# Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview

Andreas Herten
Forschungszentrum Jülich
Jülich Supercomputing Centre
Jülich, Germany
a.herten@fz-juelich.de

## ABSTRACT

In recent history, GPUs became a key driver of compute performance in HPC. With the installation of the Frontier supercomputer, they became the enablers of the Exascale era; further largest-scale installations are in progress (Aurora, El Capitan, JUPITER). But the early-day dominance by NVIDIA and their CUDA programming model has changed: The current HPC GPU landscape features three vendors (AMD, Intel, NVIDIA), each with native and derived programming models. The choices are ample, but not all models are supported on all platforms, especially if support for Fortran is needed; in addition, some restrictions might apply. It is hard for scientific programmers to navigate this abundance of choices and limits. This paper gives a guide by matching the GPU platforms with supported programming models, presented in a concise table and further elaborated in detailed comments. An assessment is made regarding the level of support of a model on a platform.

## KEYWORDS

GPU, GPGPU, Programming Models, HPC, AMD, Intel, NVIDIA, CUDA, HIP, SYCL

**ACM Reference Format:**
Andreas Herten. 2023. Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3624178

## 1 INTRODUCTION

Taking the TOP500 list of June 2023 as a reference [10], more than 60 % of the available FLOP/s are delivered by Graphics Processing Units (GPUs). The devices were first installed in HPC systems in the mid 2000s and steadily matured over the next decades. The most-recent culmination came in 2022, when the first Exascale supercomputer, Frontier at Oak Ridge National Lab, was added to the TOP500 list, utilizing more than 37 000 GPUs to deliver 1194 PFLOP/s (*Rmax*) of compute performance – alone delivering about 20 % of the entire list's compute performance. Further largest-scale installations using GPUs are planned or already on the way, like Aurora (at Argonne National Lab), El Capitan (at Lawrence Livermore National Lab), or JUPITER (at Jülich Supercomputing Centre).

While the first years of GPU usage in HPC was dominated by NVIDIA GPUs and NVIDIA's CUDA programming model, the landscape significantly changed in the last years. Frontier utilizes AMD GPUs (37 888× AMD Radeon Instinct MI250X) and Aurora uses Intel GPUs (63 744× Intel Data Center GPU Max Series, codename *Ponte Vecchio*); also El Capitan will use next-generation AMD GPUs (AMD Radeon Instinct MI300A). Each GPU platform has a selected major native programming model: CUDA for NVIDIA, HIP for AMD, and SYCL for Intel[1]. They are augmented with further vendor- or community-driven models, usually presenting higher-level abstractions. Examples are OpenMP and OpenACC as the two major directive-based models; Kokkos, RAJA, and Alpaka which enable GPU programming through high-level abstractions for parallel algorithms and data management; and Standard-based parallelism which utilizes modern features of programming languages themselves to access GPUs. The key scientific programming language is C++ (sometimes programmed in a plain C-style), but also Fortran is still prevalent in many scientific applications. Also Python has become a popular choice in recent years [8, 54]; as an even higher-level, interpreted programming language it relies on *backends* in lower-level languages – mostly C/C++ – and rather implements interfaces.

Although the evolving combinatorial explosion of choices[2] is a good sign for the health of the GPU ecosystem, the field can at times be hard to navigate – for established GPU developers but especially for novice users. With the selection made in this paper, more than 50 routes for programming a GPU device are identified when no further limitations (pre-)exist. This work gives a guide into the current GPU programming ecosystem, by categorizing the individual possibilities in a concise table and explaining each combination in detail.

The main contributions of this paper are the categories of rating support of programming models on GPU devices in section 3, the application in the overview table in Figure 1, and the accompanying list of explanations in section 4, with many links to further resources.

The paper is structured as follows: In section 3, the six rating categories are explained in detail and some comments to the method are made. In section 4, the core of this paper, the overview table (Figure 1), is presented and explained with detailed comments for

*SC-W 2023, November 12–17, 2023, Denver, CO, USA*


[1] Intel bundles their parallel programming infrastructure into *oneAPI*, which includes – amongst others – DPC++, their SYCL implementation. Next to SYCL, also OpenMP is a prominently promoted programming model by Intel.
[2] GPU platforms × programming models × programming languages

each possible choice. Section 5 shows limitations and caveats of the table and methodology. Finally, section 6 concludes this paper.

