SCA/ HPCAsia 2026

Investigation of GPU Programming Paradigms with regard to Code Complexity and Performance Portability

Jonas Schuhmacher[(1)], Hans-Joachim Bungartz[(1)]

[(1)]Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany, **jonas.schuhmacher@tum.de** | bungartz@cit.tum.de

Research Question

- How **Performant** is the paradigm?
- How **Portable** is the paradigm?
- How **Productive** am I using this paradigm?

Performance Portability

Definition by Pennycook et al. (2019), for application a solving problem p:

Across different Compute Platforms $H = \{h_1, h_2, \dots, h_n\}$

Performance Portability $\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{h \in H} \frac{1}{e_h(a,p)}} & \text{if } e_h \text{ is available on } H \\ 0 & \text{otherwise} \end{cases}$

or

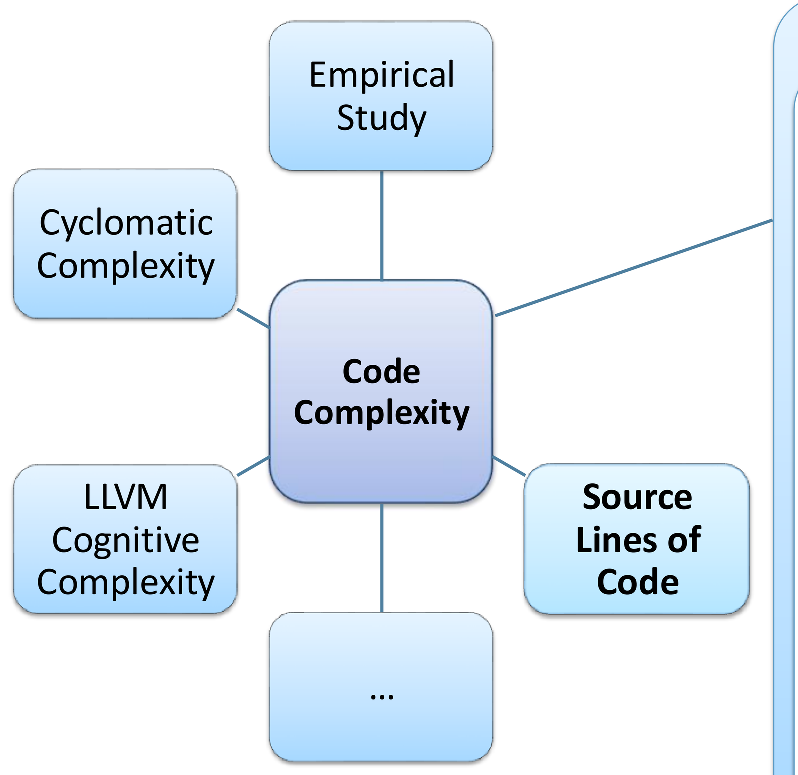
On the same Platform $h \in H$

- Architectural Efficiency e [%] — Fraction of theoretical peak hardware performance on h
- Application Efficiency e [%] — Achieved Performance as a fraction of the best observed value on h

Example Application Efficiency

A Matrix Multiplication p is developed in 2 different paradigms and benchmarked with an RTX5080 (platform h). Application a takes 10s, Application b takes 2s; then $e_h(a,p) = 20\%$ and $e_h(b,p) = 100\%$ since it was the best observed value.

Code Complexity



Halstead Complexity

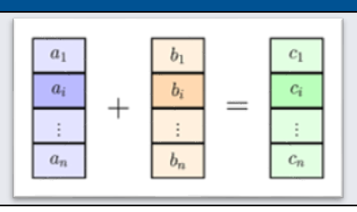
- Accounts not only for program length, but also takes vocabulary variance into consideration
- Solves the issue that a developer needs to "learn" a function only once, e.g., five times `cudaMalloc()`

Number of (distinct) operators/ operands	n_1, n_2, N_1, N_1
Program Vocabulary n	$n_1 + n_2$
Program Length N	$N_1 + N_2$
Volume V	$N \cdot \log_2 n$
Difficulty D	$\frac{n_1}{2} \cdot \frac{N_2}{n_2}$
Effort E	$D \cdot V$

