Chair of Scientific Computing
TUM School of Computation, Information and Technology
Technical University of Munich

Investigation of Parallelization Paradigms regarding Performance and Productivity in Context of a Polyhedral Gravity Model

Robin Brase ✉ **and Jonas Schumacher**
TUM School of Computation, Information and Technology, Technical University of Munich
✉ robin.brase@tum.de
April 27, 2025

Abstract — Modern scientific computing is relying more and more on GPU acceleration to meet the growing demands of computation. This paper compares parallelization frameworks, including performance-portable solutions (AdaptiveCpp, Kokkos, OpenCL, OpenMP, OpenACC) and graphics APIs (Vulkan, SLANG, WebGPU), using a polyhedral gravity model as a test case. We evaluate performance gains, numerical accuracy with different floating-point precisions, and implementation effort.

Results on consumer-level GPUs show that SLANG compiled to CUDA can achieve a speedup of over 250× when using single precision floats, followed by OpenCL and Kokkos. However, we also show that single precision floats can lead to significant deviations in the results, which particularly affects graphic APIs.

1 Introduction

Today, almost every field of science needs fast and efficient computing to analyze large amounts of data. However, CPU performance improvements have slowed down in recent years, making GPU acceleration more important than ever.

This need has led to the development of several frameworks looking to simplify the development of programs using the GPU for general purpose computing. Another goal of such frameworks is the ability to use the same code for GPUs from different manufacturers to run on as many machines as possible.

Many comparisons [1, 2], solely focus on such compute frameworks and do not include frameworks geared towards graphics programming such as Vulkan [3] or SLANG [4]. However, these also offer the possibility of general purpose computation through compute shaders and are therefore also investigated in this work to provide a comprehensive comparison.

Our comparison is made in the context of a polyhedral gravity model, which has been previously implemented in a TUM/ ESA collaboration[1] [5, 6].

The model is re-implemented using the parallelization frameworks mentioned above, to study the performance increase, their numerical accuracy using different floating point precisions as well as the implementation effort required. Given their performance and complexity, this investigation creates a reference for the suitability of GPU programming paradigms for application scientists to inspire uncomplicated accelerated scientific software.

2 Polyhedral Gravity Model

The problem we investigate in this paper is the calculation of the gravity field using a polyhedral gravity model based on the line integral approach proposed by Petrović [7] and the refined version by Tsoulis and Petrović [8].

For an arbitrary point P, this model calculates the gravitational potential V:

$$V = G\rho \iiint_U \frac{1}{l}\, dU \tag{1}$$

as well as the attraction V_{x_i} (with $i = 1, 2, 3$)

$$V_{x_i} = G\rho \iiint_U \frac{\partial}{\partial x_i}\left(\frac{1}{l}\right) dU \tag{2}$$

and the gradiometric tensor $V_{x_i x_j}$ $(i, j = 1, 2, 3)$:

$$V_{x_i x_j} = G\rho \iiint_U \frac{\partial^2}{\partial x_i \partial x_j}\left(\frac{1}{l}\right) dU \tag{3}$$

where U is the volume, ρ the density, G the gravitational constant and l the Cartesian distance of P to the origin [9].

[1] https://github.com/esa/polyhedral-gravity-model, last accessed: 13.04.2025

For the calculation with an arbitrary polygon mesh, these triple integrals are converted into a double summation, where the outer one sums over all faces p and the inner one over the respective segments q [7, 8]. For the attraction V_{x_i}, for example, the resulting formula then looks like the following:

$$V_{x_i} = G\rho \cdot \sum_{p=1}^{n} \cos(\overrightarrow{N_p}, \overrightarrow{e_i}) \cdot \left[\sum_{q=1}^{m} \sigma_{pq} h_{pq} LN_{pq} + h_p \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + sing_{A_p} \right] \tag{4}$$

The equations for the other quantities as well as the exact meanings of the contained symbols can be found in the paper by Tsoulis and Petrović [8]. What is relevant for this paper is that, m (number of segments) is always three for a mesh consisting only of triangles and that the calculation of some of the inner variables (e.g. LN_{pq} and AN_{pq}) requires the evaluation of transcendental functions such as the logarithm or the inverse tangent.

3 Parallelization Frameworks

GPU parallelization frameworks allow developers to use the parallel processing capabilities of GPUs for general-purpose computing. One approach to GPU programming is through compute shaders, which originated from graphics APIs (e.g. OpenGL) which were initially designed for rendering pipelines but were extended to allow for general purpose computing while still integrating with existing graphics APIs and hardware [10]. However, most graphics applications only require floats with single precision, which is why such frameworks usually have only limited support for double precision floats [3].

Another disadvantage is the weak coupling between the code that is executed on the host-side (CPU) and the code for the GPU, which usually has to be written in a special shader language. There are no static checks to ensure that enough parameters or the correct data types are passed when invoking code on the GPU.

Specialized compute frameworks offer more direct and flexible approaches to GPU parallelization. These frameworks are designed specifically for general-purpose GPU computing, providing libraries, APIs, and runtime environments optimized for high-performance tasks [11].

There are different approaches to achieve this beyond native frameworks like CUDA [11], which provide low-level control over on GPU but are vendor specific. Library-based frameworks (e.g., Kokkos [12, 13]) provide easy-to-use functions and high-level API for common tasks. Compiler-based frameworks (e.g., AdaptiveCpp [14]) translate existing C++ code into GPU-executable instructions. Pragma-based systems (e.g., OpenMP [15]) use directives inserted into existing code to automate parallelization.

3.1 AdaptiveCpp

AdaptiveCpp (formerly hipSYCL/Open SYCL) [14, 16] is an SYCL implementation that enables cross-architecture execution of C++ code on CPUs, GPUs, and accelerators. SYCL is an open standard developed for heterogeneous computing developed by the Khronos Group [17]. The standard makes it possible to write both the host side and the device code in standard C++ within a single source file.

AdaptiveCpp supports execution on multiple target architectures, including NVIDIA GPUs, AMD GPUs, Intel GPUs, OpenCL devices, and CPUs via LLVM.

