# A Comparison of OpenCL, CUDA, and HIP as Compilation Targets for a Functional Array Language

Troels Henriksen
athas@sigkill.dk
University of Copenhagen
Copenhagen, Denmark

## Abstract

This paper compares OpenCL, CUDA, and HIP as compilation targets for Futhark, a functional array language. We compare the performance of OpenCL versus CUDA, and OpenCL versus HIP, on the code generated by the Futhark compiler on a collection of 48 application benchmarks on two different GPUs. Despite the generated code in most cases being equivalent, we observe significant performance differences on the same hardware, ranging from 0.42x to 1.72x in the most extreme cases. We identify the root causes of most of these differences, many of which are due to relatively superficial details such as inconsistent defaults regarding compiler optimisation and numerical accuracy, although a few remain mysterious.

***CCS Concepts:*** • **Computing methodologies** → **Parallel programming languages**; Massively parallel and high-performance simulations; • **General and reference** → *Performance*.

***Keywords:*** GPU, functional programming, parallel programming, performance measurement

**ACM Reference Format:**
Troels Henriksen. 2024. A Comparison of OpenCL, CUDA, and HIP as Compilation Targets for a Functional Array Language. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Programming for Productivity and Performance (FProPer '24), September 6, 2024, Milan, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3677997.3678226

## 1 Introduction

Graphics Processing Units (***GPU***s) are massively parallel processors that are commonly used for many non-graphics workloads, including AI and scientific computing. As perhaps the most affordable and easily available massively parallel computers, while being notoriously tedious to program directly, they are an enticing compilation target for high level languages.

In contrast to CPUs, GPUs are typically not programmed by directly generating and loading machine code. Instead, the programmer must use fairly complicated software APIs to compile the GPU code and communicate with the GPU hardware. Various GPU APIs mostly targeting graphics programming exist, including OpenGL [29], DirectX, and Vulkan [29]. However, in this paper we are concerned with APIs for general-purpose non-graphics GPU programming (***GPGPU***): specifically, CUDA, OpenCL, and HIP.

CUDA was published by NVIDIA in 2007 as a proprietary API and library for NVIDIA GPUs. It has since become the most popular API for GPGPU , largely aided by the single-source CUDA C++ programming model provided by the `nvcc` compiler. In response, OpenCL was published in 2009 by Khronos as an open standard for heterogeneous computing [19]. In particular, OpenCL was adopted by AMD and Intel as the main way to perform GPGPU on their GPUs, and is also supported by NVIDIA. For reasons that are outside the scope of this paper, OpenCL has so far failed to reach the popularity of CUDA.

The dominance of CUDA posed a market problem for AMD, since software written in CUDA can only be executed on an NVIDIA GPU. Since 2016, AMD has been developing HIP, an API that is largely identical to CUDA, and which includes tools for automatically translating CUDA programs to HIP (`hipify`). Since HIP is so similar to CUDA, an implementation of the HIP API in terms of CUDA is straightforward, and is also supplied by AMD. The consequence is that a HIP application can also be run on both AMD and NVIDIA hardware, often without any performance overhead [18], although we do not investigate that in this paper.

While HIP is clearly intended as a strategic response to the large amount of existing CUDA software, HIP can also be used by newly written code. The potential advantage is that HIP (and CUDA) exposes more GPU features than OpenCL, as OpenCL is a more slow-moving and hardware-agnostic specification developed by a committee, which cannot be extended unilaterally by GPU manufacturers.


*FProPer '24, September 6, 2024, Milan, Italy*

ACM ISBN 979-8-4007-1100-8/24/09
https://doi.org/10.1145/3677997.3678226

Futhark [13] is a functional array language that supports a discipline of data-parallel programming through a vocabulary of ***second order array combinators*** (SOACs), resembling common functional programming constructs such as `map`, `reduce`, and `scan`. Through a language-based cost model in the style of NESL, Futhark provides asymptotic guarantees about parallel execution, allowing the programmer to reason about the parallelism of their algorithms without taking hardware details into account.

The Futhark compiler is an aggressively optimising ahead-of-time compiler. Although the Futhark language itself is hardware-agnostic, and the compiler supports various compilation targets, the most mature backends are the ones that target GPUs. Three different GPU backends, targeting different GPU APIs, are supported by the compiler: OpenCL, HIP, and CUDA. These backends share the same compilation pipeline (including all optimisations), and differ only in the final code generation step.