## 2 RELATED WORK

While no other work is known outlining and assessing the usage of programming models on certain GPU devices to the extent presented here, related work previously compared specific aspects or sub-sets, usually with a focus on performance. Hammond et al. [19] compared performance for standard language parallelism in Fortran, by using the BabelStream benchmarks. Markomanolis et al. [40] cast a wider net around applications, but focused solely on the LUMI supercomputer. Hammond evaluated several NVIDIA-compatible GPU programming models in [18]. A very detailed comparison of GPU support through various OpenMP-capable compilers was given at the 2022 ECP Community BOF Days [50]. Deeper, more technical insights can be gained by dedicated validation suites [24, 34]. Further examples are discussed in section 4.

## 3 METHOD, CATEGORIES

Figure 1 matches the three GPU vendors AMD, Intel, and NVIDIA (row) with programming models (columns). Each column is additionally separated into two sub-columns for the two programming language C++ and Fortran. The presented programming models are the three *native* models (CUDA, HIP, SYCL), the two major directive-based models (OpenMP, OpenACC), two examples of community-driven, higher-level models with foci on platform-portability (Kokkos, Alpaka), and the upcoming GPU usage through standard features in the programming language (*Standard*). In addition, support by Python is summarized for each platform. In total, 51 possible combinations are explored and explained in 44 unique descriptions.

No restrictions are exposed regarding language version of C++ and Fortran, as it would add another level of complexity and is usually not a limiting factor for scientists due to backward compatibility. While C++ is required by most programming models, some models can also be used in a C-like manner. For the sake of brevity, this paper considers C++.

To assess and describe the 51 possible combinations, a review of available literature and online resources was conducted. The available information and its level of detail and coverage varies significantly between the models; it is most extensive for the native models and sparsest for some community-driven, explorative implementations. The combinations are assessed by this available information and – to a limited extend – the experience of the author. The paper strives to be objective and derives the rating with thorough descriptions. Of course, classifying into six distinct categories has some limitations, outlined within the descriptions themselves and discussed further in section 5.

This work introduces six categories to assess the coverage of a certain combination of programming model and language on a respective GPU platform. The categories are indicated by symbols, reaching from ● (full support) to ⁄ (no support), with various intermediate steps. The following list explains the categorizing symbols and also names the categories for completeness.

- ● The programming model for this language is fully supported on this GPU platform by the vendor. The vendor provides complete implementation of the combination and extensive documentation. The model is regularly updated and the vendor provides support in case of errors. *Category Name:* ***full support***
- ◒ The combination of programming model and language is indirectly, but comprehensively supported by the vendor of the GPU device. This is usually achieved by (semi-)automatically mapping/translating a *foreign* model to a *native* one. *Category Name:* ***indirect good support***
- ■ Model/language are supported on this GPU device by the vendor, but the support is not (yet) comprehensive. Usually, the model can be used for the majority of applications, but some specific features are not available. *Category Name:* ***some support***
- ▲ Comprehensive support is available for this combination of programming model and programming language on a GPU device, but not by the vendor of the GPU device itself. Usually, higher-level models driven by the community implement support and utilize vendor-native infrastructure in the background, unexposed to the user. *Category Name:* ***non-vendor good support***
- ★ Some very limited support is available for this programming model and language on a certain GPU device. The support might be indirect, through extensive effort by the user, and/or very incomplete. *Category Name:* ***limited support***
- ⁄ No direct support is available for the model/language on the device. There are certainly ways to still utilize the device, like creating custom headers and linking to libraries directly, or utilizing `ISO_C_BINDING` in Fortran. *Category Name:* ***no support***

The following section 4 lists the descriptions of each possible combination, referring the categorizing symbols of Figure 1 with reference numbers[3]. In each description, links to online resources are overlaid as hyperlinks. The key references for an item are included as entries in the bibliography. The three native programming models (CUDA, HIP, SYCL) are explained in greatest details for their main platform (NVIDIA, AMD, Intel GPU devices, respectively). At times, descriptions for entries are similar. This is by choice due to the encyclopedic nature of the document in which readers might look up only single entries.