Developers can either specify the backend during compilation or use the Just-in-Time (JIT) compilation feature to automatically select the optimal backend at runtime based on the available hardware.

As an alternative to the low-level sycl operations, AdaptiveCpp also supports automatic offloading for the standard C++ parallel algorithms like `std::for_each` or `std::reduce` [18].

It is mainly developed at the university of Heidelberg with additional contributions by the open source community.

3.2 CUDA

CUDA (Compute Unified Device Architecture) [11] is a parallel computing platform and API developed by NVIDIA used for general-purpose programming on GPUs. It extends C/C++ with constructs for GPU programming and a GPU compatible implementation of the C++ standard library (libcudacxx). The SDK also provides higher level abstraction (CUB/Thrust) which simplify the implementation of common operations such as reduction algorithms[19]. CUDA is widely adopted in scientific computing and AI [20].

3.3 Kokkos

Kokkos [12, 13] is a parallel programming model and C++ library that simplifies performance portability across various high-performance computing (HPC) architectures, such as CPUs, GPUs, and accelerators. It introduces abstractions for parallel execution and memory management, allowing developers to write code that performs efficiently on different hardware platforms. Kokkos includes high-level tools for parallel patterns like loops, reductions, and scans. The supported backends are CUDA, HIP, SYCL, HPX, OpenMP, and C++ threads [13].

3.4 OpenACC

OpenACC [21] is a directive-based programming model designed to accelerate computational workloads on GPUs and other accelerators. It is supported by multiple compilers such as the NVIDIA HPC SDK [22] and partially in GCC [23]. The model works by inserting pragmas into the existing code, for example to execute for-loops in parallel on the GPU. While reductions are available for standard data types, they are not yet supported for custom data types [21].

3.5 OpenCL

OpenCL [24] (Open Computing Language) is a framework for parallel programming on heterogeneous platforms, developed by the Khronos Group. It offers a C-like (and C++-like in the newest versions) language for writing kernels which can be executed on the target device (e.g. CPU, GPU, FPGAs, etc.).

The kernels are compiled at runtime using the OpenCL driver provided by hardware vendor. These are then launched via the host side API, which is also used to manage the memory [25].

While OpenCL does not offer higher level APIs for common operations such as reductions, as of version 2.0 there are features for reductions within the individual workgroups inside the kernels [25].

3.6 GLSL & Vulkan

Vulkan Compute Shaders are a component of the Vulkan API, designed for general-purpose parallel computation on GPUs [3]. They are written in high-level shading languages such as GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language) [26], which are compiled into SPIR-V (Standard Portable Intermediate Representation), a binary intermediate representation. SPIR-V serves as a common representation of shader code, which can be optimized and executed across different hardware [27].

On the host side, Vulkan's API is highly explicit and verbose, requiring developers to manage memory allocation, synchronization, and pipeline setup.

3.7 OpenMP

OpenMP [15] is a pragma-based parallel programming model for shared-memory multiprocessing, designed to parallelize existing code by inserting compiler directives (pragmas). These pragmas instruct the compiler to distribute work across multiple threads, typically for parallelizing loops. It has been extended to support GPU computation through its target offloading features, introduced in OpenMP 4.0. The generation of device-specific code (e.g. CUDA for NVIDIA GPUs or HIP for AMD GPUs) is done by the compiler while the host side API remains minimal. [28]

Target offloading is supported by major compilers, including GCC, LLVM, and Intel's compiler suite, and is compatible with a range of GPU architectures from vendors like NVIDIA, AMD, and Intel [29].

3.8 SLANG

The SLANG shading language is a extension to HLSL developed by NVIDIA to provide modern language features such as generics or interfaces [4]. Just like SYCL and Vulkan, this project is now managed by the Khronos Group [30].

Compute shaders written in SLANG can be compiled to multiple backends, most notably CUDA and Vulkan compute shaders (using SPIR-V). Compilation to Metal (to support apple computers), CPU side C++ code and WebGPU is also possible, but currently still at an experimental level. [31]

3.9 WebGPU

WebGPU is an API designed to enable GPU acceleration on the web, providing access to graphics and compute capabilities [32]. It supports compute shaders, which are written in the WebGPU Shading Language (WGSL) [33], a language created for this use with a Rust like syntax.

While WebGPU is primarily intended for web browsers, implementations like Dawn (used in Google Chrome) [34] and wgpu (written for Firefox) [35] allow it to be used outside the browser.

4 Implementation

The respective implementations basically consist of a single class (GravityEvaluable) outlined in Listing 1 and its evaluate method, which calculates the gravity at the given point. The required GPU memory and other needed resources, such as pipelines, are requested directly in the constructor and released in the destructor.

Since data is only copied to GPU memory once during initialization, our implementation is purely compute-bound and not limited by the memory bandwidth between the CPU and GPU.

```
struct GravityModelResult {
  FloatType potential;
  Array3 acceleration;
  Array6 gradiometricTensor;

 auto operator+(GravityModelResult rhs);
}

class GravityEvaluable {
public:
  GravityEvaluable(
    const std::vector<Array3> &Vertices,
    const std::vector<IndexArray3> &Faces,
    const double density
  );

  GravityModelResult evaluate(
    const Array3 Point
  );
};
```

Listing 1 Prototype of GravityModelResult class with its addition operator used for reduction and the GravityEvaluable class implemented with each framework.

What many frameworks have in common is that memory on the GPU has to be managed explicitly, with only minor differences in usage. C++ native frameworks often offer template-based classes that manage the memory automatically using the RAII concept, whereas in the implementations for frameworks with C-API such a class must be implemented by the user.

One major and relevant difference for us is the existence of mechanisms for efficient reductions, in our case the final summation of the results. Some frameworks offer high-level functions that are easy to use, while with others this has to be implemented manually.

4.1 Baseline

The baseline implementation is a simplified version of the implementation by Schuhmacher et al. [6]. In our implementation, the results calculated for each face are first stored in a vector, which is then reduced using `std::accumulate`.