This paper investigates the performance and convenience of these three backends. The theme of our contribution is to relay our experiences to other implementers of languages or libraries that wish to target one or more of OpenCL, HIP, and CUDA. In particular, we wish to identify subtle performance or correctness pitfalls. We do this by:

1. A qualitiative explanation of how the compiler backends have been structured and implemented (section 2).
2. A quantitative evaluation of the performance of the three backends on a sizeable collection of benchmark programs (section 3), including an analysis of the root source of performance differences.

Due to the number of benchmarks and the sheer complexity of the GPU software stacks and their somewhat inscrutable optimising compilers, we cannot yet explain *all* observed performance differences, but we identify several root causes that affect multiple benchmarks.

### 1.1 Nomenclature

We use of CUDA terms rather than OpenCL terms, even when discussing non-NVIDIA hardware, as the CUDA nomenclature is more commonly known in the GPGPU community.

Parts of this paper will assume basic knowledge of GPU architecture, their programming model, memory hierarchy, and performance characteristics. The first couple of chapters of the *CUDA C++ Programming Guide* [25] are a good introduction.

## 2 Compilation model

The Futhark compiler supports three GPU backends: OpenCL, HIP, and CUDA. All three backends use exactly the same compilation pipeline, including all optimisations, except for the final code generation stage. The result of compilation is conceptually two parts: a ***GPU program*** that contains definitions of GPU functions (***kernels***) that will ultimately run on the GPU and a ***host program***, in C, that runs on the ***host*** (in practice, the CPU) and contains invocations of the chosen GPU API. As a purely practical matter, the GPU program is also embedded in the host program as a string literal. At runtime, the host program will pass the GPU program to the ***kernel compiler*** provided by the GPU driver, which will generate machine code for the GPU in use.

The OpenCL backend was the first to be implemented, starting in around 2015 and becoming operational in 2016. The CUDA backend was implemented by Jakob Stokholm Bertelsen in 2019 [2], largely in imitation of the OpenCL backend, motivated by the somewhat lacking enthusiasm for OpenCL demonstrated by NVIDIA. For similar reasons, the HIP backend was implemented in 2023. While the OpenCL backend might be more mature purely due to age, the backends make use of the same optimisation pipeline and, as we shall see, almost the same code generator, and so produce code of near identical quality.

The difference between the code generated by the three GPU backends is (almost) exclusively down to which GPU API is invoked at runtime. To keep the backends easy to maintain, we have designed a simple abstraction layer, shown in fig. 1, that exposes the few operations that are required by Futhark. The operations can be grouped into two themes:

1. A way to allocate and copy memory.
2. A way to retrieve handles to GPU kernels from the compiled GPU program, and then execute them on some provided array of arguments.

This interface has been defined to be straightforward to implement with all of OpenCL, HIP, and CUDA. For HIP and CUDA, we use the so-called "driver API" layer, which is somewhat more low-level than what most CUDA programmers are used to and which is similar to OpenCL in terms of abstraction level and verbosity. The abstraction layer does not expose any features that are not universally available (which are therefore not usable by the Futhark code generator).

There is no significant difference between the backends regarding how difficult this portability layer is to implement. CUDA requires 231 lines of code, HIP 233, and OpenCL 255. This excludes platform-specific startup and configuration logic, but even the most verbose backend (OpenCL), consists of a total of 1124 lines of backend-specific host code.

### 2.1 GPU kernel code

As far as Futhark is concerned, the differences between the languages used for expressing the GPU program itself (CUDA C, OpenCL C, and HIP C) are minor, almost down to mere syntax, and do not merit much elaboration. This is partially because Futhark does not make use of any language-level abstraction features and merely uses the human-readable syntax as a form of portable assembly code. In particular, all these backends have complete support for integer and floating-point types of all the sizes one might expect, as

```
// Types that must be defined by the backend implementation:
struct futhark_context { ... };
typedef ... gpu_mem;
typedef ... gpu_kernel;
// Memory management:
int gpu_free_actual(struct futhark_context *ctx, gpu_mem mem);
int gpu_alloc_actual(struct futhark_context *ctx, size_t size, gpu_mem *mem_out);
int gpu_memcpy(struct futhark_context* ctx,
               gpu_mem dst, int64_t dst_offset,
               gpu_mem src, int64_t src_offset,
               int64_t nbytes);
int gpu_scalar_from_device(struct futhark_context* ctx,
                           void *dst,
                           gpu_mem src, size_t offset, size_t size);
int gpu_scalar_to_device(struct futhark_context* ctx,
                         gpu_mem dst, size_t offset, size_t size,
                         void *src);
// Kernel invocation:
void gpu_create_kernel(struct futhark_context *ctx,
                       gpu_kernel* kernel,
                       const char* name);
int gpu_launch_kernel(struct futhark_context* ctx,
                      gpu_kernel kernel, const char *name,
                      const int32_t grid[3],
                      const int32_t block[3],
                      unsigned int shared_mem_bytes,
                      int num_args,
                      void* args[num_args],
                      size_t args_sizes[num_args]);
```