## 4 DESCRIPTIONS

1 **NVIDIA • CUDA • C++:** CUDA C/C++ is supported on NVIDIA GPUs through the CUDA Toolkit. First released in 2007, the toolkit covers nearly all aspects of the NVIDIA platform: an API for programming (incl. language extensions), libraries, tools for profiling and debugging, compiler, management tools, and more. The current version is CUDA 12.2. Usually, when referring to *CUDA* without any additional context, the CUDA API is meant. While incorporating some Open Source components, the CUDA platform in its entirety is proprietary and closed sourced. The low-level CUDA instruction set architecture is PTX, to which higher languages like the CUDA C/C++ are translated to. PTX is compiled to SASS, the binary code executed on the device. As it is the reference for platform, the

[3] In the PDF version of this document, both number can be clicked and move between table and description.

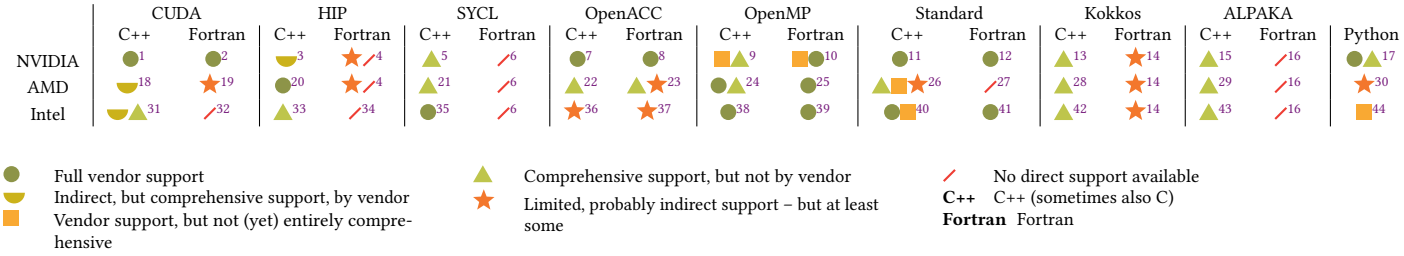


**Figure 1: Overview table comparing a selection of major GPU programming models with the current state of support by the three vendors of dedicated HPC GPUs (AMD, Intel, NVIDIA) for C++ and Fortran. See section 3 for more detailed explanations of the categories.**

support for NVIDIA GPUs through CUDA C/C++ is very comprehensive. In addition to support through the CUDA toolkit, NVIDIA GPUs can also be used by Clang, utilizing the LLVM toolchain to emit PTX code and compile it subsequently. [45]

2 **NVIDIA • CUDA • Fortran:** CUDA Fortran, a proprietary Fortran extension by NVIDIA, is supported on NVIDIA GPUs via the NVIDIA HPC SDK (*NVHPC*). NVHPC implements most features of the CUDA API in Fortran and is activated through the `-cuda` switch in the `nvfortran` compiler. The CUDA extensions for Fortran are modeled closely after the CUDA C/C++ definitions. In addition to creating explicit kernels in Fortran, CUDA Fortran also supports *cuf kernels*, a way to let the compiler generate GPU parallel code automatically. Very recently, CUDA Fortran support was also merged into Flang, the LLVM-based Fortran compiler. [43]

3 **NVIDIA • HIP • C++:** HIP programs can directly use NVIDIA GPUs via a CUDA backend. As HIP is strongly inspired by CUDA, the mapping is relatively straight-forward; API calls are named similarly (for example: `hipMalloc()` instead of `cudaMalloc()`) and keywords of the kernel syntax are identical. HIP also supports some CUDA libraries and creates interfaces to them (like `hipblasSaxpy()` instead of `cublasSaxpy()`). To target NVIDIA GPUs through the HIP compiler (`hipcc`), `HIP_PLATFORM=nvidia` needs to be set in the environment. In order to initially create a HIP code from CUDA, AMD offers the HIPIFY conversion tool. [4]

4 **NVIDIA, AMD • HIP • Fortran:** No Fortran version of HIP exists; HIP is solely a C/C++ model. But AMD offers an extensive set of ready-made interfaces to the HIP API and HIP and ROCm libraries with hipfort (MIT-licensed). All interfaces implement C functionality and CUDA-like Fortran extensions, for example to write kernels, are available. [5]

