

chipStar: Making HIP/CUDA Applications Cross-Vendor Portable by Building on Open Standards

Paulius Velesko¹, Pekka Jääskeläinen⁴, Henry Linjamäki⁴, Michal Babej⁴, Peng Tu³, Sarbojit Sarkar³, Ben Ashbaugh³, Colleen Bertoni², Jenny Chen⁹, Philip C. Roth⁵, Wael Elwasif⁵, Rahul Kumar Gayatri⁶, Jisheng Zhao⁷, Karol Herbst⁸, Kevin Harms², Brice Videau²

Abstract

We describe *chipStar*, an open source software stack that enables building unmodified CUDA and HIP programs into binaries that rely solely on open cross-vendor compute standards OpenCL and SPIR-V. The relevant technical aspects of *chipStar* and the feature mismatches between the CUDA/HIP APIs and OpenCL are discussed along with a set of standard extension proposals to bridge the essential gaps in the future. The key benefit of the software stack is its portability, which is demonstrated by providing performance evaluations on a diversity of less common CPU/GPU platforms including RISC-V/PowerVR and ARM Mali. A comparison against the original AMD HIP platform provides a geometric mean of 0.75, a reasonable price to pay for the enhanced portability. *chipStar* is now considered mature enough for wider testing and even production use, which is demonstrated by successful porting and competitive performance of GAMESS-GPU-HF, a complex HPC application.

Keywords

CUDA, HIP, OpenCL, SPIR-V, Portability, Shared Virtual Memory

Introduction

The walled garden strategy is popular among market-dominating companies. The idea behind walled garden strategy is to lock in customers to a company’s products by making escaping the gates of the garden as costly as possible. NVIDIA’s CUDA software platform is considered to be one of such walled gardens. It helps NVIDIA to expand and retain their GPU market advantage, and at the same time maintain a high innovation pace on the software APIs since there is no need to work with standardization committees that always have to aim for a consensus among multiple participating vendors.

Naturally, for end-users and the competing hardware vendors, the situation of a single-vendor dictated API is not ideal. End-users would prefer open standard software interfaces that enable switching the targeted hardware without incurring significant non-recurring engineering costs required for porting the applications and libraries to a new software platform just to be able to utilize the newly purchased hardware optimally. Similarly, other hardware vendors, aiming to get their piece of the market pie, would prefer an API that is not controlled by a single vendor.

AMD’s ROCm (AMD, Inc 2023) software platform and its Heterogeneous-compute Interface for Portability (HIP) language (AMD, Inc. 2025a) helps escaping the CUDA walled garden by providing a route out from the NVIDIA CUDA platform to AMD’s devices. HIP defines a subset of CUDA that is more easily portable to various hardware, thanks mainly to omitting various advanced features available in the later CUDA versions.

In order to enable an easy automated transition path from CUDA applications, HIP is largely a copy of a CUDA C/C++ API subset with a few minor differences and renamed functions. HIP alleviates the CUDA portability problem, but doesn’t solve it satisfactorily due to AMD targeting their self-specified low-level ROCm APIs which are not actively supported on non-AMD platforms. An open source HIP/CUDA software platform solely based on open standards with a sincere aim for cross-vendor portability is still lacking.

With the *chipStar* software stack described in this article we aim to alleviate the CUDA/HIP application portability problem. In contrast to previous solutions that either require source-to-source conversion from CUDA programs (Intel Corporation 2021), that can lead to costly multiple codebase maintenance, or aim for binary-level compatibility of existing CUDA/HIP programs that rely on questionable reverse engineering of proprietary binary interfaces – a brittle longer-term strategy – *chipStar* chooses a middle-ground approach which enables source-level compatibility of HIP/CUDA programs by compiling them to a runtime portable “fat binary” (a single executable that contains multiple code objects, each compiled for a different architecture) that utilizes solely open standards and can

¹PGLC Consulting ²Argonne National Laboratory ³Intel Corporation
⁴Tampere University ⁵Oak Ridge National Laboratory ⁶National Energy Research Scientific Computing Center ⁷Georgia Institute of Technology
⁸Red Hat, Inc. ⁹Purdue University

Corresponding author:

Paulius Velesko, pvelesko@pglc.io

execute on any platform supporting the required standard features without recompilation.

With this article and the associated open source code base we make the following contributions¹:

1. We publish internal design choices for the software platform *chipStar* that enables porting applications from the NVIDIA-driven CUDA and AMD-driven ROCm platforms to any current and future platform supporting the OpenCL and SPIR-V cross-vendor open standards.
2. We evaluated *chipStar* performance in comparison to running the CUDA applications directly using the NVIDIA SDK or converting the applications to a popular open-standard-based CUDA alternative SYCL.
3. We demonstrate a case study for the usability of OpenCL as a portability layer to implement other languages/APIs on top. Portability is shown by providing performance numbers on a RISC-V CPU & PowerVR GPU, ARM CPU and GPU, as well as on discrete GPUs from NVIDIA, AMD, and Intel.

The rest of the article is structured as follows: the “Rationale for the Chosen Standard APIs” Section discusses our rationale for choosing OpenCL (Khronos® OpenCL Working Group 2025c) and its device-side program representation SPIR-V (John Kessenich and Boaz Ouriel and Raun Krisch 2023) as the core APIs to support runtime portability in *chipStar*. The “Implementing HIP/CUDA on OpenCL Runtime API” Section details the key technical issues in implementing the HIP/CUDA runtime on these APIs, followed by the Compilation Aspects Section. Performance evaluation results are shown in the Evaluation Section, followed by a section presenting and HPC Application Case Study with the GAMESS-GPU-HF code. The Conclusions Section summarizes the paper and discusses future work.

Rationale for the Chosen Standard APIs

As a technical background, we review the various options available for the portable runtime API and the target-independent device program representation in *chipStar* and provide a rationale for our choice. The discussion is split into 1) runtime APIs used to control the execution from the host side and 2) device program representations providing an abstraction for the kernel side programs in a portable manner. Due to the abundance of potential target devices available for acceleration, we consider it important to be able to embed the device programs in an open standard-based Intermediate Representation (IR) and to use just-in-time (JIT) compilation for lowering the program to the target Instruction Set Architecture (ISA) of the accelerator at deployment or launch time. This enables future-proof “fat binaries” which can be supported on new platforms by implementing the specification of the IR. Another alternative would be to provide only source-level compatibility where the application needs to be recompiled for each host and a device pair of interest, hindering binary distribution.

Runtime APIs

In practice, both CUDA and HIP are single-vendor supported programming models. This is reflected, for example, in their platform property APIs which define limited queries for device properties, highlighting features in each vendor’s GPU offerings. The goal of *chipStar* is to expand the portability of applications implemented using the CUDA/HIP APIs. Therefore, the key requirement to the underlying “platform/device portability API” is to cover as many of the essential features of CUDA/HIP as possible to provide functional correctness and to exploit the potential performance benefits. This includes, for example, parallel and asynchronous execution of tasks, overlapping of data transfers with task execution, and by providing access to shared memory communication, if available. Furthermore, the portability API should provide services to enhance performance portability of the implementation by allowing to query the capabilities of the devices to tune the execution at runtime to match the target’s features.

There are not a large number of choices for such a runtime API, especially if limiting the list to alternatives that enjoy official driver support from multiple accelerator vendors or to those that have a portable long-maintained open source implementation. In this regard, an open standard based API that has increased in popularity is SYCL (The Khronos® Group Inc 2014). SYCL resembles CUDA in being a C++-based single-source API. However, as a layer for supporting other languages on top, it lacks means to explicitly load and launch pre-built kernels defined in non-SYCL language at host runtime. It has an interoperability layer that allows one to build OpenCL C kernels, but then the additional value of using SYCL as a portability layer instead of using OpenCL directly is not obvious.

Recently, OpenMP (OpenMP Architecture Review Board 2021) has been considered for portability layer usage and, in fact, specifically for implementing CUDA in (Doerfert et al. 2023). While OpenMP enjoys support from a wide range of vendors, we believe OpenMP is not ideal for this use case since it doesn’t define a device-side program representation, making future-proof cross-vendor portable fat binary generation difficult. It also can be considered a high-level “application-programmer-facing” API similarly to SYCL, thus offers constructs and overheads for programmer-productivity which are unnecessary for a portability layer.

Heterogeneous System Architecture (HSA) is an open heterogeneous platform specification that also defines a runtime API (Heterogeneous System Architecture Foundation 2021, 2016). A key differentiating feature of HSA is that it standardizes on shared virtual memory, making system-wide virtual memory addressing a required feature from implementations. In hindsight this requirement was too much too early, as system-wide virtual memory support is only recently appearing in hardware and still usually requires explicit allocation or mapping calls from the programmer. For this work, HSA could have been a valid choice for a lightweight portability layer, but activity on the specification has ceased, with mainly AMD using only selected parts of it in their software stack (the ROCr runtime component).

Another option to consider would be Vulkan (The Khronos® Vulkan Working Group 2023) since it also provides a compute pipeline stage which allows specifying

general purpose compute kernels. However, the feature set of the compute kernels lacks some of OpenCL's features such as Shared Virtual Memory (SVM) which would require further standard extensions. It might be that in the future the feature gap gets narrower and it would become a viable option. Meanwhile, layering OpenCL on top of Vulkan is an interesting option pursued by multiple open source projects to cover the devices which previously only had Vulkan driver support.

Level Zero (Intel Corporation 2025) is an API at a similar level of abstraction as HSA and OpenCL. However, it's currently only supported on Intel's devices, thus is not suitable for future-proof cross-vendor fat binaries.

Interestingly, OpenCL was originally created to provide an open cross-vendor alternative to the proprietary CUDA GPGPU (General Purpose GPU) programming model. After its introduction in 2009 it has received wide support from hardware vendors, but some application developers are known to dislike it because it can be considered too low level as an application-facing API and unproductive with the driver and feature support lagging behind the proprietary alternatives. However, as demonstrated in this paper, OpenCL can provide an excellent portability layer for implementing other higher-level programming models and APIs on top. This portability is due to multiple vendor's official support for the minimum feature set of OpenCL version 3.0 and to multiple long-maintained open source implementations. For these reasons, we have chosen OpenCL as the runtime through which to access the GPU.

Device Program Representations

Heterogeneous platforms suffer from the problem of device-side program description portability. There is a wide range of instruction-set architectures the kernels can target, and when the program is distributed in a binary form, the targets are known only at run time. Thus, the choice of the format in which the device programs (kernels) are stored is critical as it should cover as many of the potential targets as possible. Furthermore, the representation should be "future-proof" in a sense that the produced fat binaries could be made to run on entirely new platforms by only referring to the API specification. At the time of this writing, there still seems to be no clear winning program representation in this regard. Various portable implementations of application-facing APIs resort to very fat binaries which store copies of the device program in multiple (virtual) instruction-set architectures to cover the various targets and offloading runtimes it might encounter at execution time. This is the case with (Doerfert et al. 2023) and originally in AdaptiveCpp (Alpay et al. 2022).