4.2 High-Precision Baseline

To better measure the numerical precision of the different frameworks, we have implemented another baseline that uses floating point numbers with 128 bits. To use 128 bits float everywhere, this implementation replaces the mathematical functions of the C++ standard library (e.g. `std::log`) with Clang specific builtin functions (e.g. `__builtin_log`). To minimise errors during summation, such as cancellations, it uses the improved Kahan–Babuška algorithm introduced by Neumaier [36]. It does this by keeping track of rounding errors using a compensation variable.

4.3 AdaptiveCPP

Because AdaptiveCPP is an implementation of the sycl standard, the sycl functions can be used for parallel execution (`parallel_for`) and for reduction of the results (`sycl::reduction`). This reduction also supports custom data types, if an reduction function is passed to the constructor of the reduction object as a lamba function. In the parallel_for loop, the current result can then be added to the final result using `reducer.combine`, making it unnecessary to write to global memory.

4.4 CUDA

There are several levels at which a reduction can be implemented in CUDA, from low-level functions like `__shfl_down_sync` which operate on a warp-level to high level function like `thrust::reduce` which can operate on arrays in GPU memory. Thrust also supports custom data types, as long as the corresponding `operator+` can be executed on the GPU and is annotated with `__device__` (which excludes some classes from the C++ standard library). The low-level functions only support the primitive datatypes, for custom datatypes have to add up each member individually.

In our tests, there was also very little performance difference between these two approaches, which is why we opted for the thrust function, which keeps the code simple. This means, our kernel first writes the result for each face into a global array, which is then summed up in a second step using `thrust::reduce`.

4.5 Kokkos

Kokkos offers a simple way to perform a parallel reduction using `Kokkos::parallel_reduce`. Therefore only one pass is required and nothing has to be manually written to global memory. If the program is compiled for CUDA, then the `operator+` has to meet the same requirements as for `thrust::reduce`.

4.6 OpenAcc

Although OpenACC offers a reduction clause in its pragmas, this only works for the primitive data types. For this reason, we have to loop over each field in our results struct and reduce them individually.

For our benchmarks we used the nvc compilers from the NVIDA HPC SDK in version 25.1-0, while gcc could not compile our implementation due to some problems with member functions.

4.7 OpenCL

Because there are no high-level functions for the reduction in OpenCL, we use the workGroupFunctions[2] (`work_group_reduce_add`) introduced in OpenCL 2.0 to perform an initial reduction within each workgroup. These functions only support primitive data types, which is why the reduction must be performed for each of the ten individual values (one for potential, three for attraction, six for gradiometric tensor). Only the first thread of each workgroup then writes this partial result to an output array in GPU memory. In a second kernel, this is then further reduced on the GPU using the same technique before the remaining partial results are summed up on the CPU.

4.8 Vulkan & GLSL

There are also no high-level functions available for the Vulkan/GLSL implementation, which is why we use the `subgroupAdd` function to get first reduction within the workgroup. The partial results are reduced further in a second shader using the same function, before getting copied to the CPU for a final summation.

4.9 OpenMP

OpenMP offers a reduction clause for the `parallel for` pragma, with which the final summation can be performed. This also supports custom data types, as long as the reduction is declared using the `declare reduction` pragma.

4.10 SLANG

In the slang implementation, we use different summation methods, depending on the backend used. In the CUDA version, the results are written to a global array and summed using thrust reduce, similar to the native CUDA implementation. In the Vulkan variant, we perform a first reduction within the workgroup using the `WaveActiveSum` function and write the result for each group into a global buffer. This buffer is then copied to the CPU where the final summation is done.

4.11 WebGPU

Because the WebGPU implementation uses an older wgpu version that does not yet include subgroup functions such as `subgroupAdd`, the results are currently still summed on the CPU.

5 Benchmarks

The benchmarks results in Figure 1 shows the speedup factor of each of our implementations relative to the baseline implementation with 64-bit floating-points.

Where possible, we benchmarked a variant with 32 bit floats and one with 64 bit floats. We measured the average execution time to calculate the gravity at a single point, setup costs for e.g. the one-time setup of pipelines are not measured. The meshes used here were one for the asteroid 67P/Churyumov-Gerasimenko [37, 38] (with roughly 256 thousand faces) and one for Eros [39] with 14k faces.

Tests were run on a computer with an Intel i5-12400F (6 cores, 4.4 GHz) and an RTX 3070 (Driver version: 550.120) using Google Benchmark 1.9.0 under Ubuntu 24.04 (kernel 6.11.0-19).

The benchmark results show that implementations using single-precision (32-bit) floating-point arithmetic achieve significantly higher performance compared to their double-precision (64-bit) counterparts. This performance advantage is particularly pronounced for the larger mesh, which benefits from GPU parallelism as the increased workload provides more opportunities for parallel execution and better utilization of the compute units.

Slang+CUDA delivers the highest performance overall. It reaches 251.52× (32-bit) versus 24.65× (64-bit) on the 256k-face mesh, and 71.04× versus

[2] https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/workGroupFunctions.html