5 **NVIDIA • SYCL • C++:** No direct support for SYCL is available by NVIDIA, but SYCL can be used on NVIDIA GPUs through multiple venues. First, SYCL can be used through DPC++, an Open-Source LLVM-based compiler project led by Intel. The DPC++ infrastructure is also available through Intel's commercial oneAPI toolkit (*Intel oneAPI DPC++/C++*) as a dedicated plugin. Upstreaming SYCL support directly into LLVM is an ongoing effort, which started in 2019. Further, SYCL can be used via Open SYCL (previously called hipSYCL), an independently developed SYCL implementation, using NVIDIA GPUs either through the CUDA support of LLVM or the `nvc++` compiler of NVHPC. A third popular possibility was the NVIDIA GPU support in ComputeCpp of CodePlay; though the product became unsupported in September 2023. In case LLVM is involved, SYCL implementations can rely on CUDA support in LLVM, which needs the CUDA toolkit available for the final compilations parts beyond PTX. In order to translate a CUDA code to SYCL, Intel offers the SYCLomatic conversion tool. [1, 32]

6 **NVIDIA, AMD, Intel • SYCL • Fortran:** SYCL is a C++-based programming model (C++17) and by its nature does not support Fortran. Also, no pre-made bindings are available. [17]

7 **NVIDIA • OpenACC • C++:** OpenACC C/C++ on NVIDIA GPUs is supported most extensively through the NVIDIA HPC SDK. Beyond the bundled libraries, frameworks, and other models, the NVIDIA HPC SDK also features the `nvc/nvc++` compilers, in which OpenACC support can be enabled with the `-acc -gpu`. The support of OpenACC in this vendor-delivered compiler is very comprehensive, it conforms to version 2.7 of the specification. A variety of compile options are available to modify the compilation process. In addition to NVIDIA HPC SDK, good support is also available in GCC since GCC 5.0, supporting OpenACC 2.6 through the `nvptx` architecture. The compiler switch to enable OpenACC in `gcc/g++` is `-fopenacc`, further options are available. Further, the Clacc compiler implements OpenACC support into the LLVM toolchain, adapting the Clang frontend. As a central design aspect, it translates OpenACC to OpenMP as part of the compilation process. OpenACC can be activated in a Clacc-`clang` via `-fopenacc`, and further compiler options exist, mostly leveraging OpenMP options. A recent study by Jarmusch et al. compared these compilers for coverage of the OpenACC 3.0 specification. [13, 16, 34, 47]

8 **NVIDIA • OpenACC • Fortran:** Support of OpenACC Fortran on NVIDIA GPUs is similar to OpenACC C/C++, albeit not identical. First, NVIDIA HPC SDK supports OpenACC in Fortran through the included `nvfortran` compiler, with options like for the C/C++ compilers. In addition, also GCC supports OpenACC through the `gfortran` compiler with identical compiler options to the C/C++ compilers. Further, similar to OpenACC support in LLVM for C/C++ through *Clacc* contributions, the LLVM frontend for Fortran, Flang (the successor of *F18*, not *classic Flang*), supports OpenACC as well. Support was initially contributed through the Flacc project and now resides in the main LLVM project. Finally, the HPE Cray Programming Environment supports OpenACC Fortran; in `ftn-hacc`. [9, 16, 47]

9 **NVIDIA • OpenMP • C++:** OpenMP in C/C++ is supported on NVIDIA GPUs (*Offloading*) through multiple venues, similarly

to OpenACC. First, the NVIDIA HPC SDK supports OpenMP GPU offloading in both `nvc` and `nvc++`, albeit only a subset of the entire OpenMP 5.0 standard (see the documentation for supported/unsupported features). The key compiler option is `-mp`. Also in GCC, OpenMP offloading can be used to NVIDIA GPUs; the compiler switch is `-fopenmp`, with options delivered through `-foffload` and `-foffload-options`. GCC currently supports OpenMP 4.5 entirely, while OpenMP features of 5.0, 5.1, and, 5.2 are currently being implemented. Similarly in Clang, where OpenMP offloading to NVIDIA GPUs is supported and enabled through `-fopenmp -fopenmp-targets=nvptx64`, with offload architectures selected via `--offload-arch=native` (or similar). Clang implements nearly all OpenMP 5.0 features and most of OpenMP 5.1/5.2. In the HPE Cray Programming Environment, a subset of OpenMP 5.0/5.1 is supported for NVIDIA GPUs. It can be activated through `-fopenmp`. Also AOMP, AMD's Clang/LLVM-based compiler, supports NVIDIA GPUs. Support of OpenMP features in the compilers was recently discussed in the OpenMP ECP BoF 2022. [14, 15, 23, 47]