Recently AdaptiveCpp started storing kernels in the LLVM (Lattner and Adve 2004) compiler IR instead of storing multiple different binaries depending on the target. In this scheme, LLVM IR is lowered to various target-dependent formats at runtime at the point when the target is known (Alpay and Heuveline 2023). This approach has benefits in comparison to storing abundance of device binaries in the another alternative, and works in theory, but it is also known that LLVM IR is not supposed to be a portable program representation as it can embed target-specific intrinsics, has target specific data layout

and endianness among other challenges. LLVM IR is not guaranteed to be stable across LLVM versions, which means that the fat binaries should have access to an LLVM library of version the IR was generated with, which at worst requires to embed the LLVM library along and the required backends to the fat binary, forming an unnecessary dependency. The problem of LLVM IR not being target-independent nor stable across LLVM versions was attempted to be addressed by earlier Standard Portable Intermediate Representation (SPIR) versions 1.2 and 2.0 (The Khronos Group Inc 2014): These first SPIR versions were designed to support OpenCL C language kernels and were based on defined versions of LLVM IR, which proved to be difficult to maintain long term. LLVM-based SPIR versions were later obsoleted in favor of the new version of SPIR (SPIR-V) (John Kessenich and Boaz Ouriel and Raun Krisch 2023) format. The goal for SPIR-V is to provide a robust cross-vendor specified intermediate language which is not affected by LLVM upstream changes and that shares specification effort with the Vulkan community.

HSA specification defines an intermediate language called Heterogeneous system architecture intermediate language (HSAIL) and a binary representation called BRIG (Heterogeneous System Architecture Foundation 2018). A key technical difference between HSAIL and SPIR-V format, is that HSAIL has a fixed number of registers and an address space for spills unlike SPIR-V, which has infinite virtual registers due to being based on the Static Single Assignment (SSA) (Cytron et al. 1991) representation. HSA made a choice to not define a higher-level programming language (like OpenCL C) for the device programs, but only standardized a low level IR. As with the HSA runtime specification, however, the activity on the HSAIL specification has stalled. There was also a GCC-based frontend for consuming BRIGs in a target-portable fashion, but after activity on HSA quieted, the "BRIG frontend" was removed from the upstream GCC source code repository in a May 2021 commit.

In conclusion, while official SPIR-V OpenCL environment support from processor vendors is extensive as of this writing, it seems to be still the best option for a cross-platform representation given that it is an open standard defined democratically by multiple hardware vendors and is relied upon by OpenCL, Vulkan, and SYCL implementations among other use cases such as Direct-X adopting it. In addition, thanks to open source tooling support available and useful SPIR-V producers such as *chipStar* and Intel oneAPI Data Parallel C++ (DPC++) appearing, the list of supported targets is expected to grow in the future. Therefore, we considered SPIR-V and OpenCL to be future-proof open standards on which to base the implementation.

Implementing HIP/CUDA on OpenCL Runtime API

The primary goal for *chipStar* is to support the subset of CUDA features as defined by HIP and expand the feature set beyond it whenever feasible while relying on the chosen open standard APIs as much as possible. In this section, we discuss how the OpenCL/SPIR-V specifications can be matched with the commonly used features of CUDA/HIP and identify the

most impactful gaps that we believe should be covered in the future.

Memory Model

Due to their common history in GPGPU programming, CUDA/HIP and OpenCL share various platform and memory model abstractions. For example, “device memory” is the same as “global memory” in OpenCL terminology (“shared” is “local”). To avoid confusion in terminology we use only the CUDA/HIP terms in the rest of this article. Similarly, we refer to the original CUDA versions when talking about functions that have their counterparts in the HIP API.

A key difference between OpenCL and CUDA that required addressing is the fact that CUDA implicitly infers the address space of the data in the device program side whereas in OpenCL (before v2.0) the address space must be declared explicitly. The CUDA’s implicit address space inference is similar to the ‘generic’ address space concept introduced in OpenCL v2.0, which was utilized to bridge this gap.

The simplest interface in CUDA’s host-side device memory management is `cudaMalloc()`. It returns a raw pointer to the targeted device’s global memory, instead of an opaque buffer handle. In order to implement these capabilities, we utilized the SVM API that first appeared in the OpenCL v2.0.

The SVM allocation API returns a raw pointer to a shared virtual address space region. The “Coarse-grained buffer (CG) SVM” variant can be used to implement the basic device memory allocation. Mapping device memory allocation to CG SVM has a drawback that the device driver must support some of the unneeded SVM features such as mapping the allocated regions to the virtual address space although just returning physical device memory pointers would suffice. This means that the *chipStar* implementation is actually implementing CUDA’s Unified Memory model by default. To alleviate the potential performance impact of this, *chipStar* can also use the Intel Unified Shared Memory (USM) extension (`cl_intel_unified_shared_memory` (Intel Corporation 2026)), if supported by the runtime. USM enables allocating strictly device-only allocations, but still returns virtual pointers, which can be problematic for some implementations.

In order to provide a minimal allocation API matching the basic `cudaMalloc()`’s needs without requiring a SVM-capable OpenCL driver, we introduced a new extension (`cl_ext_buffer_device_address` (Khronos® OpenCL Working Group 2025a)) that enables querying the raw device pointer of a `cl_mem` allocation without needing to map the buffer to the same address range in the host’s virtual memory. *chipStar* can use any of these alternative memory management APIs, if advertised by the targeted OpenCL device.

CUDA provides an API to *pin* memory so it’s kept resident in the host memory and optionally made accessible by devices from kernel code and is not swapped out to disk. The primary APIs to this functionality are `cudaHostAlloc()` and `cudaHostRegister()`. The former allocates pinned memory directly and the latter pins a previous host allocation. `cudaHostAlloc()` is simple to implement with coarse grained SVM since by the coincidence of using a shared virtual memory allocation, the buffers are by default

accessible in both the host and the device using the same pointer. However, the allocation might not be resident for the duration of the execution, for example, if a CPU device is allowed to swap out such allocations. However, that aspect can only be observed by the programmer as a performance difference.

`cudaHostRegister()` is a bit more challenging to implement on top of CG SVM since it allows registering a host address range to be a pinned region accessible both from the host and the device *after* the host memory has been allocated. Since the allocation might not have been originally allocated with the OpenCL SVM allocation API, but with a system memory allocator or even from the stack, to implement correct functionality in this case, *chipStar* creates a shadow buffer using `clSVMAlloc()` and synchronizes it with the host region at kernel start and end points. OpenCL 2.1 added a new `clEnqueueSVMmigrateMem()` API that enables fine grained specification of where regions of SVM are migrated, but it is not useful for this case since the source of `cudaHostRegister()` can be any host memory area whereas the API handles only SVM allocations.

The NVIDIA architectures post-compute capability 6 support on-demand page migration which relies on the hardware memory management unit (page fault-based buffer migrations) for coherence of the Unified Memory allocations. This frees the programmer from the need to perform explicit memory allocation and synchronization calls. This functionality maps to the Fine-Grained System SVM of OpenCL, but since hardware and driver support for fine-grain SVM is very rare at the time of this writing, on-demand page migration is not yet implemented by *chipStar*.

Tasks and Events

The semantics of CUDA *streams* and the ability to execute tasks/commands asynchronously maps well to the *command queues* of OpenCL. Each stream is expected to execute commands in-order, which matches the in-order command queue semantics of OpenCL. Commands are allowed to execute concurrently even within in-order command queues in OpenCL, as long as the results are not observable from the outside, enabling concurrent kernel execution (Khronos® OpenCL Working Group 2025c).

Textures

chipStar supports only a subset of texture objects due to a limitation in OpenCL images. The notable differences between HIP/CUDA and OpenCL are that the texture objects are pointers to opaque C/C++ structures whereas in OpenCL/SPIR-V there is a special type per image dimensionality and that the texture objects can be loaded indirectly whereas OpenCL images can be only passed to kernels as kernel arguments. Therefore, some constructs such as the following cannot be expressed in SPIR-V:

```
hipTextureObject_t Tx = ...;
Ty Tv = cond ? tex2D<Ty>(Tx, X, Y)
             : tex1D<Ty>(Tx, X)
```

An LLVM pass is responsible for lowering texture object API based texture functions to OpenCL image fetches. The pass analyses endpoints of the texture objects by following

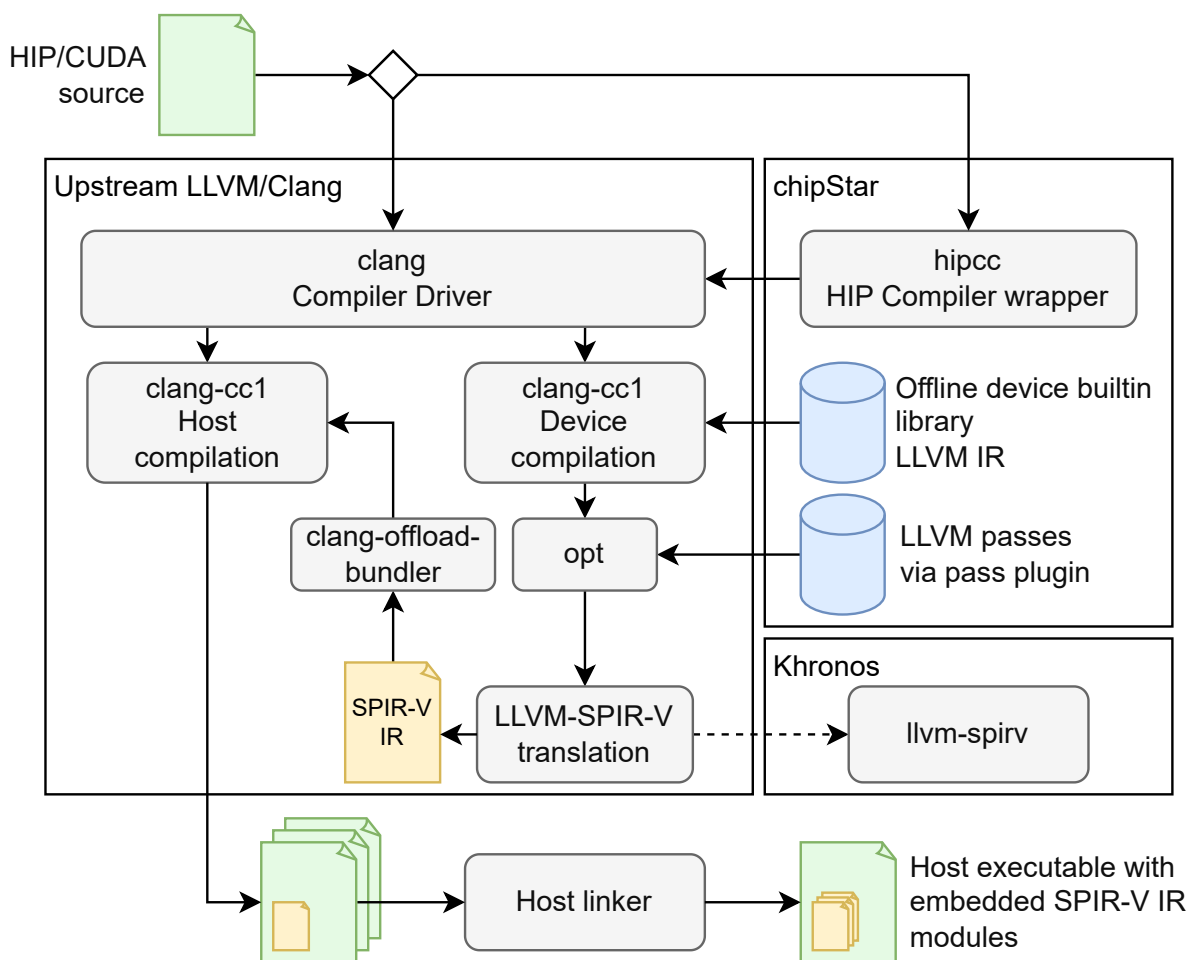


Figure 1. The construction of a HIP fat binary. Device code is compiled into LLVM IR, chipStar applies a series of LLVM passes at optimization time. The resulting IR is embedded into the binary and linked against the device library.

their use-def and def-use chains. If the pass sees that a texture object is coming from a kernel parameter and it is only used by texture fetch calls for the same dimensionality, it will replace the texture object parameter with image and sampler parameters and translates the texture fetch calls with OpenCL image fetch calls of matching dimensionality which consume the new kernel parameter.