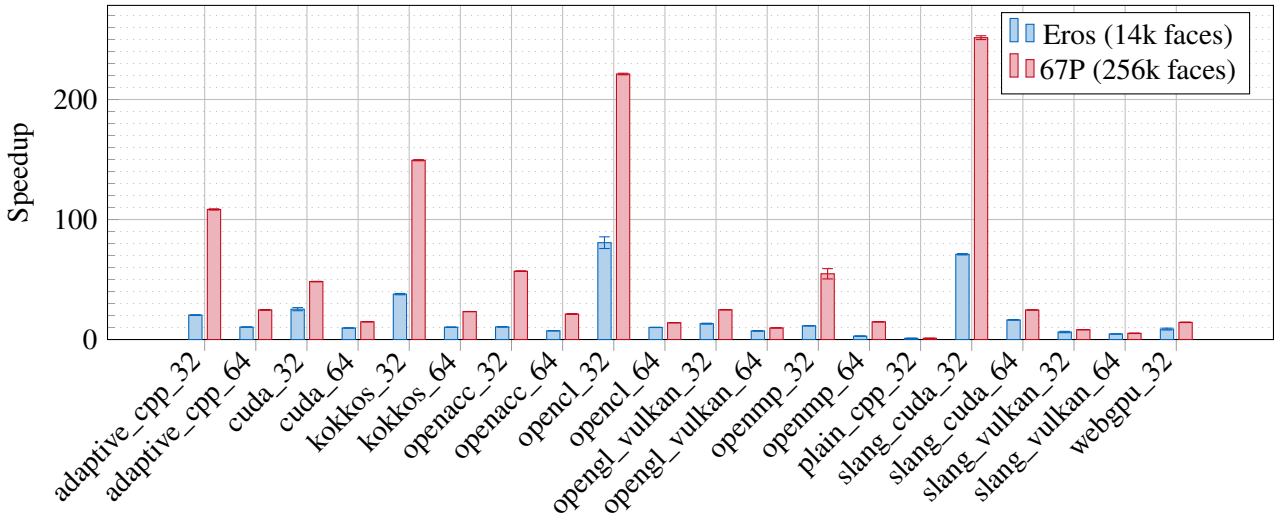


Figure 1 Speedup of gravity calculation for a model of *67P/Churyumov–Gerasimenk* [37, 38] with 256k faces compared to plain_cpp_64 (taking 46.24ms). Executed on a system with an Intel i5-12400 CPU and a NVIDIA GeForce RTX 3070.

16.36× on the 14k-face mesh. Interestingly, the hand-written CUDA implementation performs much worse (48.30× and 25.31× for 32-bit) compared to the one generated by the SLANG compiler.

OpenCL achieves the second-best performance with 221.24× (32-bit) versus 14.00× (64-bit) on the large mesh and the best performance on the small mesh with 80.71× versus 10.09×.

This is followed by Kokkos with 149.41× (32-bit) versus 23.30× (64-bit) on the 256k-face mesh and 37.85× versus 10.27× on the 14k-face mesh. Adaptive CPP shows 108.38× versus 24.66× for the large mesh and 20.38× versus 10.40× for the small mesh.

The pragma based frameworks OpenMP and OpenACC show more moderate results. OpenMP attains 54.79× (32-bit) versus 14.78× (64-bit) on the large mesh and 11.45× versus 2.85× on the small mesh. OpenACC reaches 56.97× versus 21.26× for the large mesh and 10.54× versus 7.31× for the small mesh.

Graphics APIs exhibit smaller performance differences between precision modes. OpenGL/Vulkan achieves 13.23× (32-bit) versus 7.17× (64-bit) on the large mesh. WebGPU shows 14.32× versus 8.73× for the 256k-face mesh and 8.73× versus 6.42× for the 14k-face mesh. Slang-Vulkan attains 8.20× versus 5.12× on the large mesh and 6.24× versus 4.64× on the small mesh. These reduced differences may occur because graphics APIs use 32-bit mathematical functions like log and tan even in 64-bit implementations.

We also ran the same benchmarks on a server with an AMD EPYC 7402 (24 cores, 2.8 GHz) and an NVIDA RTX 3080. The lower single core performance roughly doubles the execution time of the baseline implementation, for example 89.17 ms instead of 46.24 ms for the asteroid 67P. The implementations are also slightly faster due to the stronger GPU, which more than doubles the speedup factor on this server (e.g. 544 instead of 221 for OpenCL32).

6 Numerical Precision

While the benchmark results show that single-precision floating-point arithmetic is much faster than double precision, speed alone is not enough if the final result becomes too inaccurate. Numerical precision is crucial, especially in simulations that integrate results over time. In such cases, even small errors can accumulate and grow significantly, leading to large deviations from the correct solution.

Another factor, besides the choice of data type, can be the order in which operations, such as additions, are performed. Because the addition of floating point numbers is not associative, a different order can lead to different results.

6.1 Potential

Figure 2 shows how accurate the results of the different implementations are for the potential based on the normalised distance. For the calculation, the gravitational potential is evaluated for 642 points on a sphere with the given radius around the asteroid, Eros in this case. The shown result is the average of the absolute error compared to the high-precision baseline.

As expected, the size of the error depends mainly on the data types used. The versions with 32 bit floats

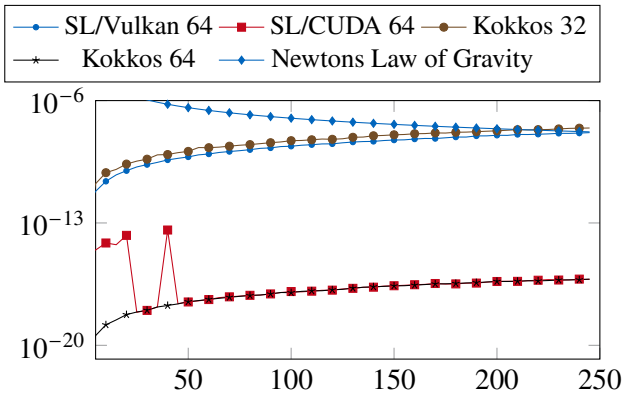


Figure 2 Average absolute error in normalized coordinates compared the reference implementation (C++ with 128 bit floats and improved Kahan–Babuška summation) for points on a sphere with increasing radius. Using a model of the asteroid Eros [39] with 15k faces. To improve readability not all implementations are shown, the 32 bit versions are all in the same error range as Kokkos 32 and the 64 bit versions in the same as Kokkos 64. The error of GLSL/Vulkan is similar to that of SL/Vulkan64.

all have similarly large errors, which is why only the Kokkos implementation is shown in Figure 2.

This is closely followed by the implementations that use 64 bits float but also graphic apis or Vulkan (GLSL/Vulkan and SLANG/Vulkan). Since mathematical functions such as sin or log are only defined for 16 or 32 bits in these frameworks, the 64 bit values have to be converted to a lower precision beforehand. This is why these implementations are only marginally better in terms of accuracy.

The situation is different with the remaining 64 bit implementations, which are significantly better. Because all implementations are in roughly the same range, only Kokkos is shown in Figure 2 to avoid clutter. The only exception here is the Slang implementation using CUDA as backend (SL/CUDA 64), which has a higher error at closer distances but at greater distances it is the same as the others

The graph also shows the error in the calculation of gravity using Newton's law of universal gravitation:

$$F = G\frac{m_1 m_2}{r^2}$$

.