**Figure 1.** C declarations for the host-side GPU abstraction layer used by the Futhark compiler. The compiler generates code targeting this interface. An implementation of the interface in terms of OpenCL, HIP, or CUDA is prepended to the generated program. A few bookkeeping and initialisation functions have been elided, as well as functions for copying entire arrays between host and GPU.

well as support for pointers and pointer arithmetic. This is unfortunately not the case for all GPU languages, with GLSL (OpenGL) or WGSL (WebGPU) being much more constrained.

Futhark makes use of hardware features such as barriers, memory fences, atomics, and shared memory, which exist in conceptually identical forms in all of the languages, merely with different names and syntax. These differences are easily handled by prefixing a small set of C macro and function definitions that provides a uniform interface.

### 2.2 Runtime compilation

The choice to perform runtime compilation of the GPU program is perhaps a little unusual, as in particular CUDA programmers are used to the single-source model of CUDA C++, where `nvcc` takes care of separating GPU code from host code, and compiles the GPU code at host compile-time.

Our choice of runtime compilation is largely a historical accident due to the OpenCL backend being the first one implemented, since OpenCL provides an API based on runtime compilation. While the OpenCL approach is certainly less ergonomic than CUDA for a human programmer, runtime compilation provides a significant advantage to a compiler, as it allows important constants such as thread block sizes, tile sizes, and other tuning parameters to be set dynamically (from the user's perspective) rather than statically, while still allowing such sizes to be visible as sizes to the kernel compiler. This enables important optimisations such as unrolling of loops over tiles. Essentially, this approach

provides a primitive but very convenient form of Just-In-Time compilation. Runtime compilation is available in CUDA through the NVRTC library and in HIP through the equivalent HIPRTC library.

Runtime compilation of GPU kernels adds significant process start-up overhead, ranging from several seconds to more than a minute for large programs. We have observed no significant difference in runtime compilation time between the different APIs. All of OpenCL, CUDA, and HIP provide effective mechanisms for retrieving the compiled form of the GPU program, which can be stored in a disk cache and loaded during the next startup, which significantly reduces the start-up time. Care must of course be taken to avoid loading an outdated compiled program from this cache. In this paper we do not measure startup overhead, and so will not further discuss this issue.

### 2.3 Parallel constructs

Futhark supports (regular) nested parallelism at the language level, but GPUs only efficiently support two levels of parallelism — the grid level and the block level, and the latter is rather restricted.[1] In practice, we consider GPUs to only support flat parallelism. To map application-level nested parallelism to flat GPU parallelism, Futhark uses a program transformation called *incremental flattening* [14], the precise details of which are outside the scope of this paper. After flattening, GPU computations are expressed in terms of a few core primitives: maps, scans, reduces, and generalised histograms [12]. These are significantly extended compared to their common incarnation in the literature or in the Futhark source language, and support arbitrary input transformations (i.e., they can represent *fused* operations), additional outputs, *scatter* outputs, block-level vs. thread-level execution, multiple operators, and multiple dimensions (i.e., regular segmented operations). Again, for the purpose of this paper, the precise details of these facilities are largely unimportant and will be brought up when relevant.

The code generator knows how to translate each of these parallel primitives to GPU code. Maps are translated into single GPU kernels, with each iteration of the map handled by a single thread. Reductions are translated using a conventional approach where the arbitrary-sized input is split among a fixed number of threads, based on the capacity of the GPU. For segmented reductions, Futhark uses a multi-versioned technique that adapts to the size of the segments at runtime [21].

Using the CUDA or HIP backends, scans are implemented using the *decoupled lookback* algorithm [23], which requires only a single pass over the input, and will be called a ***single-pass scan*** in this paper. Unfortunately, the single-pass scan requires memory model and progress guarantees that are present in CUDA and HIP, but not in OpenCL. Instead, the OpenCL backend uses a less efficient ***two-pass scan*** that manifests an intermediate array of size proportional to the input array. This is the only case for which there is a significant difference in how the CUDA, HIP, and OpenCL backends generate code for parallel constructs. The two-pass scan is also used whenever the scan operator takes arrays of statically unknown size as arguments, as this prevents a register caching technique that is crucial to the performance of the single-pass scan.