Compilation Aspects

This section discusses the compilation flow used by *chipStar*. We introduce the overall compilation flow, the device library implementation and summarize our findings on the key needs to extend the OpenCL/SPIR-V standards to bridge key feature gaps between the specifications.

The Compilation Flow

The offline compilation flow of *chipStar* is built on the LLVM Project’s (Lattner and Adve 2004) Clang² frontend which provides the frontend language handling and splitting of the single source input to the device and host parts. The overall compilation process is shown in Fig. 1. It relies on the CUDA/HIP frontend of Clang, which was extended to produce SPIR-V binaries as an option to PTX or AMDIL for the device program. The LLVM *opt* tool is used to invoke special LLVM passes provided by *chipStar* for lowering HIP

features to the OpenCL-SPIR-V environment. The SPIR-V translation is performed using Khronos’ LLVM-SPIR-V Translator tool (The Khronos® Group Inc 2019).

Most of the compilation-related changes have been upstreamed to the LLVM project and very little compilation-related functionality remains within the *chipStar* code base. The notable exceptions are compiler passes that handle CUDA vs. OpenCL differences in *printf()*, implement a device side *abort()* feature, handling of CUDA’s device-side global variables, and an indirect memory access analyzer. The indirect memory access analyzer marks kernels that are known to not indirectly access allocations, which removes unnecessary synchronizations for the majority of benchmarks seen so far. Otherwise, due to CUDA’s memory model, each launcher kernel can potentially access any previously allocated buffer, inducing significant unnecessary data synchronization overheads in the common case where the kernels only access buffers set through their arguments.

Lazy Just-in-Time Compilation

Fig. 2 shows the online compilation flow from SPIR-V to device code in the *chipStar* runtime. When a kernel launch is requested, the kernel function stub pointer is used to look up the associated SPIR-V module which, in turn, is JIT compiled to machine code. To enhance runtime portability, the built-in library of the on-line device provides variations

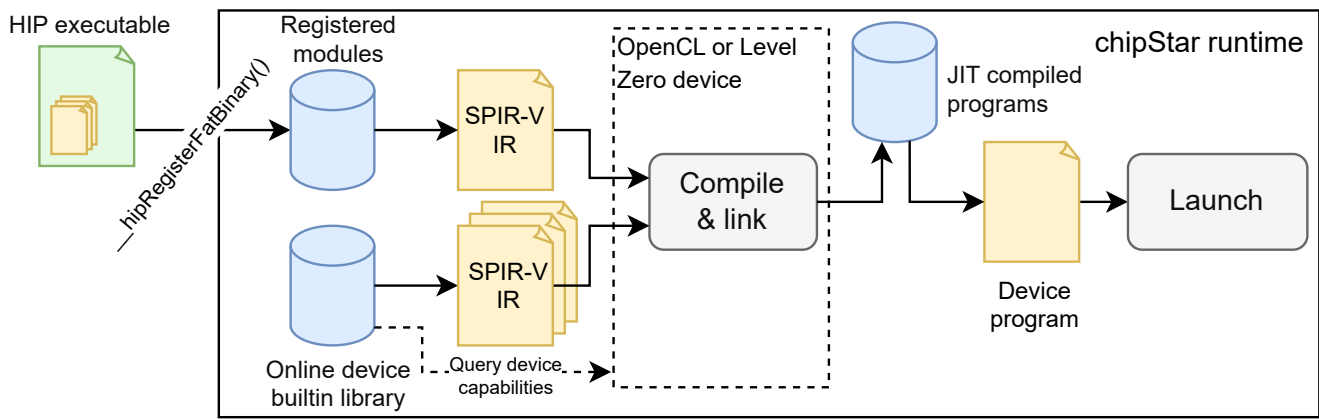


Figure 2. The just-in-time compilation flow. Once the program is executed, prior to calling main, a series of internal HIP calls are executed. These calls parse the fat binary and register the embedded SPIR-V files as modules. In lazy JIT mode, these modules are not compiled into kernels until their launch is requested.

in built-in HIP functions for different device capabilities that are linked to the user’s device programs at runtime. For example, for HIP floating-point atomics the runtime chooses between an implementation that maps them to corresponding native functions via a SPIR-V extension or emulates them via atomic exchange operations.

chipStar implements a lazy JIT compilation strategy to avoid compiling all kernels of large SPIR-V libraries such as those in neural network libraries, unless called from the host program. When compilation occurs, the SPIR-V binary is converted to the target format, build flags and options are set up, and backend-specific compilation methods (such as *clCreateProgramWithIL* for OpenCL) are used. Once compiled, modules are cached for future use. This approach significantly reduces startup time for applications with many kernels, as only the necessary kernels are compiled.

The implementation includes performance monitoring features, tracking compilation time and logging whether modules are loaded from cache or compiled fresh. The lazy JIT system is designed to be backend-agnostic, allowing different backends (Level0, OpenCL, etc.) to implement their own specific compilation strategies while maintaining consistent lazy compilation behavior. This flexibility enhances *chipStar*’s portability across various platforms and backends.

Device Library

The *chipstar* device-side library implements the HIP math API, by using a combination of OpenCL C math built-ins, LLVM built-ins, OCML (part of ROCm-Device-Libs), and custom implementations. Many of the functions in the HIP math API have an equivalent OpenCL built-in with adequate accuracy guarantees. However there are a few exceptions that cannot be mapped directly, and thus require software-based emulation such as floating-point atomics on some devices. The main challenge in terms of a fast yet portable implementation of the functions is due to differences in math accuracy requirements between CUDA/HIP and OpenCL. While CUDA provides very specific ULP accuracy requirements (NVIDIA Corporation 2018b) (most of which are higher than those in the OpenCL specification (Khronos® OpenCL Working Group 2025b)) for their math library, no

such requirements were specified for HIP 6.1 which *chipStar* implements. These ULP requirements are now specified as of HIP 6.3 (AMD, Inc. 2025b) but still seem to diverge from those required by CUDA.

Furthermore, CUDA/HIP defines a set of *intrinsics*, which are faster yet less accurate versions of the standard functions. This exposes a further difficulty when aiming for a portable, yet fast implementation: the level of accuracy achievable depends heavily on the targeted platform. Since CUDA is inherently meant not to be cross-vendor portable, the intrinsics are defined only to match the CUDA microarchitecture in an optimal manner, which might not be the case for other devices.

OpenCL covers the use case of accessing fast but less accurate hardware operations by means of 1) a relaxed mathematics flag that can be enabled at device program build time and 2) with so-called native built-in functions in the built-in kernel API. Unfortunately, neither of these are usable for implementing the CUDA intrinsics by default due to not guaranteeing enough ULP accuracy as required by CUDA API. The relaxed math in OpenCL defines maximum rounding errors, but they are usually slightly less than what the CUDA intrinsics require Fig. 1. The OpenCL native built-in functions are an even worse fit for this use since they guarantee nothing of the accuracy but leave it entirely up to the implementation Fig. 2. There is not even a possibility to query for the maximum error via a runtime API, the accuracy must be discovered via trial-and-error or from documentation of the hardware vendor.

Previous research (Rydahl et al. 2023) shows significant disparities in the precision of GPU math functions, with CUDA consistently achieving the highest accuracy, followed by HIP, LLVM, and OpenCL (estimated based on relaxed math). CUDA’s functions maintain ULP errors close to 1, while HIP and LLVM exhibit progressively higher errors, sometimes exceeding 3-5 ULP. OpenCL’s relaxed math and native built-in functions lack strict accuracy guarantees, making it challenging to replicate CUDA’s precision without significant adjustments or corrections.

The “correctness first” principle mandates the use of arithmetics that have guaranteed accuracy, which means to not receive any performance benefits of simplified

Function	Intel Arc A770	Intel UHD 770	AMD gfx906	Intel i9-13900K
cos	1	1	1	2
exp	2	2	1	1
log	1	1	2	1

Table 1. Maximum ULP Differences for Standard OpenCL Math Functions. OpenCL specifies ULP requirements of 4, 3, and 3 for cos, exp, and log, respectively.

Function	Intel Arc A770	Intel UHD 770	AMD gfx906	Intel i9-13900K
native_cos	27.54	27.54	6.07	1084.87
native_exp	0.84	0.84	0.67	474.68
native_log	0.36	0.36	0.46	154.81

Table 2. Average ULP Differences for Native OpenCL Math Functions. OpenCL does not have ULP requirements for native functions.

Extension	CUDA/HIP feature(s)
cl_intel_unified_shared_memory	Used for optimized <i>cudaMalloc()</i> when available.
cl_ext_buffer_device_address	Used for optimized <i>cudaMalloc()</i> when USM nor SVM are available.
cl_intel_required_subgroup_size	Used to fix the warp-size.
cl_khr_fp64	If double precision floating point is used.
cl_khr_subgroups	Warp-level synchronization with <i>__syncwarp()</i> .
cl_khr_subgroup_ballot	Warp-level ballot operations.
cl_khr_subgroup_shuffle	Warp-level shuffle operations.

Table 3. OpenCL 3.0 standard extensions that *chipStar* can use currently to implement CUDA/HIP features if the compiled application uses them.

implementations. This approach is not taken by DPC++, for example, as it defaults to a more relaxed floating point precision model (-fp-model=fast). Achieving ULP requirements specified in the CUDA API would require emulating them in software. Performance impact would likely be too drastic to make *chipStar* usable for high performance workloads. Since we cannot make any guarantees about precision, we have chosen to map regular and intrinsic functions to the same OpenCL default accuracy function implementations.

We plan to optimize this aspect in the future via a new standard extension with a set of built-ins that guarantee the CUDA accuracy requirements to the application programmer while enabling the targeted platform to optimize and implement them as efficiently as possible.

OpenCL Extensions

The *chipStar* compilation flow is built such that different advanced OpenCL features and extensions are not required from the target platform’s driver or device unless the compiled input application specifically needs them. Although the minimal OpenCL 3.0 feature set plus coarse-grained SVM and SPIR-V consumption support covers a significant part of the most commonly used CUDA and HIP features, some functionalities require or can be improved with extensions to the OpenCL or SPIR-V specifications.

In Table 3 we summarize the standard extensions *chipStar* can already utilize and which CUDA/HIP feature triggers their need. Table 4 describes further work-in-progress extension proposals we have identified to be useful for CUDA/HIP portability. These extensions are in different stages in the Khronos Group standardization process, which is noted in the table.

Most of the extensions are relatively straightforward and the brief description in the table should suffice to grasp their purpose. However, the handling of warp-level primitives calls for a bit more thorough explanation: One of the execution model differences between CUDA and OpenCL is that CUDA presents a finer grained fixed size grouping of the threads (OpenCL work-items) than the blocks (work-groups) called a *warp*. In earlier CUDA versions, the threads in a warp could be assumed to execute in lock-step, implying that the enabled threads in the same warp would execute the same instruction. This implied that in some cases explicit synchronization could be omitted: In case of a usual read-modify-update case, the programmer could trust that the warp’s threads all execute the read part before any of them proceeds to the update part, enabling in-place-updates without explicit synchronization. In later versions of the CUDA specification, the use of lock-step behavior in program logic was deprecated, but the feature has to be supported for legacy applications (NVIDIA Corporation 2018a).