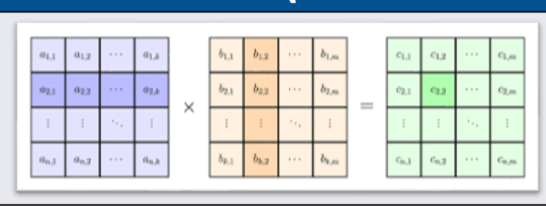
Example: `int x = increment(1);`

Methodology

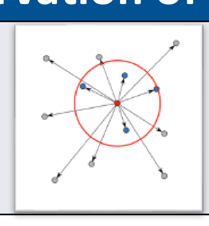
4 different problems p with variable size n (to also allow the observation of how performance/ efficiency scales in the paradigm)



Vector Addition

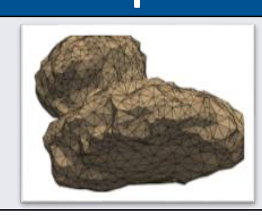


Matrix Multiplication



$U(x_i, x_j) = 4\epsilon\left(\left(\frac{\sigma}{\|x_i - x_j\|_2}\right)^{12} - \left(\frac{\sigma}{\|x_i - x_j\|_2}\right)^{6}\right)$

Pairwise (MD) N-Body Simulation



$V \sim \sum_{faces} \sum_{segments} \dots$

ESA's Polyhedral Gravity Model

Implement these four problems in all GPU paradigms using the <u>same</u> feature subset and verify correctness

Benchmark them on four platforms calculating Application Efficiency e_h for every framework on a single platform given a pair (p, n)