Generalised histograms are compiled to GPU kernels that use atomic operations to update a shared array. Depending on the size of the histogram, this array is kept in global or shared memory. Additionally, to minimise the overhead of conflicts when multiple threads try to update the same bin, Futhark employes a technique based on multi-histogramming and multi-passing [12].

[1] One may define "warp level" parallelism as a third level.

## 3 Application Benchmarks

We consider 48 benchmark programs manually ported to Futhark from Accelerate [3], Rodinia [4], Parboil [30], and PBBS [1]. Some of these are variants of the same problem. For example, there are four different implementations of the Rodinia breadth-first-search benchmark, as well as one for PBBS. The benchmarks are of medium size and comprise about 5000 source lines of code (not counting comments and blank lines). The median program size is about 100 lines of Futhark code.

Most of the benchmarks contain multiple ***workloads*** of varying sizes. Each workload is executed at least ten times, and possibly more in order to establish statistical confidence in the measurements [16]. For each workload, we measure the average observed wall clock runtime. For a given benchmark executed with two different backends on the same GPU, we then report the average speedup across all workloads, as well as the standard deviation of speedups. We do not measure GPU initialisation or other such startup overheads, nor do we measure the cost of copying the initial input data to the GPU or copying the final output data from the GPU. All other communication is included in the wall clock measurement.

The complete list of application benchmarks, as well as their origin, follows.

**Accelerate:** *tunnel, fluid, smoothlife, trace, fft, canny, crystal, pagerank, nbody-bh, kmeans, hashcat, nbody, mandelbrot.*

**PBBS:** *convexHull, histogram, breadthFirstSearch, maximalMatching, minSpanningForest, maximalIndependentSet, merge_sort, ray, radix_sort, quick_sort.*

**Rodinia:** *backprop, cfd, bfs_iter_work_ok, hotspot, pathfinder, bfs_filt_padded_fused, bfs_heuristic, kmeans, bfs_asympt_ok_but_slow, lud, lavaMD, myocyte, nw, nn, particlefilter, srad.*

**Parboil:** *tpacf, histo, sgemm, stencil, mri-q, lbm.*

Due to size constraints, we do not discuss each benchmark in detail, except as needed to explain its performance. We benchmark on two different GPUs: an NVIDIA A100 using the CUDA and OpenCL backends, and an AMD MI100 using the HIP and OpenCL backends. We benchmark without any backend-specific tuning or configuration. As we will see below, one can argue that the Futhark backends should be modified to automatically pass certain non-standard options to the GPU kernel compilers, in order to make the comparison more fair. Further, the results might also look different if an auto-tuner was used to calibrate various tuning parameters, such as thread block sizes.

The speedup of using the OpenCL backend compared to the CUDA backend on A100 can be seen in fig. 2a, and similarly for OpenCL compared with HIP on MI100 on fig. 2b. A number higher than 1 means that OpenCL is faster than CUDA or HIP, respectively. A wide error bar indicates that the performance difference between backends is different for different workloads.

In the interest of reproducibility, the experimental infastructure is publicly available and fully documented.[2]

### 3.1 Causes of performance differences

In an ideal world, we would observe no performance differences between backends. However, as discussed in section 2, Futhark does not use equivalent parallel algorithms in all cases. And even for those benchmarks where we *do* generate equivalent code no matter the backend, we still observe differences. The causes of these differences are many and require manual investigation to uncover, sometimes requiring inspection of generated machine code. In the following we will identify the major sources of performance differences, and list the affected benchmarks.

**3.1.1 Defaults for numerical operations.** OpenCL is significantly faster on some benchmarks, such as *mandelbrot* on MI100, where it outperforms CUDA by 1.71×. The reason for this is that OpenCL by default allows a less numerically precise (but faster) implementation of single-precision division and square roots. This is for backwards compatibility with code written for older GPUs, which did not support correct rounding. An OpenCL build option[3] can be used to force correct rounding of these operations, which matches the default behaviour of CUDA and HIP. This explains the performance discrepancies for the benchmarks *nbody*, *trace*, *ray*, *tunnel*, and *mandelbrot* on both MI100 and A100. On MI100, this makes the OpenCL version of *convexHull* slower than the HIP version (due to scans, discussed below). Similarly, passing `-ffast-math` to HIP on MI100 makes it match OpenCL for *srad*, although it is not clear what precisely it influences in this case. An argument could be made that the Futhark compiler should automatically pass the necessary options to ensure consistent numerical behaviour across all backends.

