juliacon

High-performance xPU Stencil Computations in Julia

Samuel Omlin[1] and Ludovic Räss[2, 3]

[1]Swiss National Supercomputing Centre (CSCS), ETH Zurich, Lugano, Switzerland
[2]Laboratory of Hydraulics, Hydrology and Glaciology (VAW), ETH Zurich, Zurich, Switzerland
[3]Swiss Federal Institute for Forest, Snow and Landscape Research (WSL), Birmensdorf, Switzerland

ABSTRACT

We present an efficient approach for writing architecture-agnostic parallel high-performance stencil computations in Julia, which is instantiated in the package `ParallelStencil.jl`. Powerful metaprogramming, costless abstractions and multiple dispatch enable writing a single code that is suitable for both productive prototyping on a single CPU thread and high performance production runs on GPU or CPU workstations or, if used in combination with distributed parallelization packages as `ImplicitGlobalGrid.jl`, on supercomputers. We demonstrate performance close to the theoretical upper bound on GPUs for a 3-D heat diffusion solver, which is a massive improvement over reachable performance with `CUDA.jl` Array programming.

Keywords

Julia, xPU, GPU, Stencil Computations, Code Generation, Architecture-agnostic, Shared Memory Parallelization, Communication-Computation Overlap, Supercomputing

1. Introduction

Graphics processing units (GPUs) capable of general-purpose computing have revolutionized the hardware industry and as a result High Performance Computing (HPC) since the dawn of the 21st century. While industry and academia are doing their best to adapt their software to the new hardware landscape, the latter continues to be reshaped constantly. In addition, new unconventional highly innovative hardware developments driven by the powerful AI industry (e.g. the Cerebras WSEs and Graphcore IPUs) draw up yet the next potential hardware revolution. In the light of the high pace and increasing diversity in hardware evolution, the HPC community has identified the 3 "P"s - (scalable) Performance, (performance) Portability and Productivity - as fundamental requirements for today's and tomorrow's software development. The approach and package development presented in this paper responds to each of the 3 "P"s. We present an approach for automatic parallelization and optimization of architecture-agnostic stencil computations deployable on both GPU and CPU (in the remainder we use xPU to refer simultaneously to GPU and CPU); the computations can furthermore automatically hide the communication needed for distributed parallelization as required for large scale supercomputing (note that the distributed parallelization itself is not part of this contribution).

```julia
using ParallelStencil
using ParallelStencil.FiniteDifferences3D
@init_parallel_stencil(CUDA, Float64, 3)

@parallel memopt=true optvars=T function step!(
    T2, T, Ci, lam, dt, _dx, _dy, _dz)
    @inn(T2) = @inn(T) + dt*(
        lam*@inn(Ci)*(@d2_xi(T)*_dx^2 +
                      @d2_yi(T)*_dy^2 +
                      @d2_zi(T)*_dz^2 ) )
    return
end

function diffusion3D()
    # Physics
    lam       = 1.0           #Thermal conductivity
    c0        = 2.0           #Heat capacity
    lx=ly=lz  = 1.0           #Domain length x|y|z

    # Numerics
    nx=ny=nz  = 512           #Nb gridpoints x|y|z
    nt        = 100           #Nb time steps
    dx        = lx/(nx-1)     #Space step in x
    dy        = ly/(ny-1)     #Space step in y
    dz        = lz/(nz-1)     #Space step in z
    _dx, _dy, _dz = 1.0/dx, 1.0/dy, 1.0/dz

    # Initial conditions
    T  = @ones(nx,ny,nz).*1.7 #Temperature
    T2 = copy(T)              #Temperature (2nd)
    Ci = @ones(nx,ny,nz)./c0  #1/Heat capacity

    # Time loop
    dt = min(dx^2,dy^2,dz^2)/lam/maximum(Ci)/6.1
    for it = 1:nt
        @parallel memopt=true step!(
            T2, T, Ci, lam, dt, _dx, _dy, _dz)
        T, T2 = T2, T
    end

end

diffusion3D()
```

Fig. 1: Stencil-based 3-D heat diffusion xPU solver implemented using ParallelStencil with time step kernel written in math-close notation.

2. Approach

Our approach for the expression of architecture-agnostic high-performance stencil computations relies on the usage of Julia's powerful metaprogramming capacities, costless high-level abstractions and multiple dispatch. We have instantiated the approach in

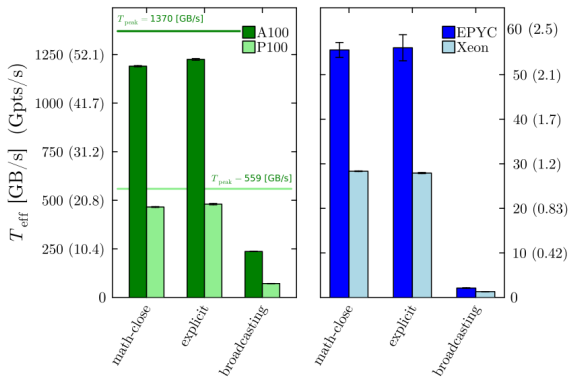

Fig. 2: Effective memory throughput T_{eff} (and conversion to Gpts/s) for Nvidia Tesla A100 SXM4 and P100 PCIe GPUs, respectively, and for 2 x 16 Core AMD EPYC 7282 and 12 Core Intel Xeon E5-2690 v3 CPUs, respectively. The error bars visualize the 95% confidence interval of the reported medians (20 samples). The raw data and plotting script are available in `github.com/omlins/ParallelStencil.jl/tree/JuliaConProceeding2022/paper`.

the Julia package `ParallelStencil.jl`. Using ParallelStencil, a simple call to the macro `@parallel` is sufficient to parallelize and launch a kernel that contains stencil computations, which can be expressed explicitly or with math-close notation. The latter is defined in isolated submodules (e.g., line 2) that are easily understandable and extensible by domain scientists in order to support new numerical methods (currently available is math-close notation for finite differences). Fig. 1 shows a stencil-based 3-D heat diffusion xPU solver implemented using ParallelStencil, where the kernel defining an explicit time step is written in math-close notation (lines 5-12) and the macro `@parallel` is used for its parallelization (line 5) and launch (line 36).