10 **NVIDIA • OpenMP • Fortran:** OpenMP in Fortran is supported on NVIDIA GPUs nearly identical to C/C++. NVIDIA HPC SDK's `nvfortran` implements support, GCC's `gfortran`, LLVM's Flang (through `-mp`, and only when Flang is compiled via Clang), and also the HPE Cray Programming Environment. [14, 23, 39, 47]

11 **NVIDIA • Standard • C++:** Standard language parallelism of C++, namely algorithms and data structures of the *parallel STL*, is supported on NVIDIA GPUs through the `nvc++` compiler of the NVIDIA HPC SDK. The key compiler option is `-stdpar=gpu`, which enables offloading of parallel algorithms to the GPU. Also, currently Open SYCL is in the process of implementing support for pSTL algorithms, enabled via `--hipsycl-stdpar`. Further, NVIDIA GPUs can be targeted from Intel's DPC++ compiler, enabling usage of pSTL algorithms implemented in Intel's Open Source oneDPL (*oneAPI DPC++ Library*) on NVIDIA GPUs. Finally, a current proposal in the LLVM community aims at implementing pSTL support through an OpenMP backend. [1, 30, 47]

12 **NVIDIA • Standard • Fortran:** Standard language parallelism of Fortran, mainly `do concurrent`, is supported on NVIDIA GPUs through the `nvfortran` compiler of the NVIDIA HPC SDK. As for the C++ case, it is enabled through the `-stdpar=gpu` compiler option. [47]

13 **NVIDIA • Kokkos • C++:** Kokkos supports NVIDIA GPUs in C++. Kokkos has multiple backends available with NVIDIA GPU support: a native CUDA C/C++ backend (using `nvcc`), an NVIDIA HPC SDK backend (using CUDA support in `nvc++`), and a Clang backend, using either Clang's CUDA support directly or via the OpenMP offloading facilities (via `clang++`). [55]

14 **NVIDIA, AMD, Intel • Kokkos • Fortran:** Kokkos is a C++ programming model, but an official compatibility layer for Fortran (*Fortran Language Compatibility Layer*, FLCL) is available. Through this layer, GPUs can be used as supported by Kokkos C++. [55]

15 **NVIDIA • ALPAKA • C++:** Alpaka supports NVIDIA GPUs in C++ (C++17), either through the NVIDIA CUDA C/C++ compiler `nvcc` or LLVM/Clang's support of CUDA in `clang++`. [41]

16 **NVIDIA, AMD, Intel • ALPAKA • Fortran:** Alpaka is a C++ programming model and no ready-made Fortran support exists. [41]

17 **NVIDIA • etc • Python:** Using NVIDIA GPUs from Python code can be achieved through multiple venues. NVIDIA itself offers CUDA Python, a package delivering low-level interfaces to CUDA C/C++. Typically, code is not directly written using CUDA Python, but rather CUDA Python functions as a backend for higher level models. CUDA Python is available on PyPI as `cuda-python`. An alternative to CUDA Python from the community is PyCUDA, which adds some higher-level features and functionality and comes with its own C++ base layer. PyCUDA is available on PyPI as `pycuda`. The most well-known, higher-level abstraction is CuPy, which implements primitives known from Numpy with GPU support, offers functionality for defining custom kernels, and bindings to libraries. CuPy is available on PyPI as `cupy-cuda12x` (for CUDA 12.x). Two packages arguably providing even higher abstractions are Numba and CuNumeric. Numba offers access to NVIDIA GPUs and features acceleration of functions through Python decorators (*functions wrapping functions*); it is available as `numba` on PyPI. cuNumeric, a project by NVIDIA, allows to access the GPU via Numpy-inspired functions (like CuPy), but utilizes the Legate library to transparently scale to multiple GPUs. [36, 37, 44, 46, 48]