RTX 2080 | RTX 3080 | RTX 4060 | RTX 5080

Calculate the Performance Portability Φ

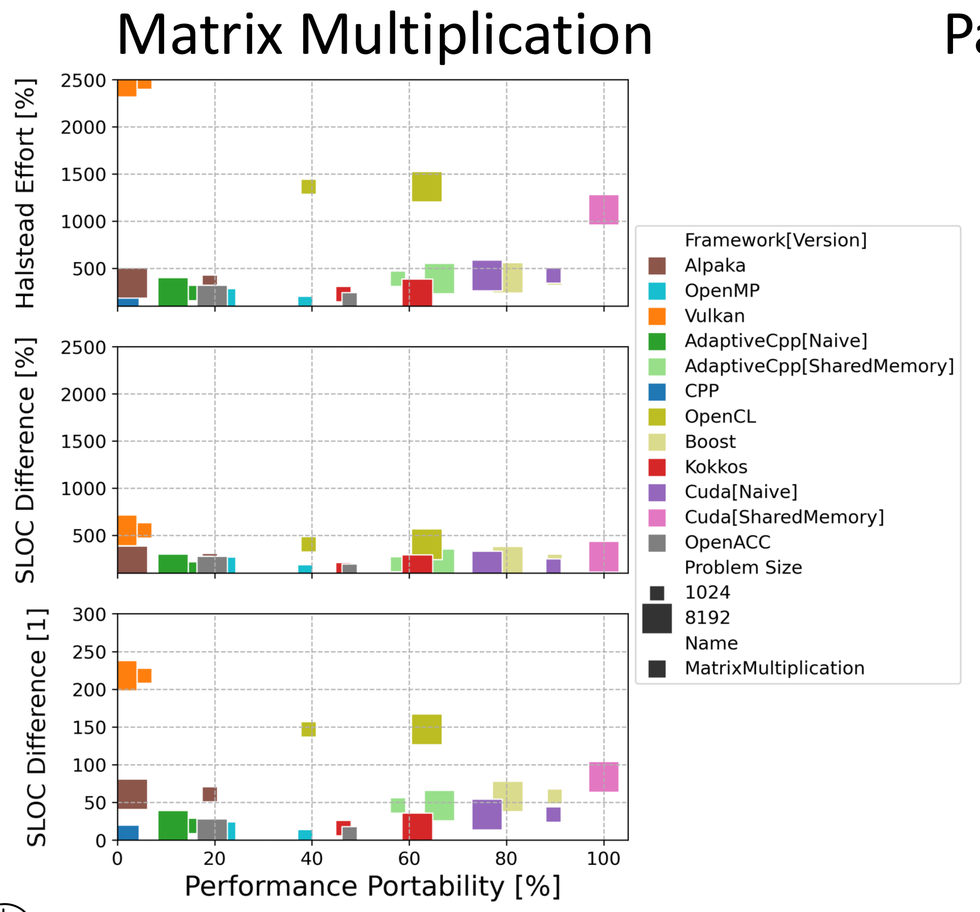
Measure the Code Complexity

Source Lines of Code (SLOC) | Halstead Complexity

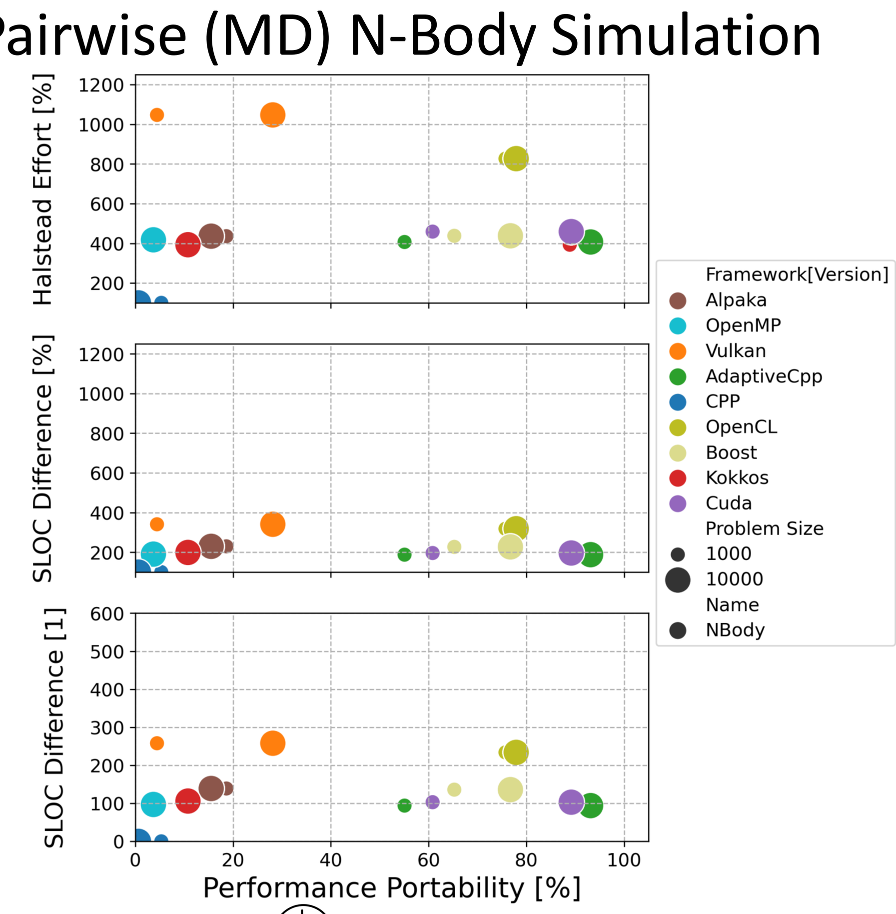
Normalize SLOC and Halstead Effort comparing them to a CPU-only implementation