For points that are very close to the asteroid, this equation has a substantial error, as it assumes the asteroid to be a point mass. As the distance increases this error decreases and at a distance of around 250 it is even more accurate than the 32 Bit implementations.

6.2 Gravity Simulation

To determine how accurate the computed acceleration values are, we have implemented a very simple gravity simulation. This computes the motion of an object in orbit around the body over time using a simple euler integration. Figure 3 shows the cartesian distance to the high-precision baseline during such a simulation around the asteroid Eros, for a total of 100000 time steps (with $dt = 10$). For the calculations we use normalised coordinates, where one unit of length corresponds to approximately 20413.87 metres.

The distance rises and falls periodically in all implementations because the errors move the simulated object into slightly different orbits around the asteroid. However, the maximum deviation increases steadily over time, although it is limited to the diameter of the orbit because the escape velocity is never reached.

As expected, the level of deviation is significantly greater in the implementations with single precision floats compared to the variants using double precision.

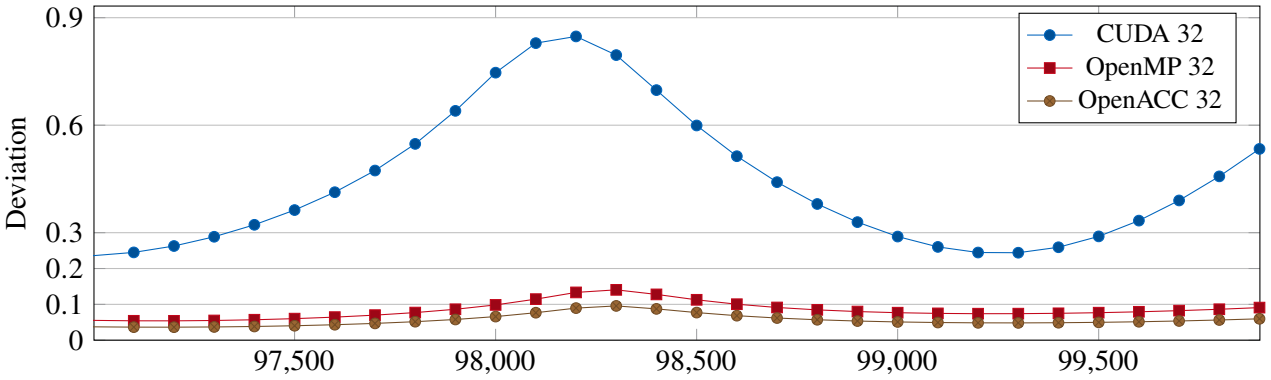
The native CUDA implementation with 32 bit floats shows by far the largest deviation. After 100k timesteps, the deviation at the point of greatest separation is 0.848, which is roughly 17 311 m and thus longer than mean diameter of Eros (16.84 km). At this point, the object in the simulation with CUDA would therefore be on the exact opposite side compared to the high-precision variant.

This is followed at a significant distance by three other variants of compute frameworks using single precision floats: OpenMP (0.14 / 2858 m), OpenACC (0.1 / 2041 m) and Adaptive Cpp (0.08 / 1633 m).

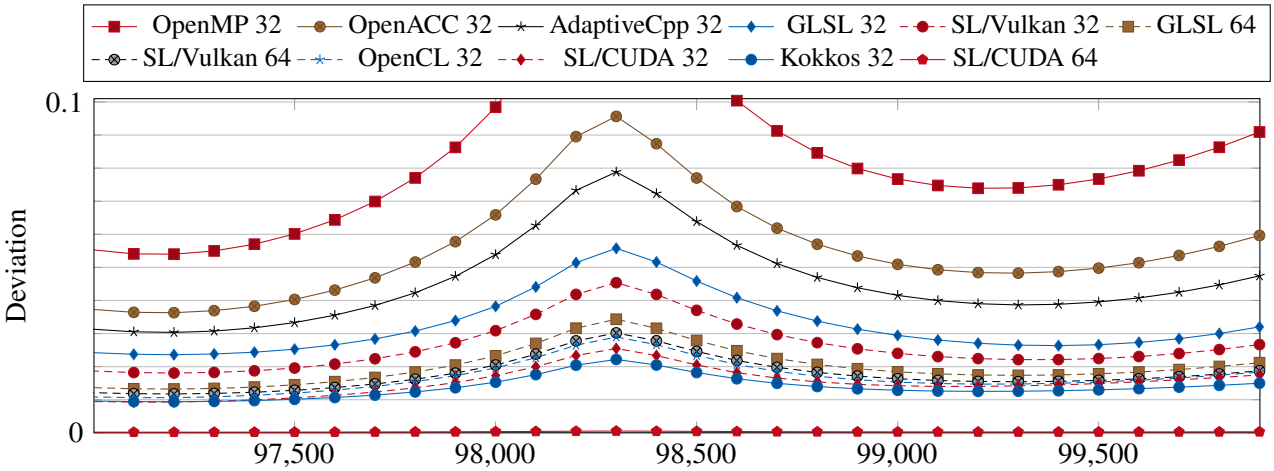
Next are all the implementations that use the Vulkan graphic API. The deviations for the two 32 bit variants of GLSL (0.056 / 1143 m) and SLANG with Vulkan backend (0.045 / 919 m) are about 50% larger compared to the respective 64 bit variants (GLSL: 0.034, SLANG: 0.030). However, for both precisions the SLANG implementation is slightly better than the respective GLSL variant.

A little better are the two winners of the benchmarks, OpenCL 32 (0.029 / 592 m) and SLANG with CUDA as backend (0.0255 / 521 m). The Kokkos implementation is the best of the 32 bit variants with a maximum deviation of 0.022 (449 m) and even beats the baseline implementation (0.0246) using single precision floats.

Out of the remaining 64 bit variants, only the SLANG variant with CUDA backend has a significant deviation of 0.0005, which corresponds to roughly 10.21 m. All other variants have a maximum deviation



(a) Comparison of the bottom three implementations in terms of distance to the reference implementation during the last three thousand time steps of the gravity simulation.



(b) Comparison of the remaining implementation with non-negligible errors.