**3.1.2 Different scan implementations.** As discussed above, Futhark's OpenCL backend uses a less-efficient two-pass scan algorithm, rather than a single-pass scan. For benchmarks that make heavy use of scans, the impact is significant. This affects benchmarks such as *nbody-bh*, all BFS variants, *convexhull*, *maximalIndependentSet*, *maximalMatching*, *radix_sort*, *canny*, and *pagerank*. Interestingly, the *quick_sort* benchmark contains a scan operator with particularly large operands (50 bytes each), which interacts poorly with the register caching done by the single-pass scan implementation. As a result, the OpenCL version of this benchmark is faster on the MI100.

**3.1.3 Smaller thread block sizes.** For unknown reasons, AMD's implementation of OpenCL limits thread blocks to 256 threads. This may be a historical limitation, as older AMD GPUs did not support thread blocks larger than this. However, modern AMD GPUs support up to 1024 threads in a thread block (as does CUDA) and this is fully supported by HIP. This limit means that some code versions generated by incremental flattening for nested parallelism [14] are not runnable with OpenCL on MI100, as the size of nested parallelism (and thus the thread block size) exceeds 256, forcing the program to fall back on fully flattened code versions with worse locality. The *fft*, *smoothlife*, *nw*, *lud*, and *sgemm* benchmarks on MI100 suffer most from this. The wide error bars for *fft* and *smoothlife* are due to only the largest workloads being affected.

**3.1.4 Missing atomics.** OpenCL does not provide atomic operations on floating point values. Despite their nondeterminism due to the nonassociativity of floating point arithmetic, floating-point atomics are efficiently supported by CUDA and HIP and are used in the implementation of generalised histograms when the element type is floating point. With the OpenCL backend, such histograms are implemented with less efficient Compare-And-Swap (CAS) operations. However, this affects none of the benchmarks in this paper, and this potential cause is included only out of a sense of completeness.

**3.1.5 Imprecise cache information.** OpenCL makes it more difficult to query some hardware properties. For example, Futhark's implementation of generalised histograms uses the size of the GPU L2 cache to balance redundant work with reduction of conflicts through a multi-pass technique [12]. With CUDA and HIP we can query this size precisely, but OpenCL does not reliably provide such a facility[4]. The Futhark runtime system makes qualified guess that is close to the correct value, but which is incorrect on AMD

[2] https://github.com/diku-dk/futhark-fproper24

[3] `-cl-fp32-correctly-rounded-divide-sqrt`

[4] On AMD GPUs, the `CL_DEVICE_GLOBAL_MEM_CACHE_SIZE` property returns the *L1* cache size, and on NVIDIA GPUs it returns the L2 cache size.

GPUs. This affects some histogram-heavy benchmarks, such as (unsurprisingly) *histo* and *histogram*, as well as *tpacf*.

**3.1.6 Imprecise thread information.** OpenCL makes it difficult to query how many threads are needed to fully occupy the GPU. On OpenCL, Futhark makes a heuristic guess (the number of compute units multiplied by 1024), while on HIP and CUDA, Futhark directly queries the maximum thread capacity. This information, which can be manually configured by the user as well, is used to decide how many thread blocks to launch for scans, reductions, and histograms. In most cases, small differences in thread count have no performance impact, but *hashcat* and *myocyte* on MI100 are very sensitive to the thread count, and run faster with the OpenCL-computed number.

This also occurs with some of the *histogram* datasets on A100 (which explains the enormous variance), where the number of threads is used to determine the number of passes needed over the input to avoid excessive bin conflicts. The OpenCL backend launches fewer threads and performs a single pass over the input, rather than two. Some of the workloads have innately very few conflicts, which makes this run well, although other workloads run much slower.

The performance difference can be removed by configuring HIP to use the same number of threads as OpenCL. Ideally, the thread count should be decided on a case-by-case basis through auto-tuning, as the optimal number is difficult to determine analytically.

**3.1.7 API overhead.** For some applications, the performance difference is not attributable to measurable GPU operations. For example, *trace* on the MI100 is faster in wall-clock terms with HIP than with OpenCL, although low-level profiling information reveals that the runtimes of actual GPU operations are very similar. This benchmark runs for a very brief period (around 250$\mu s$ with OpenCL), which makes it sensitive to minor overheads in the CPU-side code. We have not attempted to pinpoint the source of these inefficiencies, but generally we observe that they are higher for OpenCL than for CUDA and HIP. Benchmarks that have a longer total runtime, but small individual GPU operations, are also sensitive to this effect, especially when the GPU operations are interspersed with CPU-side control flow that require transfer of GPU data. The most affected benchmarks on MI100 include *nn* and *cfd*. On A100, the large variance on *nbody* is due to a small workload that runs in 124$\mu s$ with OpenCL, but 69$\mu s$ with CUDA, where the difference due to API overhead, and similar case occurs for *sgemm*.