18 **AMD • CUDA • C++:** While CUDA is not directly supported on AMD GPUs, it can be translated to HIP through AMD's HIPIFY. Using `hipcc` and `HIP_PLATFORM=amd` in the environment, CUDA-to-HIP-translated code can be executed. [4]

19 **AMD • CUDA • Fortran:** No direct support for CUDA Fortran on AMD GPUs is available, but AMD offers a source-to-source translator, GPUFORT, to convert some CUDA Fortran to either Fortran with OpenMP (via AOMP) or Fortran with HIP bindings and extracted C kernels (via hipfort). As stated in the project repository, the covered functionality is driven by use-case requirements; the last commit is two years old. [3]

20 **AMD • HIP • C++:** HIP C++ is the *native* programming model for AMD GPUs and, as such, fully supports the devices. It is part of AMD's GPU-targeted ROCm platform, which includes compilers, libraries, tool, and drivers and mostly consists of Open Source Software. HIP code can be compiled with `hipcc`, utilizing the correct environment variables (like `HIP_PLATFORM=amd`) and compiler options (like `--offload-arch=gfx90a`). `hipcc` is a *compiler driver* (wrapper script) which assembles the correct compilation string, finally calling AMD's Clang compiler to generate host/device code (using the AMDGPU backend). [4]

21 **AMD • SYCL • C++:** No direct support for SYCL is available by AMD for their GPU devices. But like for the NVIDIA ecosystem, SYCL C++ can be used on AMD GPUs through third-party software. First, Open SYCL (previously *hipSYCL*) supports AMD GPUs, relying on HIP/ROCm support in Clang. All available internal compilation models can target AMD GPUs. Second, also AMD GPUs can be targeted through both DPC++, Intel's LLVM-based Open Source compiler, and the commercial version included in the oneAPI toolkit (via an AMD ROCm plugin). In comparison to SYCL support for CUDA, no conversion tool like SYCLomatic exists. [1, 32]

22 **AMD • OpenACC • C++:** OpenACC C/C++ is not supported by AMD itself, but third-party support is available for AMD GPUs through GCC or Clacc (similarly to their support of OpenACC C/C++ for NVIDI GPUS). In GCC, OpenACC support can be activated through `-fopenacc`, and further specified for AMD GPUs with, for example, `-foffload=amdgcn-amdhsa="-march=gfx906"`. Clacc also supports OpenACC C/C++ on AMD GPUs by translating OpenACC to OpenMP and using LLVM's AMD support. The enabling compiler switch is `-fopenacc`, and AMD GPU targets can be further specified by, for example, `-fopenmp-targets=amdgcn-amd-amdhsa`. Intel's OpenACC to OpenMP source-to-source translator can also be used for AMD's platform. [13, 16]

23 **AMD • OpenACC • Fortran:** No native support for OpenACC on AMD GPUs for Fortran is available, but AMD supplies GPUFORT, a research project to source-to-source translate OpenACC Fortran to either Fortran with added OpenMP or Fortran with HIP bindings and extracted C kernels (using hipfort). The covered functionality of GPUFORT is driven by use-case requirements, the last commit is two years old. Support for OpenACC Fortran is also available by the community through GCC (`gfortran`) and upcoming in LLVM (Flacc). Also the HPE Cray Programming Environment supports OpenACC Fortran on AMD GPUs. In addition, the translator tool to convert OpenACC source to OpenMP source by Intel can be used. [3, 9, 16]

24 **AMD • OpenMP • C++:** AMD offers AOMP, a dedicated, Clang-based compiler for using OpenMP C/C++ on AMD GPUs (*offloading*). AOMP is usually shipped with ROCm. The compiler supports most OpenMP 4.5 and some OpenMP 5.0 features. Since the compiler is Clang-based, the usual Clang compiler options apply (`-fopenmp` to enable OpenMP parsing, and others). Also in the upstream Clang compiler, AMD GPUs can be targeted through OpenMP; as outlined for NVIDIA GPUs, the support for OpenMP 5.0 is nearly complete, and support for OpenMP 5.1/5.2 is comprehensive. In addition, the HPE Cray Programming Environment supports OpenMP on AMD GPUs. [2, 23, 50]