Figure 3 Distance (in normalised coordinates) to the reference implementation during a gravity simulation using euler integration with 100000 timestamps and $dt = 10$. Using a model of the asteroid Eros [39] with 15k faces. Implementations not covered here (OpenMP 64, AdaptiveCpp 64, CUDA 64, OpenACC64 and OpenCL64) only have a marginal error between 1.75×10^{-10} and 1.6×10^{-11}.

between 1.75×10^{-10} and 1.6×10^{-11} which is likely negligible (smaller than 4 µm).

7 Code Complexity

Table 1 compares the complexity of our implementations, based on their lines of code (LoC). While it does not capture all aspects of complexity it still reflects the effort required to write, maintain, and understand the code. A higher LoC count can indicates more a more verbose setup, missing high-abstractions, or the need of explicit resource management.

The baseline implementation, written in C++, consists of 245 LoC and serves as a reference point for evaluating the relative complexity.

Kokkos, with 254 LoC, is only slightly more complex than the baseline. The additional code consists mainly of the explicit management of host and device memory using the `Kokkos::View`. When porting existing C++ code, it is also important to note that Kokkos uses round brackets to access array elements, as opposed to square brackets in C++.

Similarly, the pragma-based OpenMP (272 LoC) and OpenAcc (277 LoC) show a modest increases in complexity. Since our program keeps memory permanently on the GPU, we have to manage the memory buffers manually. For a program that transfers the data to the GPU for each calculation, only one additional line per loop would usually be necessary.

When comparing the two, it should also be noted that OpenMP allows reduction clauses with custom datatypes on the GPU if this has been declared using `omp declare reduction`. OpenACC, on the other hand, needs an additional loop to add up all members of the result struct individually, which is why the implementation is a few lines longer.

Table 1 Lines of code (without blank lines and comments) for our implementations and the involved languages.

Implementation	Language	Lines of Code	
Baseline	C++		243
Kokkos	C++		254
OpenMP	C++		272
OpenACC	C++		277
AdaptiveCpp	C++		310
CUDA	CUDA C++		426
OpenCL	OpenCL C	290	
	C++	+180	= 470
Slang / CUDA	Slang	315	
	CUDA	+195	= 510
WebGPU	WGSL	276	
	C++	+272	= 548
Slang / Vulkan	Slang	315	
	C++	+310	= 625
Vulkan	C++	456	
	GLSL	+ 337	= 793

AdaptiveCpp (310 LoC) needs not only the buffers for device memory but also accessors for them, which roughly doubles the number of additional lines needed compared to the prama-based frameworks (which only need to declare the buffers).

The CUDA implementation needs a lot of additional code, resulting in a total of 426 lines of code. Again, on reason is the explicit memory management and the calculation of block sizes and grid dimensions for launching the kernel. In addition, further data types and functions are needed to use the cuda native data types for small vectors (e.g. float3, float4) instead of the data types used in the baseline.

What is not included in this line count are common mathematical operations for these datatypes, even if they are not part of the CUDA library and have to be taken from the CUDA samples[3] code instead.

The implementations that combine multiple languages tend to be significantly more complex. For example, OpenCL, which uses both OpenCL C and C++, totals 470 LoC, nearly double the baseline. The build system is also more complex for such frameworks, as either a separate shader compiler must be called or the shaders must be embedded as strings in the compiled program.

Similarly, WebGPU, which combines WGSL with C++, totals 548 LoC. In contrast to OpenCl, WebGPU only provides a C API, so all resources in the code must either be cleaned up manually (e.g. by calling function like `wgpuDeviceRelease`) or have to be wrapped in classes which do this in the destructor.

It is also worth mentioning that the number of lines of code in the WebGPU implementation does not reflect the required porting effort quite as well. This is because WGSL is syntactically not a C like language, as is the case with all the others, which makes it more difficult to translate the original C++ code.

The Slang-based implementations also exhibit higher complexity. Slang paired with CUDA results in 510 LoC, so there is a bit of overhead compared to the native CUDA implementation. However Slang combined with Vulkan reaches 625 LoC.

Vulkan, combining C++ and GLSL, reaches 793 LoC, making it the most complex implementation. Vulkan's low-level API requires a detailed setup of pipelines, shaders, and device buffers.

8 Related Work

While CUDA has long dominated GPU programming, modern supercomputers increasingly rely on AMD and Intel accelerators, which highlights the importance of performance portability [40].

Davis et al. [2] provide a comparison of seven poppular parallel programming models (CUDA, HIP, Kokkos, RAJA, OpenMP, OpenACC, and SYCL) for GPU computing on NVIDIA and AMD server-grade hardware. Five scientific applications were used to test performance across these different programming approaches. Their results show CUDA achieves the best performance on NVIDIA systems, but Kokkos, RAJA and SYCL can sometimes match or even beat this performance. The Pragma based frameworks OpenMP and OpenACC, however, are usually slower than the native frameworks. [2]

But other papers, like the one from Bartolomeu et al. [40], demonstrate that OpenMP and OpenACC can outperform CUDA, at least on some GPUs/

The paper by Breyer et al. [41] compares the performance of CUDA, OpenCL, OpenMP and SYCL (hypSYCL and DPC++) on different consumer-level platforms including GPUs from NVIDIA, AMD, and Intel. The evaluation uses a parallel implementation of a Least Squares Support Vector Machine (LS-

[3] https://github.com/NVIDIA/cuda-samples

SVM) as a test case, focusing on the contained matrix-vector multiplication. Their results show that no single framework works best on all hardware: CUDA achieves the best performance on NVIDIA GPUs, while OpenCL performs best on AMD and Intel GPUs. SYCL implementations have a good portability but cannot fully match the performance of vendor-specific solutions. However, none of the frameworks supported all of the platforms used due to compilation errors or missing runtime libraries.

The benchmarks done by Silva et al. [42] also show that programs using SYCL are slower compared to OpenCL or OpenMP. However, they also show that it is easier to program using the SYCL APIs because it requires fewer lines of code and fewer API calls.

There are only a very few papers that compare the performance of compute frameworks like CUDA or OpenMP with that of graphics APIs like Vulkan.

Mazaheri et al. [43] show that their Vulkan implementation of a convolutional neural network performed better then their CUDA and OpenCL backends.