**3.1.8 Bounds checking.** Futhark supports bounds checking of code running on GPU, despite lacking hardware support, through a program transformation that is careful never to introduce invalid control flow or unsafe memory operations [11]. While the overhead of bounds checking is generally quite small (around 2-3%), we conjecture that its unusual control flow can sometimes inhibit kernel compiler optimisations, with inconsistent impact on CUDA, HIP, and OpenCL. The *lbm* benchmark on both MI100 and A100 is an example of this, as the performance difference between backends almost disappears when compiled without bounds checking (although interestingly, with the bounds checks, OpenCL does better on MI100, and worse on A100).

**3.1.9 Inexplicable.** Some benchmarks show inexplicable performance differences, where our investigation has been unable to pinpoint the cause. For example, *LocVolCalib* on MI100 is substantially faster with OpenCL than HIP. This is down to a rather complicated kernel that performs several block-wide scans and stores all intermediate results in shared memory. Since this kernel is compute-bound, its performance may be sensitive to minor aspects of register allocation or instruction selection, which may differ between the OpenCL and HIP kernel compilers. GPUs are very sensitive to register usage, as high register pressure lowers the number of threads that can run concurrently. The Futhark compiler leaves all decisions regarding register allocation to the kernel compiler. Similar inexplicable performance discrepancies for compute-bound kernels occur on the MI100 for *tunnel* and *OptionPricing*.

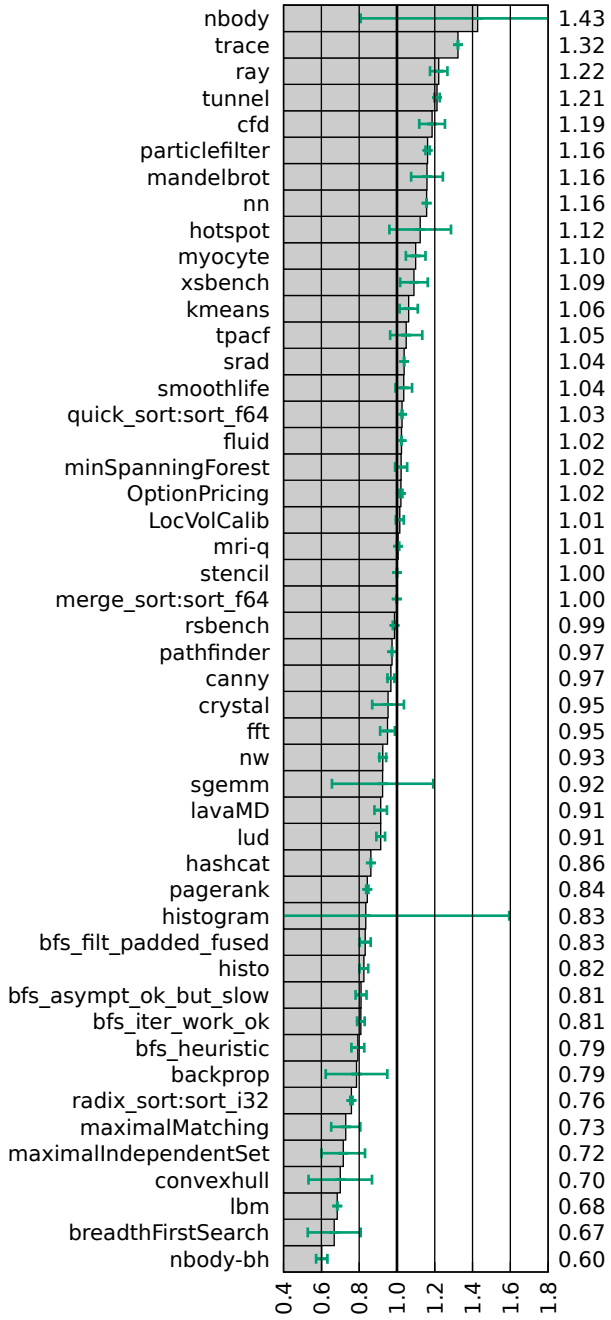
### 3.2 Discussion

Based on the results above, we might reasonably ask whether targeting OpenCL is worthwhile. Almost all cases where OpenCL outperforms CUDA or HIP are due to unfair comparisons, such as differences in default floating-point behaviour (section 3.1.1), or scheduling decisions based on inaccurate hardware information (sections 3.1.5 and 3.1.6) that happens to perform well by coincidence on some workloads. On the other hand, when OpenCL is slow, it is because of more fundamental issues, such as missing functionality (sections 3.1.2 to 3.1.4) or API overhead (section 3.1.7).

