$\lvert\Delta q\rvert$, m^2/s	$1.0 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$6.5 \cdot 10^{-5}$

Performance portability on CPUs, GPUs and FPGAs

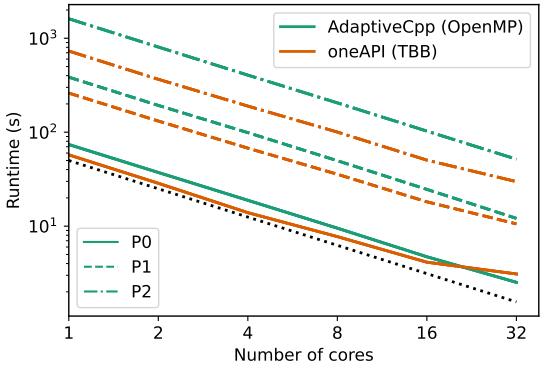


Compiled with Intel oneAPI 2023.2

CPUs: Strong scaling



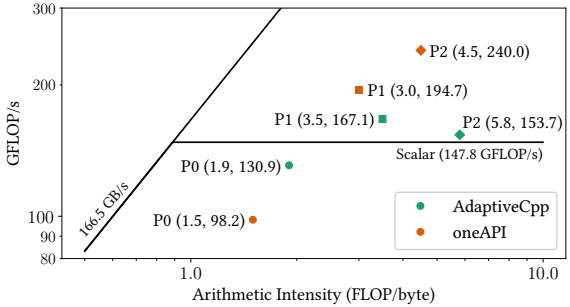
(a) Intel Xeon Platinum 8358 ("Icelake")



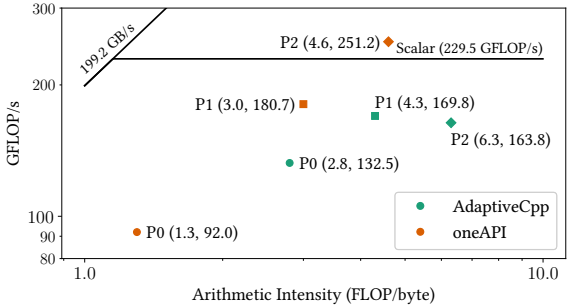
(b) AMD EPYC 7543 ("Milan")

- Scaling tests compute a total of 1000 time steps
- Good strong scaling on both CPUs for oneAPI and AdaptiveCpp

CPUs: Roofline models



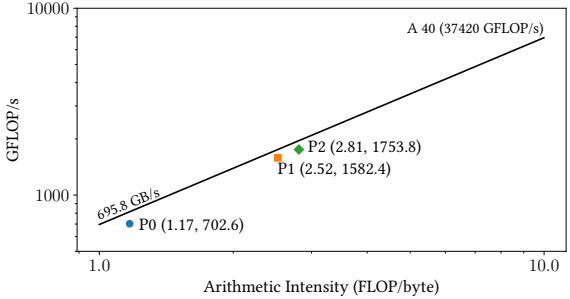
(a) Intel Xeon Platinum 8358 ("Icelake")



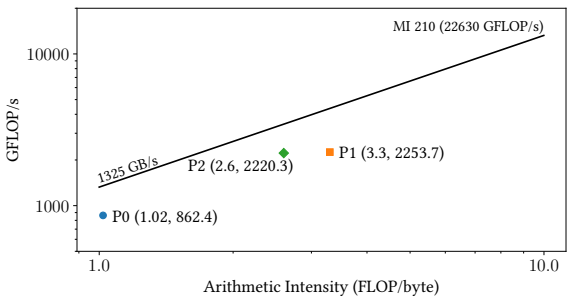
(b) AMD EPYC 7543 ("Milan")

- oneAPI has higher performance especially for P1 and P2
- AdaptiveCpp has higher arithmetic intensity and lower performance

GPUs: Roofline models for AMD and Nvidia GPUs



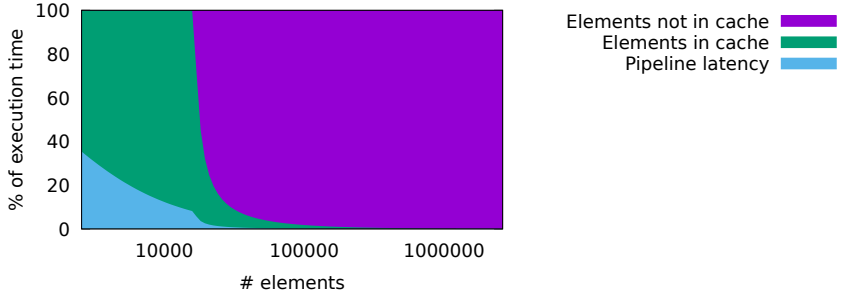
(a) Nvidia A 40



(b) AMD MI 210

- Compiler: oneAPI 2023.2
- 90 % memory bandwidth utilization on A 40, 65 % on MI 210
- P2 has lower arithmetic intensity on MI 210 than P1

FPGA pipeline performance



Model: Pipeline latency + No. elements in cache $\cdot 1 \frac{cycle}{element}$ + Remaining elements $\cdot 12 \frac{cycles}{element}$

- Pipeline latency significant for small meshes
- Processing time of elements outside of cache dominates for larger meshes
- Peak performance when pipeline latency is proportionally low, all elements in cache

Summary and Outlook

UNIVERSITÄT BAYREUTH

- First shallow water solver targeting CPUs, GPUs and FPGAs from same codebase
- Numerical algorithm separated from memory access patterns
- Separate kernel launch code for CPUs/GPUs and FPGAs
- Good scalability on current generation Intel and AMD CPUs
- Comparable performance on Nvidia, AMD and Intel data center GPUs
- Performance can vary significantly between compilers
- Custom-designed, optional caches give high throughput for low mesh sizes without imposing hard limit on number of elements

Ongoing and future work includes:

- ARM CPUs
- FPGAs with High Bandwidth Memory
- More performance analysis and optimizations

Thank you

UNIVERSITÄT BAYREUTH

Questions?

Acknowledgements:

- Research funded by Deutsche Forschungsgemeinschaft (DFG) under grants AI 117/7-1 and KE 2844/1-1.
- NHR@FAU, PC^2: compute resources