Lu [44] developed a Vulkan backend for Accelerate, a Haskell DSL for high-performance array computations. They show that Vulkan can achieve competitive performance compared to their PTX backend in some benchmarks. They also mention the lower precision of the Vulkan implementation, the lack of support for double precision math functions and the high code verbosity of the Vulkan API[44].

9 Discussion and Summary

In this paper we have shown that performance portability frameworks can greatly accelerate applications by utilising the GPU, in some cases even with minimal changes to the source code.

The latter is particularly the case with Kokkos, which was able to increase performance by up to 150 times with just a handful of additional lines. Thus, a straightforward port of C++ code to CUDA is not necessarily worthwhile, given that the native CUDA implementation cannot keep up with Kokkos either in terms of accuracy or performance while also requiring more implementation effort.

In our benchmarks, SLANG was able to achieve the greatest increase in performance, although it requires a little more effort to implement. Another small disadvantage is that even the variant with 64bit floats had significant errors in the gravity simulation, even if those were rather small in comparison.

Another framework to be recommended is OpenCL, which also delivers strong performance with a fairly low overhead. The big advantage of OpenCL is that the code running on the GPU is only compiled at runtime by the vendor specific driver, which makes it easier to distribute. Frameworks with CUDA as backend are either bound to certain CUDA versions or have to recompile the whole code. Especially non-technical end users rarely have the necessary complete compilation toolchain including a working CUDA SDK.

The implementations that use graphic APIs all have the disadvantage that the mathematical functions in the standard library are not available for double precision, which can lead to significant errors in the calculation. Therefore, their use can only be recommended if single precision floats are sufficient in terms of accuracy.

In our tests on consumer level GPUs (RTX 3070, 3080), we saw that high speedup factors are only possible with single precision floating points, which can however lead to significant deviations in the results. Whether this is an acceptable trade-off must be decided on a case-by-case basis. NVIDIAs HPC GPUs have significantly more units for 64-bit floating points, which is why the performance difference should be much smaller when using them.

10 Future Work

Our tests have clearly shown that frameworks such as Kokkos or OpenCL can achieve very high performance on some Nvidia GPU from their GeForce RTX 30 series. For future work on performance portability, however, it would be interesting to see how this would look like on other NVIDIA models and on graphics cards from other manufacturers, e.g. AMD or Intel. Tests on MacOS would also be interesting, even if not all frameworks support it natively.

For a comprehensive comparison of the frameworks, other programs would also have to be considered. For example, programs that are limited by memory bandwidth would be interesting to see how much control the respective frameworks offer over memory access. Additionally, one could also look at problems that may benefit from special hardware units such as tensor or raytracing cores to see which frameworks allow access to such special features.

It would also be interesting to see whether a better trade-off between performance and accuracy can be achieved with a mixed use of floating point precisions.

11 Transparency Note on AI/LLM Usage

The following AI/LLM based tools where used during the development of this project and the writing process of this paper:

- The locally running **CLion Inline Completion** was used throughout the code development to automatically complete lines of code.
- **DeepL** was used for translation during the writing process.
- **ChatGPT** and **deepseek** where used to help with writing data visualization code, some of the boilerplate code required for WebGPU and the LaTeXcode for diagrams and tables in this paper.

References

[1] S.J. Pennycook, J.D. Sewall, and V.W. Lee. "Implications of a metric for performance portability". In: *Future Generation Computer Systems* 92 (2019), pp. 947–958. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17300559.

[2] Joshua Hoke Davis et al. *An Evaluative Comparison of Performance Portability across GPU Programming Models*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2024.

[3] The Khronos Vulkan Working Group. *Vulkan 1.4 Specification*. Khronos Group. Apr. 2025.

[4] Sai Bangaru et al. "SLANG.D: Fast, Modular and Differentiable Shader Programming". In: *ACM Transactions on Graphics (SIGGRAPH Asia)* 42.6 (Dec. 2023), pp. 1–28. DOI: 10.1145/3618353.

[5] Jonas Schuhmacher. *Efficient Polyhedral Gravity Modeling in Modern C++*. Tech. rep. Technical University of Munich, Dec. 2022. URL: https://mediatum.ub.tum.de/doc/1695208/1695208.pdf.

[6] Jonas Schuhmacher et al. "Efficient Polyhedral Gravity Modeling in Modern C++ and Python". In: *Journal of Open Source Software* 9.98 (June 2024), p. 6384. DOI: 10.21105/joss.06384. URL: https://joss.theoj.org/papers/10.21105/joss.06384.

[7] S Petrović. "Determination of the potential of homogeneous polyhedral bodies using line integrals". In: *Journal of Geodesy* 71.1 (1996), pp. 44–52. DOI: 10.1007/s001900050074.

[8] Dimitrios Tsoulis and Sveto Petrović. "On the singularities of the gravity field of a homogeneous polyhedral body". In: *Geophysics* 66.2 (2001), pp. 535–539. DOI: 10.1190/1.1444944.

[9] Dimitrios Tsoulis. "Analytical computation of the full gravity tensor of a homogeneous arbitrarily shaped polyhedral source using line integrals". In: *Geophysics* 77.2 (2012), F1–F11. DOI: 10.1190/geo2010-0334.1.

[10] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to learning OpenGL, version 4.5*. Addison-Wesley Longman Publishing Co., Inc., 2016.

[11] Ian Buck. "GPU computing with NVIDIA CUDA". In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, 6–es. ISBN: 9781450318235. DOI: 10.1145/1281500.1281647. URL: https://doi.org/10.1145/1281500.1281647.

[12] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling many-core performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (July 2014). ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003. URL: https://www.osti.gov/biblio/1106586.

[13] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[14] Aksel Alpay and Vincent Heuveline. "One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends". In: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585351. URL: https://doi.org/10.1145/3585341.3585351.

[15] *OpenMP Application Programming Interface. Version 5.2 November 2021*. OpenMP Architecture Review Board, Nov. 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[16] Ronan Keryell, Ruyman Reyes, and Lee Howes. "Khronos SYCL for OpenCL: a tutorial". In: *Proceedings of the 3rd International Workshop on OpenCL*. IWOCL '15. Palo Alto, California: Association for Computing Machinery, 2015. ISBN: 9781450334846. DOI: 10.1145/2791321.2791345. URL: https://doi.org/10.1145/2791321.2791345.