In addition to older CUDA programs potentially relying on the lock-step semantics to omit explicit synchronization, the fixed size warps (32 threads for NVIDIA and usually 64 threads in AMD devices) affect the execution semantics when executing warp-level functions that rely on the warp grouping and the mapping of the threads to the lanes of the warp. Such primitives include the warp shuffles, which read data from a specific lane within the warp, and the explicit warp synchronization primitives. The OpenCL specification, on the other hand, doesn’t have a warp concept, but the work-items are free to make progress in any order and grouping. The specification, however, has a feature extension called “subgroups” that is used to implement the warp semantics in

Extension (working title)	CUDA/HIP feature(s)	Status
cl_ext_alive_only_barrier	A special work-group barrier for barrier calls which might not be reached by work-items that have exited the kernel as allowed by the CUDA's execution model.	Draft.
cl_ext_cuda_math	Implement math functions and intrinsics with precision requirements that match CUDA's. To enable more optimized reduced precision intrinsics.	To be proposed.
cl_ext_device_side_abort	Implement <code>__trap()</code> on the low-level runtime side. The current implementation requires compiler transformations.	Public draft.
cl_ext_extended_device_properties	<code>hipGetDeviceProperties()</code> can be used to query more device properties than the basic OpenCL device or platform query APIs support, this fills the gap.	To be proposed.
cl_ext_relaxed_printf_address_space	CUDA's <code>printf()</code> behavior with non-constant address spaces. Currently handled with compiler transformations.	Public draft.
cl_khr_command_buffer	For optimized implementation of CUDA graph re-execution.	Public.
cl_ext_command_buffer_host_data	For optimized implementation of CUDA graphs which transfer data between the host and the device.	Draft.
cl_ext_command_buffer_host_sync	For optimized implementation of CUDA graphs which synchronize with the host.	Public draft.
cl_ext_subgroup_id_mapping	For forcing the desired thread id mapping when calling warp-level primitives that depend on the fixed warp size or the thread id ordering. but Luckily, in practical targets mapping is already the desired one by default.	Draft.

Table 4. Planned or drafted OpenCL 3.0 standard extensions that *chipStar* might use in the future to implement CUDA/HIP features if the application uses them. The status column describes the state of the extension at the time of this article's publication.

chipStar when the kernel is detected to need it. However, in contrast to warps which have a specified form and content which allows the programmer to utilize them reliably, the basic subgroups of OpenCL are "implementation-oriented"; they enable grouped execution in a manner that is simplest or most efficient for the driver and the hardware at hand. The sizes of the OpenCL subgroups are not fixed, but must be queried per kernel by the programmer in the basic extension. Also the way work-items are mapped to subgroup lanes so they can be referred to when using cross-lane intrinsics is also implementation-defined. To close the gap between subgroups and warps, an enhanced extension that *forces* the subgroup size of the kernel to the desired size along with the linear id mapping is being proposed.

Unsupported HIP/CUDA APIs

The following APIs have not yet been implemented as feature implementation timelines are driven by application requirements: Cooperative Groups, Memory Pools, Inter-process Communication, Peer-to-Peer Access, and Occupancy APIs. Implementing some of these APIs might require additional OpenCL extensions. This is left for future work.

Evaluation

We evaluated *chipStar* performance on various OpenCL-capable CPU and GPU platforms using a subset of the HeCBench benchmark collection (Jin and Vetter 2023). All of the results were produced using the *chipStar* v1.2.1 release.

The HeCBench benchmark application selection criteria for each comparison was as follows:

1. The application had the necessary API/language variations with a HIP version that could be built with *chipStar* v1.2.1 and its ported libraries.
2. For SYCL/CUDA comparisons, there must not have been significant identified performance-affecting structural or implementation differences between the SYCL/CUDA and HIP versions of the application. Some of the identified "unfair differences" were fixed and submitted to the HeCBench repository. The exact branch used for benchmarking can be found here: <https://github.com/CHIP-SPV/HeCBench/tree/chipStar-bench>
3. The application could verify its results and had to validate correctly on all platforms involved in the comparison.
4. Applications that required hardware or OpenCL driver features that were missing or too limited on the platform were omitted. This mostly concerned the runs on embedded/integrated GPUs with limited memory or lack of double precision floating point support.

OneAPI DPC++ on Intel oneAPI SDK

For this evaluation we chose a subset of benchmarks included in the HeCBench suite and compared the performance of their HIP versions of the benchmarks against the SYCL versions. The SYCL versions were compiled with Intel's DPC++ shipped with the oneAPI v2024.2.2 release.

The following consumer-grade hardware from Intel, Nvidia, and AMD was used to perform the benchmarking: Intel A770 discrete GPU, Intel UHD 770 integrated GPU, Nvidia RTX3060, AMD Vega VII and Intel i9-13900k CPU.

Since both *chipStar* and DPC++ can use OpenCL as a backend, the evaluations are performed using the same OpenCL driver on the same GPUs. This isolates the

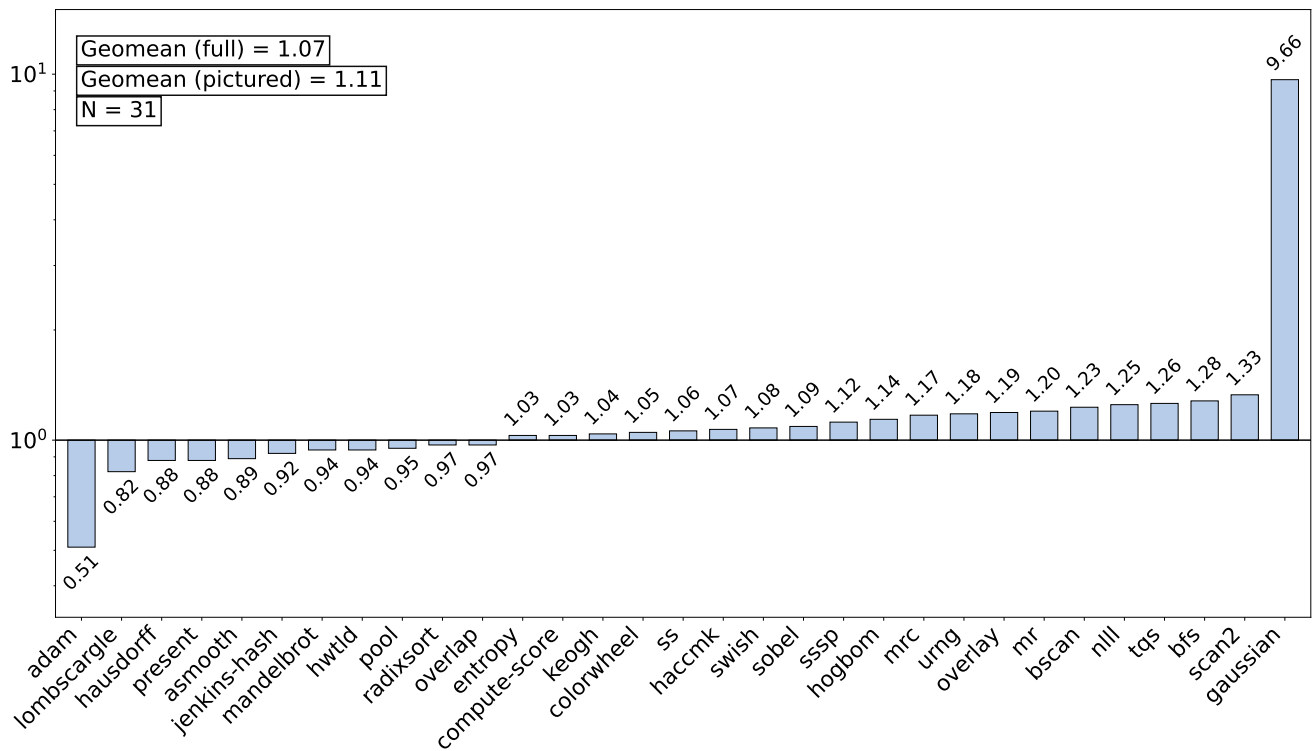


Figure 3. chipStar speedup over SYCL via DPC++ with relaxed math, excluding 20 benchmarks where the performance difference was under 3%

differences between the tested software stacks to the runtime and the LLVM IR level device code compiler optimizations.

In Fig. 3, we compare the expected fastest results between SYCL/DPC++ and chipStar when floating point relaxations are enabled to the extent the implementations can utilize them. As expected, there are only minor differences in the performance since both DPC++ and *chipStar* implement their respective runtime by leveraging OpenCL as the GPU API and have a similar compilation flow: kernels are compiled into LLVM IR which is then translated to SPIR-V which, in turn, is then JIT compiled to device code at runtime.

For the vast majority of benchmarks, the performance difference between DPC++ and *chipStar* was negligible, but there are outliers to both directions. DPC++ can currently benefit from FP relaxations more as it defaults to `-fp-model=fast` (Intel Corporation 2023) which enables optimizations like floating-point reassociation, fused multiply-add operations, and the omission of certain intermediate rounding steps, as well as the use of less accurate device-side math functions. The HIP/CUDA equivalent is `-use_fast_math` (NVIDIA Corporation 2026a) but *chipStar* v.1.2.1 does not yet support this compiler flag.

We investigated the outliers and identified several reasons for the performance achieved:

- **Device library differences.** There are significant differences in the device libraries: DPC++ uses the Intel Math Functions (IMF) Device Library (Intel Corporation 2024) whereas *chipStar* relies on a combination of LLVM built-ins, native OpenCL operations, a few custom implementations, and finally ROCm’s OCML bitcode implementations to provide

complete coverage. Furthermore, DPC++ defaulting to `-fp-model=fast` enables the use of `_native` intrinsics which are significantly faster (and more accurate) than non-native implementations.

- **Effect of JIT Flags.** *chipStar* sees significantly higher performance increases when JIT flag `-cl-fast-relaxed-math` is passed to the OpenCL runtime.
- **Floating point relaxations.** DPC++ defaults to `-fp-model=fast` which results in more aggressive optimizations by removing IEEE 754 guarantees and assuming no NaN/Inf values, generation of FMAs. In comparison, HIP allows only FMA contractions by default.
- **Value assumptions.** IR produced by DPC++ takes advantage of `AssumeTrueKHR` which is a SPIR-V instruction that tells the compiler to assume a given condition is always true, allowing it to optimize code more aggressively under that assumption. Using this instruction should give it a performance advantage over *chipStar*.
- **Intel-specific decorators.** DPC++ tends to produce IR which often contains Intel-specific operations that are meant to help with optimization when targeting Intel platforms which support the extensions, such as `OpAliasDomainDeclINTEL`, `OpAliasScopeDeclINTEL` whereas *chipStar* IR is more generic.

One thing to note is that the following results include only the benchmarks which passed correctness checks and in this regard *chipStar* ended up successfully running 6 fewer benchmarks compared to SYCL. A lot of these benchmarks have quite tight epsilon bounds so it’s likely that with small adjustments the number could be reduced.