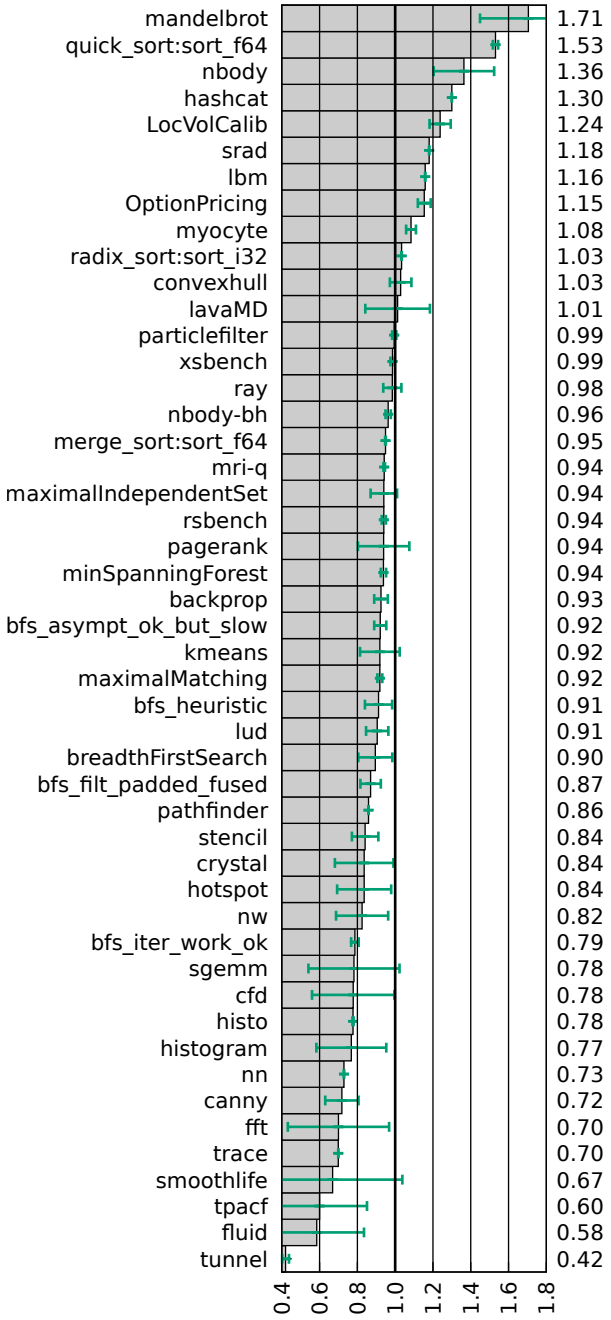
One argument in favour of OpenCL is its portability—an OpenCL program can be executed on any OpenCL implementation, which includes not just GPUs, but also multicore CPUs and more exotic hardware such as FPGAs [5]. However, OpenCL does not guarantee *performance portability*, and it is well known that OpenCL programs may need significant modification in order to perform well on different platforms [26, 32]. Indeed, the Futhark compiler itself uses a completely different compiler pipeline and code generator in its multicore CPU backend [22, 31].

A stronger argument in favour of OpenCL is that it is one of the main APIs for targeting some hardware, such as Intel Xe GPUs [20]. An interesting avenue of future work would be to investigate how OpenCL performs compared to the other APIs available for that platform.

Finally, a reasonable question is whether the differences we observe are simply due to Futhark generating poor code. While this possibility is hard to exclude in every case, Futhark




**(a)** Speedup of OpenCL backend over CUDA backend on NVIDIA A100 GPU.




**(b)** Speedup of OpenCL backend over HIP backend on AMD MI100 GPU.

**Figure 2.** Performance comparison on 48 benchmarks on two different GPUs. The error bars show the standard deviation of *speedups* across the different workloads for a single benchmark, not variability in absolute runtime performance. Thus, a wide error bar indicates that the performance difference between backends depends on the workload.

has been demonstrated in the past to generate code that is often competitive with hand-written programs [12, 14, 24, 28], including evaluations on various subsets of the benchmarks considered above, and so we find it reasonable to assume that the code generated by Futhark is adequate for the 48 programs investigated in this paper.

## 4 Conclusions and Related Work

Based on a comparison of 48 benchmarks, we have found that even for equivalent code running on the same GPU, there can be significant performance differences between OpenCL versus CUDA, and OpenCL versus HIP. The differences are often down to differing defaults regarding floating point behaviour or unavailability of accurate hardware information. However, there are also substantial performance differences due to missing OpenCL features, such as floating-point atomics or a memory model that allows single-pass scans. We were unable to explain all performance differences and suspect small differences in register allocation, as the inexplicable cases tended to be for mostly compute-bound applications.