[17] The Khronos SYCL Working Group. *SYCL 2020 Specification*. Khronos Group. Apr. 2025.

[18] Aksel Alpay and Vincent Heuveline. "AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler". In: *Proceedings of the 12th International Workshop on OpenCL and SYCL*. IWOCL '24: International Workshop on OpenCL and SYCL. Chicago IL USA: ACM, Apr. 8, 2024, pp. 1–12. ISBN: 9798400717901. DOI: 10.1145/3648115.3648117. URL: https://dl.acm.org/doi/10.1145/3648115.3648117 (visited on 04/13/2025).

[19] CCCL Development Team. *CCCL: CUDA C++ Core Libraries*. 2023. URL: https://github.com/NVIDIA/cccl.

[20] Chris Lattner. *Modular: Democratizing AI Compute, Part 3: How did CUDA succeed?* Feb. 12, 2025. URL: https://www.modular.com/blog/democratizing-ai-compute-part-3-how-did-cuda-succeed (visited on 04/13/2025).

[21] OpenACC-Standard.org. *The OpenACC Application Programming Interface Version*. Nov. 2022. URL: OpenACC-Standard.org.

[22] NVIDIA Corporation. *OpenACC Getting Started Guide*. Jan. 2025.

[23] *OpenACC - GCC Wiki*. URL: https://gcc.gnu.org/wiki/OpenACC (visited on 04/13/2025).

[24] J. E. Stone, D. Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science and Engg.* 12.3 (May 1, 2010), pp. 66–73. ISSN: 1521-9615.

[25] The Khronos OpenCL Working Group. *The OpenCL Specification*. Khronos Group. Apr. 2025.

[26] The Khronos Vulkan Working Group. *Untitled :: Vulkan Documentation Project*. Khronos Group. URL: https://docs.vulkan.org/guide/latest/index.html (visited on 04/13/2025).

[27] The Khronos SPIR Working Group. *SPIR-V Specification*. Khronos Group. Jan. 2025.

[28] Tom Deakin and Timothy G. Mattson. *Programming your GPU with OpenMP: performance portability for GPUs*. Scientific and engineering computation. Cambridge, Massachusetts: The MIT Press, 2023. ISBN: 978-0-262-54753-6.

[29] Michael Klemm. *Intro to GPU Programming with the OpenMP API*. Oct. 20, 2021. URL: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf (visited on 04/13/2025).

[30] *Khronos Group Launches Slang Initiative, Hosting Open Source Compiler Contributed by NVIDIA*. The Khronos Group. Section: News. Nov. 21, 2024. URL: https://www.khronos.org/news/press/khronos-group-launches-slang-initiative-hosting-open-source-compiler-contributed-by-nvidia (visited on 04/13/2025).

[31] *Slang User's Guide*. slang. URL: http://shader-slang.org/slang/user-guide/ (visited on 04/13/2025).

[32] W3C GPU for the Web Community Group. *WebGPU*. Working Draft. W3C, Mar. 2025. URL: https://www.w3.org/TR/2025/CRD-webgpu-20250321/ (visited on 04/13/2025).

[33] W3C GPU for the Web Community Group. *WebGPU Shading Language*. Working Draft. W3C, Apr. 2025. URL: https://www.w3.org/TR/2025/CRD-WGSL-20250410/ (visited on 04/13/2025).

[34] The Dawn and Tint Authors. *Dawn, a WebGPU implementation*. URL: https://dawn.googlesource.com/dawn.

[35] The gfx-rs developers. *wgpu*. URL: https://github.com/gfx-rs/wgpu.

[36] A. Neumaier. "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen". In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 54.1 (1974), pp. 39–51. ISSN: 1521-4001. DOI: 10.1002/zamm.

19740540106. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19740540106 (visited on 04/13/2025).

[37] Greg Frieger. *Comet 67P/churyumov-gerasimenko*. July 2021. URL: https://3d-asteroids.space/comets/67P-Churyumov-Gerasimenko (visited on 04/20/2025).

[38] Rosetta Flight Dynamics Team. *Comet 67P/Churyumov-Gerasimenko Shape Models EAICD - Annex A: ESA NAVCAM Shape Model*. 2015. URL: https://pds-smallbodies.astro.umd.edu/holdings/ro-c-multi-5-67p-shape-v1.0/document/rmoc_eaicd.pdf.

[39] Robert Gaskell. *Gaskell Eros Shape Model Bundle V1.0*. In collab. with PDS Small Bodies Node. 2020. DOI: 10.26033/0HPF-4E64. URL: https://sbn.psi.edu/pds/resource/doi/erosshape_1.0.html (visited on 04/23/2025).

[40] Rodrigo A.C. Bartolomeu et al. "Effect of implementations of the N-body problem on the performance and portability across GPU vendors". In: *Future Generation Computer Systems* 169 (2025), p. 107802. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2025.107802. URL: https://www.sciencedirect.com/science/article/pii/S0167739X25000974.

[41] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware". In: *Proceedings of the 10th International Workshop on OpenCL*. IWOCL '22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3529980. URL: https://doi.org/10.1145/3529538.3529980.

[42] Hércules Cardoso da Silva, Flávia Pisani, and Edson Borin. "A Comparative Study of SYCL, OpenCL, and OpenMP". In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 2016, pp. 61–66. DOI: 10.1109/SBAC-PADW.2016.19.

[43] Arya Mazaheri et al. "Enhancing the Programmability and Performance Portability of GPU Tensor Operations". In: *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings*. Göttingen, Germany: Springer-Verlag, 2019, pp. 213–226. ISBN: 978-3-030-29399-4. DOI: 10.1007/978-3-030-29400-7_16. URL: https://doi.org/10.1007/978-3-030-29400-7_16.

[44] Xinliang Lu. "A Vulkan Backend for Accelerate". Master Thesis. 2024. URL: https://studenttheses.uu.nl/handle/20.500.12932/47433.