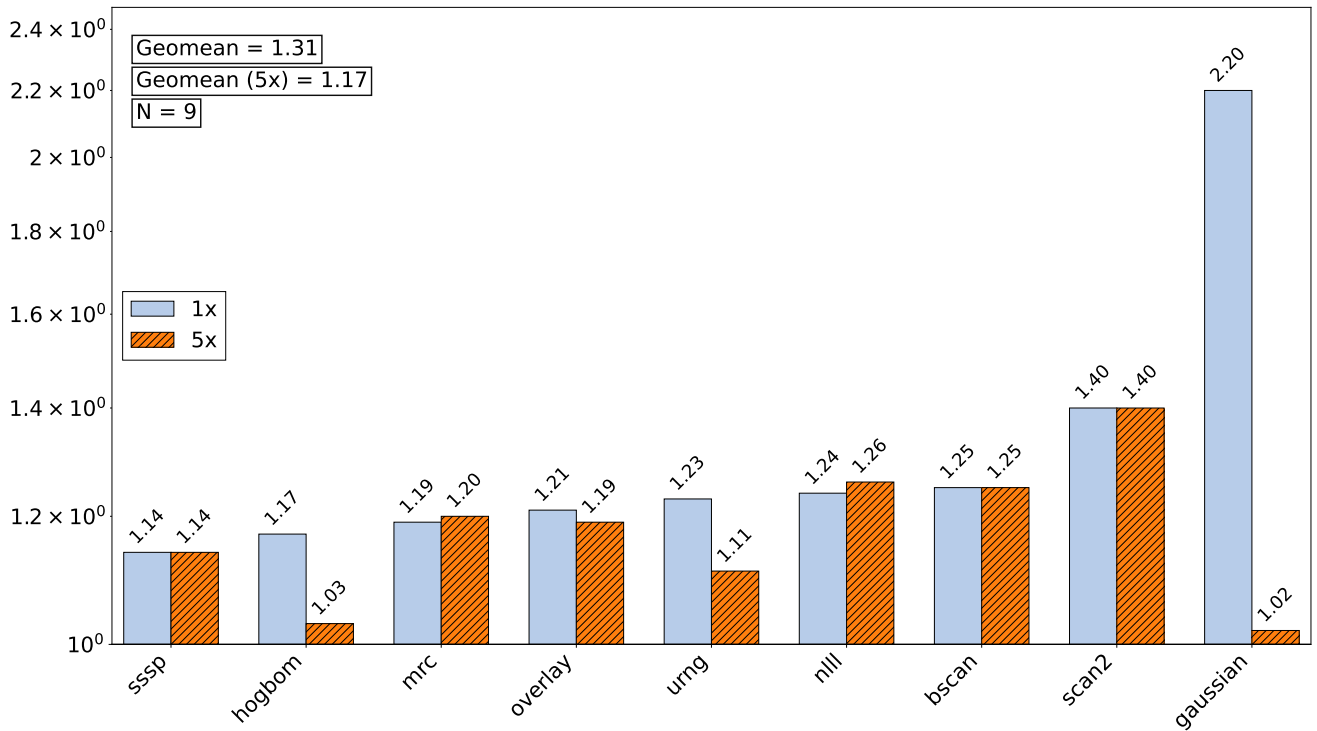


Figure 4. chipStar speedup over SYCL via DPC++ at 1x and 5x problem sizes.

In order to analyze the benefits of IMF, we modified the chipStar device library to link against IMF for exp and sqrt calls and tested the performance again which resulted in the adam benchmark going from 0.16x to 0.50x.

After we applied the `-cl-fast-relaxed math JIT` flag, all of these benchmarks performed equally well in *chipStar* and DPC++. In conclusion, all the cases where chipStar is slower than DPC++ can be explained by the differences in math device library and the more aggressive default optimization of the IR in terms of floating point calculations due to DPC++ defaulting to `-fp-model=fast`.

We selected 9 benchmarks where chipStar-v1.2.1 outperforms DPC++ for closer examination. Of these, most significant difference is seen in gaussian. Upon further examination, the time spent inside kernels is identical between these two runtimes so the most likely reason for the difference would be DPC++ overhead. This overhead cost could be comprised of either DPC++ startup costs, DPC++ kernel invocation costs or a combination of both. To test for this, we isolated the outperforming benchmarks and performed a scaling experiment by comparing performance on the original problem size to the performance achieved with the number of iterations or problem size increased by 5x (Fig. 4). This scaling leaves the runtime of a single kernel unchanged, which isolates the DPC++ startup overheads.

Scaling the number of iterations reduced the geomean chipStar-v1.2.1 speedup from 1.31x to 1.17x indicating that DPC++ has significantly higher startup costs compared to chipStar.

We analyzed the cases with the most dramatic differences and identified various explanations: Many of the benchmarks executed very short kernel commands, making the benchmark actually mostly measure the host API call execution speed. For example, the “overlay” benchmark could be sped

up significantly by switching off the profiling command queue feature. In some cases the device built-ins were more optimized in *chipStar* than in DPC++, in some cases it was the opposite. For example, when we compared the *chipStar* and DPC++ LLVM IRs of the device code for the “nlll” benchmark, we found that only *chipStar* performed the if-conversion optimization that converts some of the very small branches to conditional moves, which provided significant benefits.

In conclusion, *chipStar* has lower startup costs than DPC++ and either slightly outperforms or matches DPC++ performance across a variety of benchmarks though there are some exceptions in either direction.

CUDA on NVIDIA CUDA SDK

It is interesting to compare the performance of CUDA programs compiled and ran using the NVIDIA’s proprietary CUDA versus *chipStar* over an OpenCL runtime. In this experiment, we first compiled applications using the CUDA SDK 12.4 to get a baseline. The same benchmark cases (the CUDA versions) were then compiled using *chipStar* to the portable fat binary that uses OpenCL as the portability layer which was then run on rusticl/zink, an OpenCL implementation on top of NVidia’s proprietary Vulkan driver. The execution time when compiling using strict maths was measured on an NVIDIA RTX 3060 GPU with the results shown in Fig. 5. The geometrical of 1.01 tells that the overhead of the chipStar, the OpenCL API and the rusticl OpenCL runtimes is negligible on average. A few outlier cases vary to one direction or another.

HIP on AMD ROCm

Since *chipStar* can be viewed as a more portable implementation of HIP, it is interesting to compare its speed

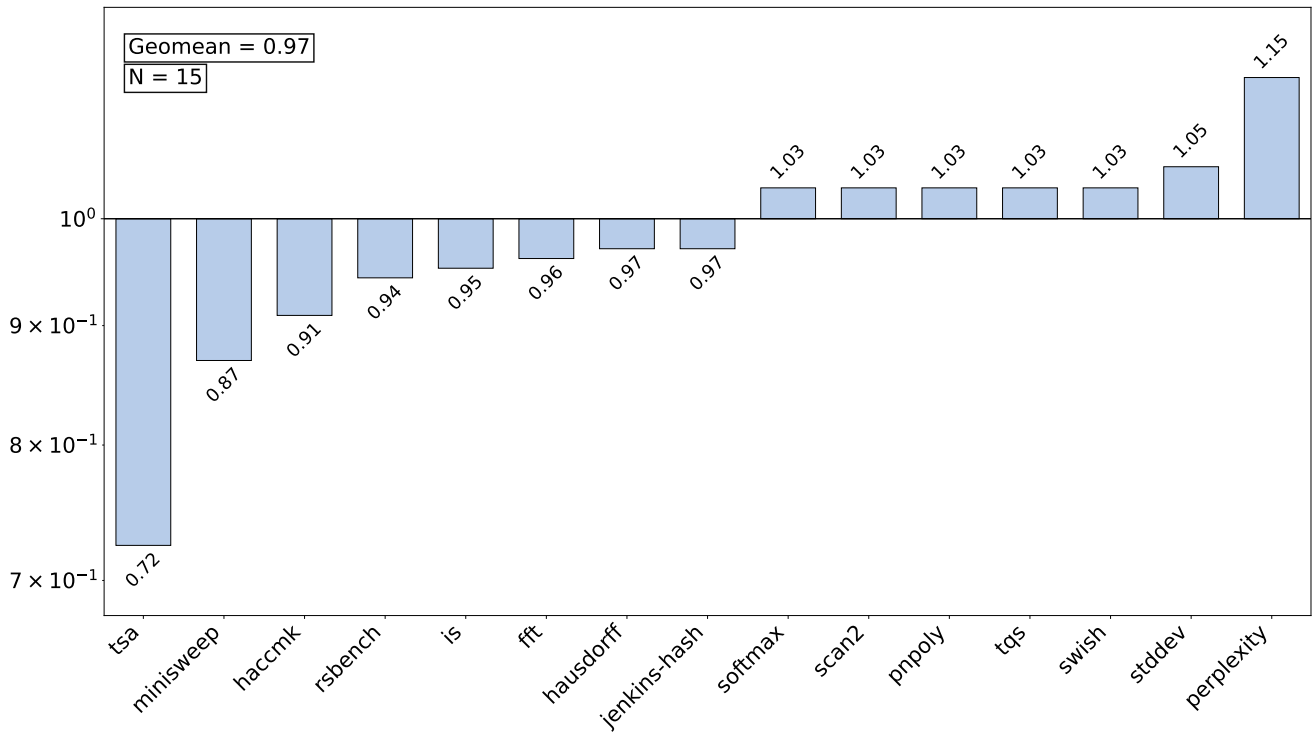


Figure 5. chipStar via rusticl/zink stack, speedup over CUDA SDK excluding 36 benchmarks where the performance difference was under 3%

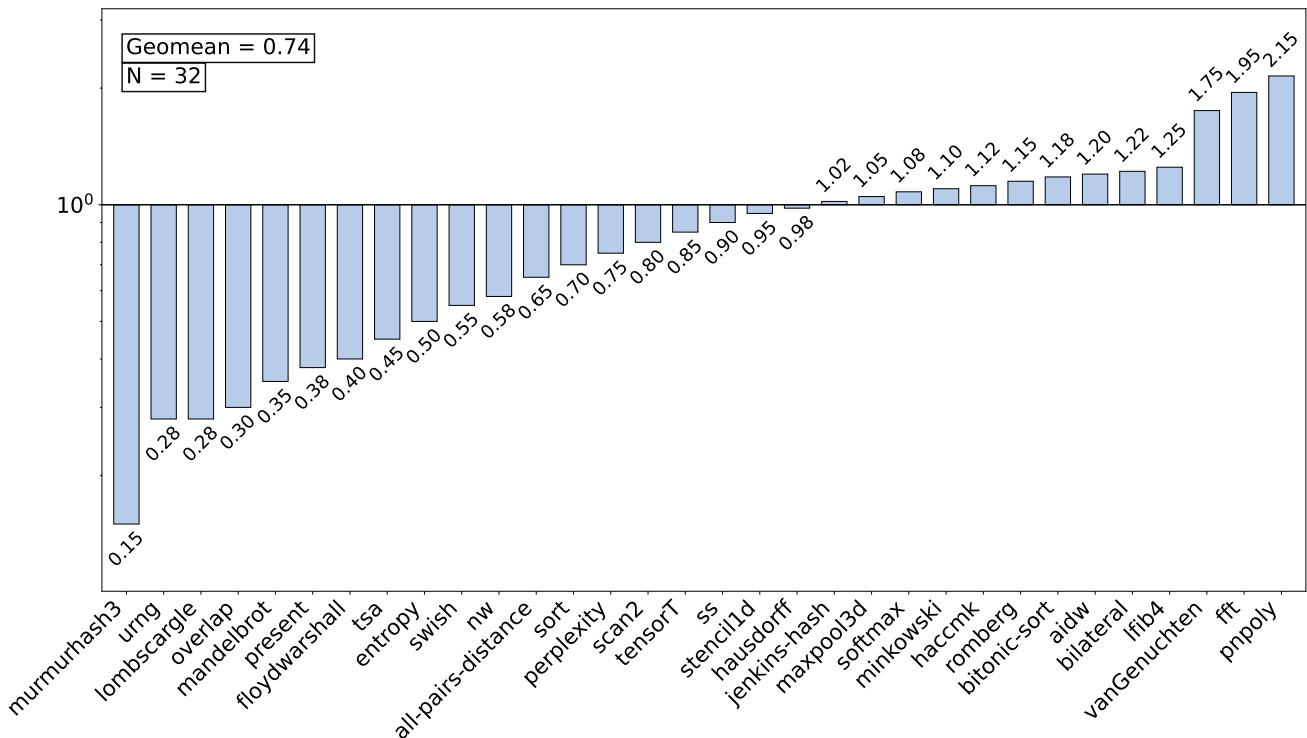


Figure 6. chipStar via rusticl/radeonsi stack speedup over ROCm. Running on AMD Radeon Pro VII.

against the original HIP implementation from AMD. For this comparison, we utilized the HIP compiler and runtime from the AMD ROCm package version 6.2.4 as a baseline to compile and execute a set of HeCbench HIP benchmarks on an AMD Radeon Pro VII GPU. To run the *chipStar* fat binaries on the same GPU, we used the rusticl OpenCL implementation on the radeonsi driver. AMD's OpenCL

implementation does not support SPIR-V input at the time of this writing, preventing its use in this comparison for running the *chipStar* binaries.