The package used underneath for parallelization is defined in a initialization call beforehand (Fig. 1, line 3). Currently supported are `CUDA.jl` [1] for running on GPU, and `Base.Threads` for CPU. Leveraging metaprogramming, ParallelStencil automatically generates high-performance code suitable for the target hardware, and automatically derives kernel launch parameters from the kernel arguments by analyzing the bounds of the contained arrays. Certain stencil-computation-specific optimizations leveraging, e.g., the on-chip memory of GPUs need to be activated with keyword arguments to the macro `@parallel` (Fig. 1, line 5). A set of architecture-agnostic low level kernel language constructs allows for explicit low level kernel programming when useful, e.g., for the explicit control of shared memory on the GPU (these low level constructs are GPU-computing-biased).

Arrays are automatically allocated on the hardware chosen for the computations (GPU or CPU) when using the allocation macros provided by ParallelStencil (Fig. 1, lines 29-31), avoiding any need of code duplication. Moreover, the allocation macros are fully declarative in order to let ParallelStencil choose the best data layout in memory. Notably, logical arrays of structs (or of small arrays) can be either laid out in memory as arrays of structs or as structs of arrays accounting for the fact that each of these allocation approaches has its use cases where it performs best.

ParallelStencil is seamlessly interoperable with packages for distributed parallelization, as e.g. `ImplicitGlobalGrid.jl` [4] or `MPI.jl`, in order to enable high-performance stencil computations on GPU or CPU supercomputers. Communication can be hidden behind computation with as simple macro call [4]. The usage of this feature solely requires that communication can be triggered explicitly as it is possible with, e.g., ImplicitGlobalGrid and `MPI.jl`.

3. Results

We here report the performance achieved on different architectures with the 3-D heat diffusion xPU solver (Fig. 1) and of an equivalent solver with explicit notation for the stencil computations and compare it to the performance obtained with a Julia solver written in a traditional way using GPU or CPU array broadcasting. We observe that using ParallelStencil we achieve an effective memory throughput, T_{eff}, of 496 GB/s and 1262 GB/s on the Nvidia P100 and A100 GPUs, which can reach a peak throughput, T_{peak}, of 559 GB/s and 1370 GB/s, respectively [2]; this means we reach 89% and 92% of the respective hardware's theoretical performance upper bound (T_{eff} and its interpretation are explained, e.g., in [6]). Furthermore, using ParallelStencil we obtain a speedup of up to a factor ≈ 5 and ≈ 29 over the versions with GPU and CPU array broadcasting (the latter is not capable of multi-threading), respectively. Moreover, we have translated solvers for highly nonlinear 3-D poro-visco-elastic two-phase flow and 3-D reactive porosity waves written in CUDA C using MPI to Julia by employing ParallelStencil (and ImplicitGlobalGrid for the distributed parallelization) and compared obtained performance. The translated solvers achieved 90% and 98% of the performance of the respective original CUDA C solvers. In addition, relying on ParallelStencil's feature to hide communication behind computation, the 3-D poro-visco-elastic two-phase flow solver achieved over 95% parallel efficiency on up to 1024 GPUs [4].

4. Conclusions

We have shown that ParallelStencil enables scalable performance, performance portability and productivity and responds to the challenge of addressing the 3 "P"s in all of its aspects. Moreover, we have outlined the effectiveness and wide applicability of our approach within geosciences. Our approach is naturally in no sense limited to geosciences as stencil computations are commonly used in many disciplines across all of science. We illustrated this in recent contributions, where we showcased a computational cognitive neuroscience application modelling visual target selection using ParallelStencil and `MPI.jl` [5] and a quantum fluid dynamics solver using the nonlinear Gross-Pitaevski equation implemented with ParallelStencil (and ImplicitGlobalGrid) [3].

5. Acknowledgments

This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID c23 through the Platform for Advanced Scientific Computing (PASC) program. We acknowledge A100 DGX-1 computing resources at VAW, ETH Zurich.

6. References

[1] T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2018. doi:10.1109/TPDS.2018.2872064.

[2] Tom Deakin, Andrei Poenaru, Tom Lin, and Simon McIntosh-Smith. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC*

(P3HPC), pages 1–13, GA, USA, November 2020. IEEE. doi:10.1109/P3HPC51967.2020.00006.

[3] S. Omlin, L. Räss, N. Keepfer, G. Kwasniewski, B. Malvoisin, and Y. Y. Podladchikov. Solving Nonlinear Partial Differential Equations on GPU Supercomputers Using Julia. PASC21 conference, 2021.

[4] S. Omlin, L. Räss, and I. Utkin. Distributed Parallelization of xPU Stencil Computations in Julia. *Proc. JuliaCon Conf.*, page 2, 2022. doi:10.48550/arXiv.2211.15716.

[5] S. Omlin, L. Räss, I. Utkin, V. Narayanan, and M. Senden. Development of Multi-GPU Solvers for Nonlinear Multi-Physics with Julia. PASC22 conference, 2022.

[6] L. Räss, I. Utkin, T. Duretz, S. Omlin, and Y. Y. Podladchikov. Assessing the robustness and scalability of the accelerated pseudo-transient method. *Geoscientific Model Development*, 15(14):5757–5786, 2022. doi:10.5194/gmd-15-5757-2022.