The basic compilation model of Futhark, where high level parallel constructs are mapped to equivalent GPU code, is fundamentally similar to the model used by other array languages such as Accelerate [3], SAC [9], Co-dfns [15], SkePU [7], or RISE [10]. While the specifics of optimisation and available constructs differ, most of the causes identified in section 3.1 are likely to remain relevant.

Prior performance comparisons of CUDA and OpenCL exist [6, 8, 17]. They tend to include only a few programs and often ones with simple kernels. This is likely because of the significant amount of tedious work required to port a CUDA program to OpenCL or vice versa. However, because the amount of code being studied is smaller, the investigation of the performance differences can be quite detailed. In contrast, this paper investigates a large set of somewhat complicated benchmarks, albeit with less in-depth analysis. The prior work, such as [8], tends to find that performance differences are often due to "unfair" comparisons, such as different defaults for grid sizes or floating-point behaviour. This is largely in line with our observations in section 3.1.

## Acknowledgments

Oclgrind [27] has proven invaluable in the development of the Futhark compiler. Cosmin Oancea initiated the Futhark project and has contributed significantly to the compiler and the backends discussed in this paper. Robert Schenck has also contributed significantly to the compiler, and assisted in improving the readability of the paper.

## References

[1] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 445–447. https://doi.org/10.1145/3503221.3508422

[2] Jakob Stokholm Bertelsen. 2019. *CUDA backend.* Bachelor's Thesis. University of Copenhagen.

[3] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 3–14.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[5] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 531–534.

[6] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391–407. https://doi.org/10.1016/j.parco.2011.10.002 APPLICATION ACCELERATORS IN HPC.

[7] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming* 46 (2018), 62–80.

[8] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*. IEEE, 216–225.

[9] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24. https://doi.org/10.1145/1926354.1926359

[10] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[11] Troels Henriksen. 2021. Bounds Checking on GPU. *International Journal of Parallel Programming* (03 2021). https://doi.org/10.1007/s10766-021-00703-4

[12] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 97, 14 pages.

[13] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[14] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[15] Aaron W Hsu. 2016. The key to a data parallel compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 32–40.

[16] Aleksander Junge. 2022. *Reactive Benchmarking.* Bachelor's Thesis. University of Copenhagen.

[17] Kamran Karimi, Neil G Dickson, and Firas Hamze. 2010. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581* (2010).

[18] Niklas Kerscher. 2022. Investigating the HIP programming model with regards to portability and performance portability. (2022).

[19] Khronos OpenCL Working Group. 2009. *The OpenCL Specification, Version 1.0.* https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[20] Filip Kruzel and Mateusz Nytko. 2022. Intel Iris Xe-LP as a platform for scientific computing.. In *FedCSIS (Communication Papers)*. 121–128.

[21] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

[22] W. Pema N. H. Malling, Louis Marott Normann, Oliver B. K. Petersen, and Kristoffer A. Kortbæk. 2022. *Extending Futhark's multicore C backend to utilize SIMD using ISPC*. Bachelor's Thesis. University of Copenhagen.

[23] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[24] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. 2022. Memory Optimizations in an Array Language. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 424–438. https://doi.ieeecomputersociety.org/

[25] NVIDIA. 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2024-05-29.

[26] Simon J Pennycook, Simon D Hammond, Steven A Wright, JA Herdman, Ian Miller, and Stephen A Jarvis. 2013. An investigation of the performance portability of OpenCL. *J. Parallel and Distrib. Comput.* 73, 11 (2013), 1439–1450.

[27] James Price and Simon McIntosh-Smith. 2015. Oclgrind: An extensible OpenCL device simulator. In *Procs. of the 3rd Int. Workshop on OpenCL*. ACM, 12.

[28] R. Schenck, O. Rønning, T. Henriksen, and C. E. Oancea. 2022. AD for an Array Language with Nested Parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 829–843. https://doi.ieeecomputersociety.org/

[29] Dave Shreiner et al. 2009. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education.

[30] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[31] Duc Minh Tran. 2020. *Multicore backend for Futhark*. Bachelor's Thesis. University of Copenhagen.

[32] Yao Zhang, Mark Sinclair, and Andrew A Chien. 2013. Improving performance portability in OpenCL programs. In *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings 28*. Springer, 136–150.

Received 2024-06-15; accepted 2024-07-05