We have already demonstrated that the *chipStar* runtime does not introduce additional overheads in Fig. 3 so the following performance differences are a product of rusticl, not *chipStar*. Rusticl is still in the development phase and

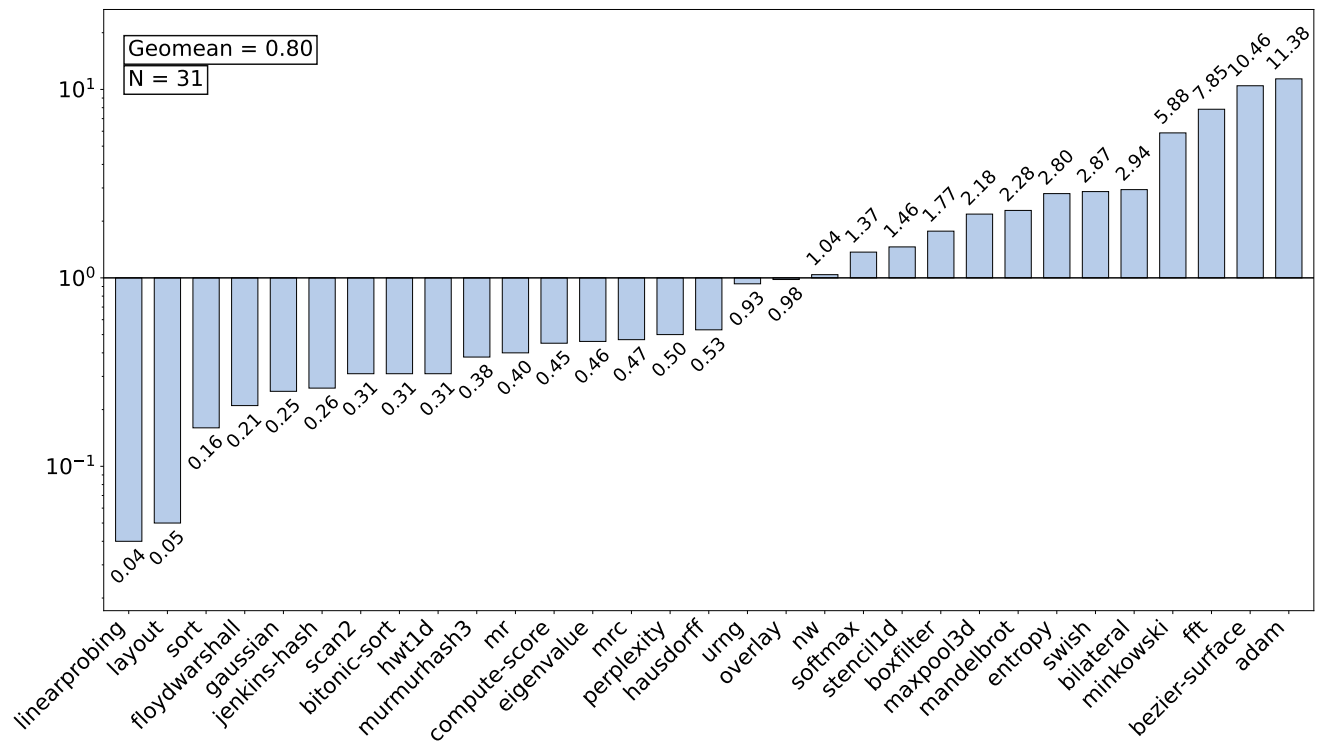


Figure 7. chipStar on PowerVR GPU, speedup over RISC-V CPU.

is not yet optimized for all targets, which explains the much larger performance differences in Fig. 6 compared to Fig. 3

Given the current state of rusticl, there is not much value in doing an extensive performance anomaly studies. However, one such anomaly is ‘pnpoly’ which is more than 2.5x faster on chipStar as can be seen in Fig. 6. The reason is that ROCm is slightly faster for tile sizes smaller than 32 and for larger ones notably slower. Unfortunately for ROCm, the benchmark tracks times for the largest tile which happens to be the slowest one on ROCm.

Portability Testing

In order to test the extent of portability of the runtime API layer based on the OpenCL standard, and to verify that offloading to a GPU via *chipStar* can bring speedups in comparison to running on the host CPU, we compiled and executed sets of HeCBench applications on various platforms which included both a CPU and a GPU with a capable enough OpenCL support to execute the same compiled *chipStar* fat binary on both devices. Given the experimental state of some of these platforms, the following performance numbers are not indicative of the true hardware potential and thus should be interpreted solely as indicators of application portability and not performance. The platforms and their results are presented below.

RISC-V CPU & PowerVR GPU: In this experiment we utilized the VisionFive2 single board computer for building and running the benchmarks. PoCL (Jääskeläinen et al. 2015) was used for running the benchmarks on the CPU and the proprietary OpenCL driver from Imagination Technologies was used for the GPU. The results are visualized in Fig. 7. The lower performance (0.74 geom mean) of the PowerVR GPU vs RISC-V CPU can be explained by the GPU having

much less on-chip resources than most benchmarks could utilize. The GPU also has a native workgroup size of only 32 (subgroup size 16), while most HeCBench benchmarks use a workgroup size ranging from 128 to 1024, leading to additional thread context switches. Due to not being able to force the subgroup size to match the warp width in this platform also prevented some of the benchmarks from running. Furthermore, the GPU’s limited local memory is used also to store images, samplers, the OpenCL constant data and pointers to global memory - in addition to the shared data of the application kernels (Imagination Technologies Limited 2023). The memory limitations and the lack of fp64 support were the main reason some of the test cases were not running at all on this platform.

ARM Cortex A53+A73 CPU & Mali G52 GPU For the CPU, PoCL (Jääskeläinen et al. 2015) was used as the OpenCL driver while the GPU was supported by the ARM’s proprietary OpenCL driver OpenCL C 3.0 v1.r40p0-01eac0.06c59e7df4d178b1ae2ad8082e91ad02. The results are shown in Fig. 8. A lot of applications were not able to run because of limited memory in the GPU and lack of double precision floating point support. However, various benchmarks showed significant benefits from CPU to GPU offloading, as expected.

HPC Application Case Study: GAMESS-GPU-HF

In order to further test *chipStar* in practice, we ported a complex HIP/CUDA-based HPC application and its dependency library using *chipStar*. The test environment for the experiment was the Aurora supercomputer utilizing Intel Datacenter Intel® Data Center GPU Max Series (referred to

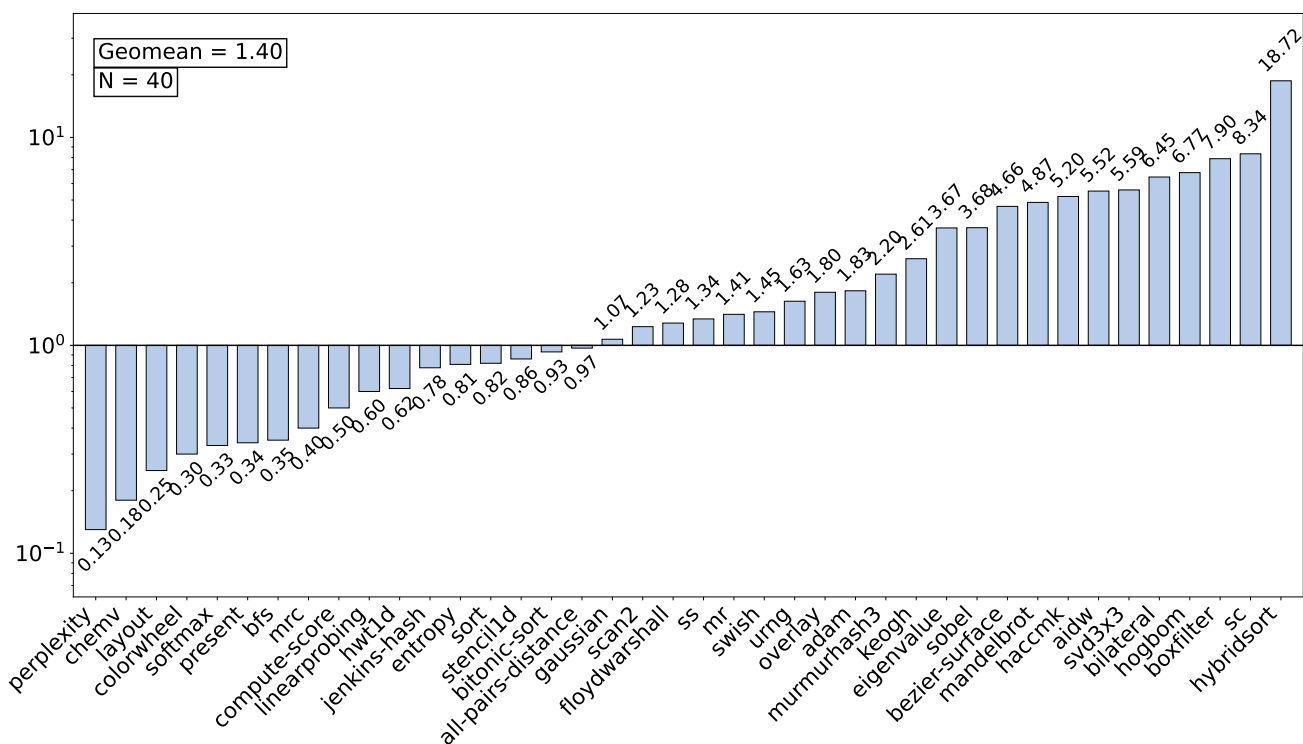


Figure 8. chipStar on ARM Mali GPU speedup over ARM Cortex CPU.

from here on as PVCs, as in Ponte Vecchio) as the accelerator part³.