Results

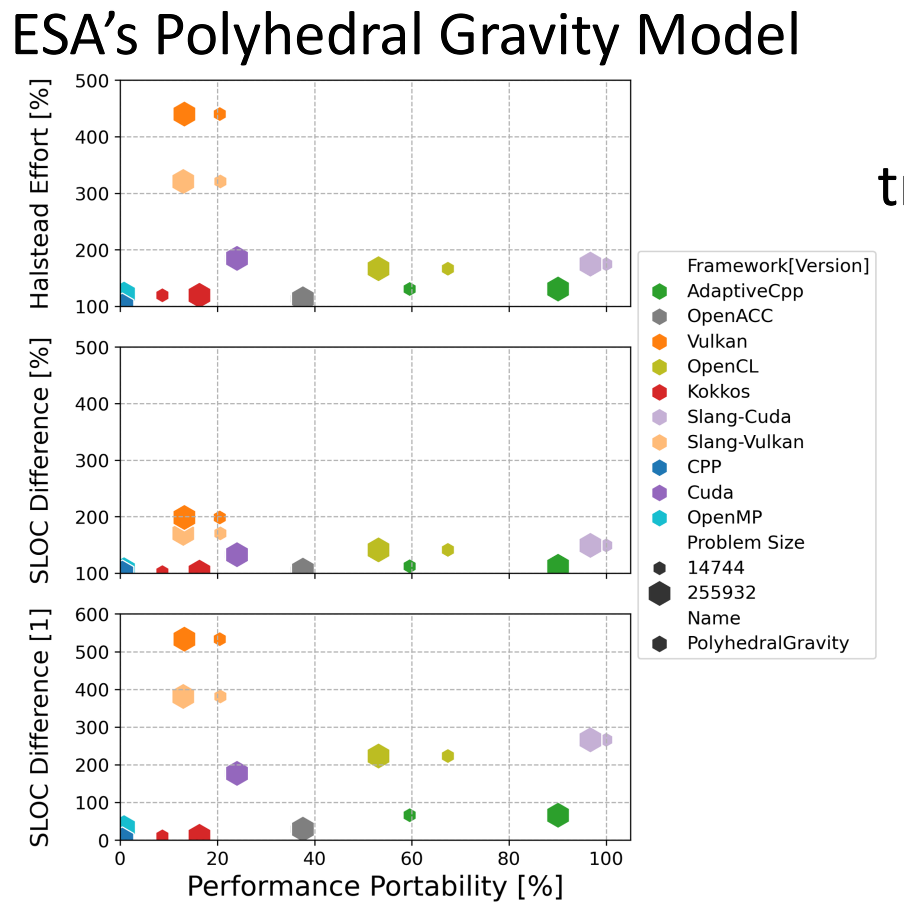
Matrix Multiplication



Pairwise (MD) N-Body Simulation



ESA's Polyhedral Gravity Model



More Results?
Have a look and
try it out on your platform!

Key Insights Performance Portability:

- Different GPU paradigms perform differently on the same platform **despite being similar syntactic implementations.**
- A simple algorithm in a native paradigm like Cuda usually outperforms portable frameworks introducing certain abstraction, but…
  - low-level paradigms like OpenCL can come close.
  - paradigms offering more control can also unlock performance.
  - the shader language `Slang` compiled to Cuda outperforms the native solution in case of a polyhedral gravity model
- **Graphics APIs can be exploited for general purpose computation** with partial success (see Polyhedral Model with `Slang`)

Key Insights Code Complexity:

- Pragma-based approaches (OpenMP, OpenACC) are the simplest ways to get an algorithm to the GPU, but far from performant
- **`AdaptiveCpp` (`hipSYCL`) and Kokkos** (though `Kokkos` struggled, e.g., on the RTX4060 with performance using the same code base) **offer the best performance per Line of Code to Learn ratio**
- **Abstraction doesn't always mean slower** (e.g. `Boost-Compute` vs. OpenCL)

Limitations and Future Work:

- Testing on AMD systems is work-in-progress, but still largely yet to be done
- Code Complexity **does not factor in effort to set-up a working compiler infrastructure** (e.g., `Kokkos` mostly working out of the box, `AdaptiveCpp` requiring a custom LLVM installation)
- Exploring the shading language `Slang` in more detail
- Exploring technical more competitive implementations (shared memory usage, etc.)
- Exploring different algorithms (e.g. LinkedCells for particle simulation)

Related Work:

Schuhmacher, J., Blazquez, E., Gratl, F., Izzo, D., & Gómez, P. (2024). Efficient Polyhedral Gravity Modeling in Modern C++ and Python. Journal of Open Source Software, 9(98), 6384. https://doi.org/10.21105/joss.06384

Brase, R. & Schuhmacher, J., (2025). Investigation of Parallelization Paradigms regarding Performance and Productivity in Context of a Polyhedral Gravity Model, Technical Report, Technische Universität München, https://mediatum.ub.tum.de/1781596

Ispas, M. (2025). Implementation and Comparison of Code Complexity Metrics to Assess Developer Productivity, Bachelor's Thesis, Technische Universität München, https://mediatum.ub.tum.de/1839972