25 **AMD • OpenMP • Fortran:** Through AOMP, AMD supports OpenMP offloading to AMD GPUs in Fortran, using the `flang` executable and Clang-typical compiler options (foremost `-fopenmp`). Support for AMD GPUs is also available through the HPE Cray Programming Environment. [2, 23]

26 **AMD • Standard • C++:** AMD does not yet provide production-grade support for Standard-language parallelism in C++ for their GPUs. Currently under development is *roc-stdpar* (ROCm Standard Parallelism Runtime Implementation), which aims to supply pSTL algorithms on the GPU and merge the implementation with upstream LLVM. Support for GPU-parallel algorithms is enabled with `-stdpar`. An alternative proposal in the LLVM community aims to support the pSTL via an OpenMP backend. Also Open SYCL is in the process of creating support for C++ parallel algorithms via a `--hipsycl-stdpar` switch. By using Open SYCL's backends, also AMD GPUs are supported. Intel provides the Open Source oneDPL (*oneAPI DPC++ Library*) which implements pSTL algorithms through the DPC++ compiler (see also *C++ Standard Parallelism for Intel GPUs*). DPC++ has experimental support for AMD GPUs. [1, 6, 30]

27 **AMD • Standard • Fortran:** There is no (known) way to launch Standard-based parallel algorithms in Fortran on AMD GPUs.

28 **AMD • Kokkos • C++:** Kokkos supports AMD GPUs in C++ mainly through the HIP/ROCm backend. Also, an OpenMP offloading backend is available. [55]

29 **AMD • ALPAKA • C++:** Alpaka supports AMD GPUs in C++ through HIP or through an OpenMP backend. [41]

30 **AMD • etc • Python:** AMD does not officially support GPU programming with Python, but third-party solutions are available. CuPy experimentally supports AMD GPUs/ROCm. The package can be found on PyPI as `cupy-rocm-5-0`. Numba once had support for AMD GPUs, but it is not maintained anymore. Low-level bindings from Python to HIP exist, for example PyHIP (available as `pyhip-interface` on PyPI). Bindings to OpenCL also exist (PyOpenCL). [44]

31 **Intel • CUDA • C++:** Intel itself does not support CUDA C/C++ on their GPUs. They offer SYCLomatic, though, an Open Source tool to translate CUDA code to SYCL code, allowing it to run on Intel GPUs. The commercial variant of SYCLomatic is called the DPC++ Compatibility Tool and bundled with oneAPI toolkit. The community project chipStar (previously called CHIP-SPV, recently released a 1.0 version) allows to target Intel GPUs from CUDA C/C++ code by using the CUDA support in Clang. chipStar delivers a Clang-wrapper, `cuspv`, which replaces calls to `nvcc`. Also ZLUDA exists, which implements CUDA support for Intel GPUs; it is not maintained anymore, though. [29, 31, 56]

32 **Intel • CUDA • Fortran:** No direct support exists for CUDA Fortran on Intel GPUs. A simple example to bind SYCL to a (CUDA) Fortran program (via ISO C BINDING) can be found on GitHub.

33 **Intel • HIP • C++:** No native support for HIP C++ on Intel GPUs exists. The Open Source third-party project chipStar (previously called CHIP-SPV), though, supports HIP on Intel GPUs by mapping it to OpenCL or Intel's Level Zero runtime. The compiler uses an LLVM-based toolchain and relies on its HIP and SPIR-V functionality. [56]

34 **Intel • HIP • Fortran:** HIP for Fortran does not exist, and also no translation efforts for Intel GPUs.

35 **Intel • SYCL • C++:** SYCL is a C++17-based standard and selected by Intel as the prime programming model for Intel GPUs. Intel implements SYCL support for their GPUs via DPC++, an LLVM-based compiler toolchain. Currently, Intel maintains an own fork of LLVM, but plans to upstream the changes to the main LLVM repository. Based on DPC++, Intel releases a commercial *Intel oneAPI DPC++* compiler as part of the oneAPI toolkit. The third-party project Open SYCL also supports Intel GPUs, by leveraging/creating LLVM support (either SPIR-V or Level Zero). A previous solution for targeting Intel GPUs from SYCL was ComputeCpp of CodePlay. The project became unsupported in September 2023 (in favor of implementations to the DPC++ project). [1, 29, 32]