GAMESS-GPU-HF

General Atomic and Molecular Electronic Structure System (GAMESS (Schmidt et al. 1993; Gordon and Schmidt 2005)) is a quantum chemistry software package which implements many electronic structure methods. The code base is primarily in Fortran 77/90 with some C/C++ and a CUDA library. Recently a new GPU version of the Hartree-Fock (HF) and RI-MP2 methods were implemented in CUDA which scales to 4096 nodes on Summit⁴, an Nvidia V100-based supercomputer (Barca et al. 2020, 2022). In this porting case we focused on the Hartree-Fock (HF) algorithm used by a CUDA library in GAMESS described in (Barca et al. 2020), which has been ported to HIP. The HF method is a common quantum chemistry method which is often the starting point for other higher-accuracy methods.

The HF method primarily involves the computation of N^4 two electron integrals (where N is a measure of molecular system size) as well as matrix contractions of the two electron integrals once they are formed, and computation of eigenvectors.

The two electron integrals are implemented as HIP/CUDA kernels which were optimized for Nvidia GPUs. The kernels total over 20,000 lines of HIP/CUDA code. The matrix contractions and eigensolves are done on the GPU via calls to the HIP math libraries hipBLAS and hipSOLVER.

Since the application uses ROCm software platform libraries hipBLAS and hipSOLVER, they needed to be ported as well. The required interfaces of these libraries were implemented for Intel hardware by using Intel oneMKL as a backend. This is done with two layers: first, a shim library, H4I-MKLShim⁵, which provides shims for the

SYCL-based oneMKL functions. This is designed to be used by other libraries which wish to call the oneMKL functions from a different API, such as hipBLAS, to use Intel GPUs. The chipStar project currently implements hipBLAS, hipSOLVER, and hipFFT in the libraries H4I-HipBLAS⁶, H4I-HipSOLVER⁷, and H4I-HipFFT⁸, respectively. These allow calls to hipBLAS, hipSOLVER, and hipFFT functions to run on Intel GPUs.

In terms of functionality, the HF code compiles and was verified to run correctly with *chipStar* on PVCs. The porting effort was relatively low, with one exception due to a small but significant specification difference in CUDA vs. OpenCL related to kernel thread synchronization: In CUDA group barriers are not counting in exited threads, meaning that there can be early returns from the kernel by a subset of the threads after which it is still legal to perform barrier synchronization with the remaining subset – the exited threads are just not counted in. In OpenCL this case is undefined behavior and in many implementations can lead to a deadlock. To tackle this gap, an OpenCL extension adding a group barrier with similar semantics would be needed (see *cl_ext_alive_only_barrier* in Table 3).

Performance The performance of the HF code was measured by compiling and running the same HIP source code on a PVC through *chipStar* (Release 1.2.1), an Nvidia A100 through CUDA 12.2.2 with ROCm 6.0.0, and an AMD MI250 through ROCm 6.3.0. To be clear, this is comparing *chipStar* on the PVC GPU to native HIP and CUDA on the Nvidia and AMD GPUs. To investigate the performance, a HF energy computation of a cluster of 150 water molecules with a STO-3G basis set was run 10 times on PVC, A100, and MI250. The average and standard deviation of the runtimes are displayed in Table 5.

	Average SCF Time (s) (Ratio over Nvidia A100)	Standard Deviation
Nvidia A100	1.66 (1x)	0.01
AMD MI250 (one GCD)	4.71 (2.8x)	0.01
Intel PVC (one Stack, OpenCL)	2.84 (1.7x)	0.03

Table 5. Timing comparison for SCF, Fock, and DIIS times (s) for GPU integral code across Intel, AMD, and Nvidia. Average over 10 runs.

Table 5 shows that the total HF energy calculation time (the SCF time) on the Nvidia A100 is shortest (1.66 s) and on one GCD of an AMD MI250 is the longest (4.71 s). The Intel PVC time, through *chipStar* with the OpenCL backend, is 2.8s, about 1.7x the runtime on the A100 GPU. From comparing the memory bandwidth and peak double-precision floating point operations possible on an A100 (Kwack et al. 2021) and a PVC stack (Applencourt et al. 2024), we see that we expect memory-bound codes on a PVC stack to take about 1.3x ($1.3 \frac{TB}{s} / 1.0 \frac{TB}{s}$) the time on an A100 and compute-bound codes on a PVC stack to take about 0.56x ($9.4 \frac{TFlop}{s} / 17.0 \frac{TFlop}{s}$) the time on an A100. Note that the 1.3x and 0.56x are upper bounds for performance since not all the runtime is on the GPUs, and the GPU library is a complex code with multiple asynchronous kernels and memory copies. A full performance analysis is out of the scope of this paper.

Although the Intel PVC time through *chipStar* is 1.7x slower than the A100 time, this is not too far from the 1.3x and 0.56x upper bound expectations based on hardware comparisons. The runtime on a PVC stack with *chipStar* is competitive with the runtime on other architectures and roughly within expectations based on the compute- and memory-peak comparisons of a PVC stack and an A100. Thus we expect HIP applications currently running on Nvidia and AMD GPUs to run and perform reasonably well with *chipStar* on Intel GPUs.

Related Work

The origin of *chipstar* is on the HIPCL (Babej and Jääskeläinen 2020) prototype which first tested the concept of compiling HIP programs to fat binaries relying on OpenCL and SPIR-V. The *chipStar* tool described in this article is a result of an almost a complete rewrite of the HIPCL code base and over approximately three years of continuous development work by multiple partners and HPC users. The HIPCL code base was initially forked to a separate code base to utilize the Level Zero (Intel Corporation 2025) low level API directly (HIPLZ (Zhao et al. 2022)) after which the OpenCL backend of HIPCL and the Level Zero backend of HIPLZ were merged to the same code base discussed in this article. A large number of missing essential features have been implemented since the initial prototypes were published. This article thus significantly expands upon the

original poster abstract that introduced the early-prototype-stage HIPCL and now presents a much more mature software stack usable for a wider range of real-world workloads.

When comparing *chipStar* to other HIP implementations, obviously the original ROCm, the AMD’s official GPU software platform (AMD, Inc 2023), is the baseline. ROCm consists of the general purpose programming API compilation and runtime support for HIP, and a set of libraries that support different degrees of compatibility with the CUDA platform. *chipStar* is not a new backend in addition to the AMD GPU and NVIDIA GPU backends provided by the AMD’s offering, but has an important technical difference: *chipStar* aims to offer runtime portability by its open standard based fat binary, removing the need to recompile the input software per target vendor platform, which is the case with ROCm.

SYCLomatic (Intel Corporation 2021) is a tool contributed by Intel Corporation for converting CUDA sources to the cross-vendor open standard SYCL (The Khronos® Group Inc 2014). Similar to AMD’s HIPify, but in contrast to *chipStar* which aims for source-level compatibility, SYCLomatic is a source-to-source conversion tool, which has its good and bad sides. The most apparent implication of relying on source-to-source conversion is more about maintenance aspects than technical ones; it necessitates the further development of the converted application to proceed using the SYCL API instead or in addition to CUDA. The main drawback is that in reality many code bases are difficult or impossible to convert solely to SYCL without having the CUDA version as a backup due to legacy, risk-management or technical reasons. The main benefit is that SYCL is an open standard, in contrast to CUDA, enabling more fair competition ground between hardware vendors. Thus, being able to target many platforms from a fat binary compiled from the unmodified CUDA/HIP source code base using a *chipStar*-style open platform approach can have its benefits. Furthermore, since *chipStar* is not a linkage-time or binary translation solution, but requires recompilation, it coincidentally also encourages utilizing and further developing the cross-vendor ecosystem APIs it relies upon. Furthermore, as of this writing SYCLomatic supports only CUDA, not HIP, while HIP has an increasing number of new applications implemented directly using it.

ZLUDA (Janik 2020) is a tool for running unmodified CUDA binaries on AMD GPUs. It works by reimplementing the CUDA driver API, and converting NVIDIA PTX (NVIDIA Corporation 2026b) to the vendor-specific IRs. Since it is a “drop-in solution” that works at program loading/linkage time, it can execute unmodified CUDA fat binaries, which is very comfortable to the end users as it doesn’t require access to the source code of the application. While we see ZLUDA as an excellent tool, it requires reverse engineering CUDA SDK’s binary interfaces and keeping up-to-date with the NVIDIA PTX as it evolves. We believe, in the longer term, especially as more of the missing extensions we describe are adopted by OpenCL implementations, *chipStar* can provide a more robust solution. In addition, ZLUDA also doesn’t support HIP as an input and now only targets AMD GPUs, whereas a key goal of *chipStar* is extensive cross-vendor portability.

MCUDA (Stratton et al. 2008) is the oldest tool we found for porting CUDA programs to non-NVIDIA platforms. MCUDA does source-to-source translation of kernels in a fashion that the translated kernels can execute efficiently on CPUs on a single CPU thread while respecting the barrier synchronization. In the case of *chipStar*, since it uses OpenCL as its portability layer, it can similarly target also vectorized CPU execution through CPU-targeting OpenCL implementations such as the Intel OpenCL CPU driver and PoCL’s CPU drivers (Jääskeläinen et al. 2015). Both of them are capable of vectorizing work-items (CUDA/HIP threads) inside work-groups, which translates to implicit autovectorization of CUDA/HIP kernels across CUDA threads and provide the benefits of CPU execution such as easier kernel debugging.

Swan (Harvey and De Fabritiis 2011) is another early source-to-source tool for CUDA porting. It generates OpenCL code from CUDA, providing similar level of portability as *chipStar* does. Another similar tool, CU2CL (Gardner et al. 2013) was published in the same year as (Harvey and De Fabritiis 2011). Neither Swan nor CU2CL are maintained any longer. In comparison to *chipStar*, the main technical differences to these tools are that *chipStar* utilizes the latest version of the OpenCL standard to support the newer CUDA/HIP features, uses SPIR-V as the intermediate language (no need to generate textual OpenCL C with its limitations) and it doesn’t suffer from problems related to source-to-source translations as *chipStar* provides source-level compatibility.

The closest comparable CUDA porting tool we could find is CUDA-on-CL (Perkins 2017). Like *chipStar*, it similarly compiles CUDA programs using Clang/LLVM-based compiler chain to binaries which then execute on OpenCL platforms. However, similarly to Swan and CU2L, it compiles device kernels to OpenCL C whereas *chipStar* uses SPIR-V as the portable binary format. Other technical differences in *chipStar* are related to the use of modern OpenCL standard features to implement some of the features of CUDA. These include using SVM to implement raw pointers and implementing warp-level primitives such as shuffles using the subgroup features.

Conclusions

In this article, we presented *chipStar*, a compilation flow and a runtime for CUDA/HIP applications using open cross-vendor supported standards. In comparison to previous tools, *chipStar*’s goal is on source-level compatibility which we believe has longer-term robustness benefits in comparison to binary translation. It relies on open standards at the portability layer level which in our opinion has significant inherent far-reaching value.

The performance of HIP benchmarks using *chipStar* was shown to be on par or surpass their SYCL/DPC++ versions. We’ve also shown that *chipStar* is very competitive against CUDA running on Nvidia GPU with comparisons to HIP and AMD hardware being less favorable due to OpenCL driver options on AMD hardware.

An example of the source-level compatibility was provided with GAMESS-GPU-HF, a code base with a significant number of kernel code lines. This demonstrates

that *chipStar* is a useful option for applications that are not feasible to port to more cross-vendor supported open standard input APIs such as SYCL or OpenMP.

In the future, we will focus on expanding the HIP/CUDA feature coverage such as Graphs API and collaborative workgroups in *chipStar*. We will also expand the set of supported core libraries (hipFFT, hipSPARSE, etc) in the CUDA and HIP ecosystems to enable more real-world applications.

Acronyms

CG Coarse-grained buffer. 4

DPC++ Data Parallel C++. 3, 7–10, 15

HSA Heterogeneous System Architecture. 2, 3

HSAIL Heterogeneous system architecture intermediate language. 3

IR Intermediate Representation. 2, 3, 5, 9, 10, 14

ISA Instruction Set Architecture. 2

JIT just-in-time. 2, 5, 6, 9, 10

SPIR Standard Portable Intermediate Representation. 3

SPIR-V new version of SPIR. 2–7, 9, 11, 14, 15

SSA Static Single Assignment. 3

SVM Shared Virtual Memory. 3, 4, 7, 15

USM Unified Shared Memory. 4, 7

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

This manuscript has been coauthored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>).

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which

is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility.

Tampere University authors' contributions were in part funded via the DARE SGA1 Project, which is an European High-Performance Computing Joint Undertaking (JU) under Grant Agreement No. 101202459. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, Czechia, Italy, Netherlands, Belgium, Finland, Greece, Croatia, Portugal, Poland, Sweden, France and Austria.

Notes

1. See Section Acronyms for an Acronym Table.
2. <http://clang.llvm.org/>
3. <https://www.alcf.anl.gov/aurora>
4. <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>
5. <https://github.com/CHIP-SPV/H4I-MKLShim>
6. <https://github.com/CHIP-SPV/H4I-HipBLAS>
7. <https://github.com/CHIP-SPV/H4I-HipSOLVER>
8. <https://github.com/CHIP-SPV/H4I-HipFFT>

References

- Alpay A and Heuveline V (2023) 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, IWOCCL '23*. New York, NY, USA: Association for Computing Machinery. ISBN 9798400707452. DOI:10.1145/3585341.3585351.
- Alpay A, Soproni B, Wünsche H and Heuveline V (2022) Exploring the possibility of a hipsycl-based implementation of oneapi. In: *International Workshop on OpenCL, IWOCCL'22*. New York, NY, USA: Association for Computing Machinery. ISBN 9781450396585. DOI:10.1145/3529538.3530005.
- AMD, Inc (2023) ROCm documentation - release 4.5.0. https://rocm.docs.amd.com/_downloads/en/latest/pdf/.
- AMD, Inc (2025a) HIP documentation. <https://rocm.docs.amd.com/projects/HIP/en/latest/index.html>.
- AMD, Inc (2025b) Hip math function precision guarantees. https://rocm.docs.amd.com/projects/HIP/en/latest/reference/math_api.html.
- Applencourt T, Sadawarte A, Muralidharan S, Bertoni C, Kwack J, Luo Y, Rangel E, Tramm J, Ghadar Y, Tamerus A et al. (2024) Ponte vecchio across the atlantic: Single-node benchmarking of two intel gpu systems. In: *IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*.
- Babej M and Jääskeläinen P (2020) HIPCL: Tool for porting CUDA applications to advanced OpenCL platforms through HIP. In: *Proceedings of the International Workshop on OpenCL, IWOCCL '20*. New York, NY, USA: Association for Computing Machinery. ISBN 9781450375313. DOI:10.1145/3388333.3388641. URL <https://doi.org/10.1145/3388333.3388641>.
- Barca GJ, Snowdon C, Vallejo J, Kazemian F, Rendell AP and Gordon MS (2022) Scaling correlated fragment molecular orbital calculations on summit. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–14. DOI:10.1109/SC41404.2022.00012. URL <https://doi.ieeecomputersociety.org/10.1109/SC41404.2022.00012>.
- Barca GMJ, Poole DL, Vallejo JLG, Alkan M, Bertoni C, Rendell AP and Gordon MS (2020) Scaling the hartree-fock matrix build on summit. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. pp. 1–14. DOI:10.1109/SC41405.2020.00085.
- Cytron R, Ferrante J, Rosen BK, Wegman MN and Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13(4): 451–490. DOI:10.1145/115372.115320. URL <https://doi.org/10.1145/115372.115320>.
- Doerfert J, Jasper M, Huber J, Abdelaal K, Georgakoudis G, Scogland T and Parasyris K (2023) Breaking the vendor lock: Performance portable programming through openmp as target independent runtime layer. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT '22*. New York, NY, USA: Association for Computing Machinery. ISBN 9781450398688, p. 494–504. DOI:10.1145/3559009.3569687. URL <https://doi.org/10.1145/3559009.3569687>.
- Gardner M, Sathre P, Feng Wc and Martinez G (2013) Characterizing the Challenges and Evaluating the Efficacy of a CUDA-to-OpenCL Translator. *Parallel Computing* 39(12).
- Gordon MS and Schmidt MW (2005) Chapter 41 - advances in electronic structure theory: Gamess a decade later. In: Dykstra CE, Frenking G, Kim KS and Scuseria GE (eds.) *Theory and Applications of Computational Chemistry*. Amsterdam: Elsevier, pp. 1167 – 1189.
- Harvey M and De Fabritiis G (2011) Swan: A tool for porting cuda programs to opencl. *Computer Physics Communications* 182(4): 1093–1099. DOI:<https://doi.org/10.1016/j.cpc.2010.12.052>. URL <https://www.sciencedirect.com/science/article/pii/S0010465511000117>.
- Heterogeneous System Architecture Foundation (2016) HSA runtime programmer's reference manual v1.1.1. <http://hsafoundation.com/wp-content/uploads/2021/02/HSA-Runtime-1.1.1.pdf> (accessed Apr. 13, 2023).
- Heterogeneous System Architecture Foundation (2018) HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG). <http://hsafoundation.com/wp-content/uploads/2021/02/HSA-PRM-1.2.pdf> (accessed Apr. 12, 2023).
- Heterogeneous System Architecture Foundation (2021) HSA platform system architecture specification. <http://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf> (accessed Apr. 13, 2023).
- Imagination Technologies Limited (2023) Introduction to the PowerVR Compute Development Recommendations. <https://docs.imgtec.com/performance-guides/compute-recommendations/html/>

- topics/introduction/introduction.html.
- Intel Corporation (2021) SYCLomatic: A New CUDA*-to-SYCL* Code Migration Tool. <https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html>. (accessed Feb. 3, 2026).
- Intel Corporation (2023) Intel Floating Point Model. <https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-1/fp-model-fp.html>.
- Intel Corporation (2024) Imf device library. <https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/imf-device-library.html>.
- Intel Corporation (2025) oneapi level zero: 1.15.31. <https://oneapi-src.github.io/level-zero-spec/level-zero/latest/index.html#>.
- Intel Corporation (2026) Unified Shared Memory. https://registry.khronos.org/OpenCL/extensions/intel/cl_intel_unified_shared_memory.html.
- Jääskeläinen P, de La Lama CS, Schnetter E, Raiskila K, Takala J and Berg H (2015) pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43(5): 752–785. DOI:10.1007/s10766-014-0320-y. URL <http://dx.doi.org/10.1007/s10766-014-0320-y>.
- Janik A (2020) ZLUDA: CUDA on non-NVIDIA GPUs. <https://github.com/vosen/ZLUDA>.
- Jin Z and Vetter JS (2023) A benchmark suite for improving performance portability of the sycl programming model. In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. pp. 325–327. DOI:10.1109/ISPASS57527.2023.00041.
- John Kessenich and Boaz Ouriel and Raun Krisch (2023) SPIR-V Specification. <https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.pdf> (accessed Apr. 12, 2023).
- Khronos® OpenCL Working Group (2025a) Buffer Device Address. https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/cl_ext_buffer_device_address.html.
- Khronos® OpenCL Working Group (2025b) Opencil math function precision guarantees. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html#relative-error-as-ulps.
- Khronos® OpenCL Working Group (2025c) The OpenCL® Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.
- Kwack J, Tramm J, Bertoni C, Ghadar Y, Homerding B, Rangel E, Knight C and Parker S (2021) Evaluation of performance portability of applications and mini-apps across amd, intel and nvidia gpus. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. pp. 45–56. DOI:10.1109/P3HPC54578.2021.00008.
- Lattner C and Adve V (2004) LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.
- NVIDIA Corporation (2018a) Cuda independent thread scheduling. https://docs.nvidia.com/cuda/archive/9.0/cuda-c-programming-guide/?utm_source=chatgpt.com#independent-thread-scheduling-7-x.
- NVIDIA Corporation (2018b) Cuda math function precision guarantees. <https://docs.nvidia.com/cuda/archive/9.0/cuda-c-programming-guide/index.html#mathematical-functions-appendix>.
- NVIDIA Corporation (2026a) CUDA NVCC Fast Math. https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.
- NVIDIA Corporation (2026b) Parallel thread execution isa version 9.1. <https://docs.nvidia.com/cuda/parallel-thread-execution/>.
- OpenMP Architecture Review Board (2021) OpenMP Application Programming Interface v5.2. <https://www.openmp.org/>.
- Perkins H (2017) Cuda-on-cl: A compiler and runtime for running nvidia® cuda™ c++11 applications on opencil™ 1.2 devices. In: *Proceedings of the 5th International Workshop on OpenCL, IWOCCL 2017*. New York, NY, USA: Association for Computing Machinery. ISBN 9781450352147. DOI: 10.1145/3078155.3078156. URL <https://doi.org/10.1145/3078155.3078156>.
- Rydahl A, Huber J, McDonough EL and Doerfert J (2023) Precision and performance analysis of C standard math library functions on GPUs. *Proceedings of the ACM on Programming Languages* 7(OOPSLA2): 1–26. DOI:10.1145/3624062.3624166.
- Schmidt MW, Baldrige KK, Boatz JA, Elbert ST, Gordon MS, Jensen JH, Koseki S, Matsunaga N, Nguyen KA, Su S, Windus TL, Dupuis M and Montgomery JA (1993) General atomic and molecular electronic structure system. *Journal of Computational Chemistry* 14(11): 1347–1363.
- Stratton J, Stone S and Hwu W (2008) Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In: *Languages and Compilers for Parallel Computing - 21st International Workshop, LCPC 2008, Revised Selected Papers*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). ISBN 3540897399, pp. 16–30. DOI: 10.1007/978-3-540-89740-8_2. 21st International Workshop on Languages and Compilers for Parallel Computing, LCPC 2008 ; Conference date: 31-07-2008 Through 02-08-2008.
- The Khronos Group Inc (2014) Standard Portable Intermediate Representation Specification. https://registry.khronos.org/SPIR/specs/spir_spec-2.0.pdf (accessed Apr. 12, 2023).
- The Khronos® Group Inc (2014) Khronos Releases SYCL 1.2 Provisional Specification. <https://www.khronos.org/news/press/khronos-releases-sycl-1.2-provisional-specification>. (accessed Apr. 4, 2023).
- The Khronos® Group Inc (2019) LLVM/SPIR-V Bi-Directional Translator. <https://github.com/KhronosGroup/SPIRV-LLVM-Translator>.
- The Khronos® Vulkan Working Group (2023) Vulkan® 1.3.246 - A Specification (with all registered Vulkan extensions). <https://www.khronos.org/specs/spec-ids/>

[//registry.khronos.org/vulkan/specs/1.3-extensions/pdf/vkspec.pdf](https://registry.khronos.org/vulkan/specs/1.3-extensions/pdf/vkspec.pdf) (accessed Apr. 12, 2023).

Zhao J, Bertoni C, Young J, Harms K, Sarkar V and Videau B (2022) HIPLZ: Enabling performance portability for exascale systems. In: *Proceedings of the HeteroPar 2022: 20th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*. Springer.