36 **Intel • OpenACC • C++:** No direct support for OpenACC C/C++ is available for Intel GPUs. Intel offers a Python-based tool to translate source files with OpenACC C/C++ to OpenMP C/C++, the *Application Migration Tool for OpenACC to OpenMP API*. [28]

37 **Intel • OpenACC • Fortran:** Also for OpenACC Fortran, no direct support is available for Intel GPUs. Intel's source-to-source translation tool from OpenACC to OpenMP also supports Fortran, though. [28]

38 **Intel • OpenMP • C++:** OpenMP is a second key programming model for Intel GPUs and well-supported by Intel. For C++, the support is built into the commercial version of DPC++/C++, *Intel oneAPI DPC++/C++*. All OpenMP 4.5 and most OpenMP 5.0 and 5.1 features are supported. OpenMP can be enabled through the `-qopenmp` compiler option of `icpx`; a suitable offloading target can be given via `-fopenmp-targets=spir64`. [29]

39 **Intel • OpenMP • Fortran:** OpenMP in Fortran is Intel's main selected route to bring Fortran applications to their GPUs. OpenMP offloading in Fortran is supported through Intel's Fortran Compiler `ifx` (the new LLVM-based version, not the *Fortran Compiler Classic*), part of the oneAPI HPC Toolkit. Similarly to C++, OpenMP offloading can be enabled through a combination of `-qopenmp` and `-fopenmp-targets=spir64`. [29]

40 **Intel • Standard • C++:** Intel supports C++ standard parallelism (*pSTL*) through the Open Source oneDPL (oneAPI DPC++ Library), also available as part of the oneAPI toolkit. It implements the pSTL on top of the DPC++ compiler, algorithms, data structures, and policies live in the `oneapi::dpl::` namespace. In addition, Open SYCL is current adding support for C++ parallel algorithms, to be enabled via the `--hipsycl-stdpar` compiler option. [30]

41 **Intel • Standard • Fortran:** Standard language parallelism of Fortran is supported by Intel on their GPUs through the Intel Fortran Compiler `ifx` (the new, LLVM-based compiler, not the *Classic* version), part of the oneAPI HPC toolkit. In the oneAPI update 2022.1, the `do concurrent` support was added and extended in further releases. It can be used via the `-qopenmp` compiler option together with `-fopenmp-target-do-concurrent` and `-fopenmp-targets=spir64`. [29]

42 **Intel • Kokkos • C++:** No direct support by Intel for Kokkos is available, but Kokkos supports Intel GPUs through an experimental SYCL backend. [55]

43 **Intel • ALPAKA • C++:** Since v.0.9.0, Alpaka contains experimental SYCL support with which Intel GPUs can be targeted. Also, Alpaka can fall back to an OpenMP backend.

44 **Intel • etc • Python:** Intel GPUs can be used from Python through three notable packages. First, Intel's *Data Parallel Control* (dpctl) implements low-level Python bindings to SYCL functionality. It is available on PyPI as `dpctl`. Second, a higher level, Intel's *Data-parallel Extension to Numba* (numba-dpex) supplies an extension to the JIT functionality of Numba to support Intel GPUs. It is available from Anaconda as `numba-dpex`. Finally, and arguably highest level, Intel's *Data Parallel Extension for Numpy* (dpnp) builds up on the Numpy API and extends some functions with Intel GPU support. It is available on PyPI as `dpnp`, although latest versions appear to be available only on GitHub. [25–27]

## 5 DISCUSSION

While the possible combinations were explained extensively and the given ratings motivated thoroughly, some limitations and caveats exist.

*Model Selection.* The most prominent limitation is the selection of programming models. CUDA, HIP, SYCL, OpenACC, OpenMP, Standard Parallelism, Kokkos, Alpaka, and *Python* were selected for their prevalence in the HPC community – and to focus the scope of this work. But of course, there are further programming models available and used in the community. The most notable exclusion is certainly RAJA [52]. The choice for omitting was made because it is similar in spirit to, albeit not as popular as Kokkos[4]. OpenCL [53] is a further important GPU programming model, but it has never gained much traction in the HPC-GPU space, mostly due to the lukewarm support by NVIDIA. Other models exist, like HPX [35] (which is similar to pSTL support, arguably more extensive, but less *standard*) or C++AMP [42] (which was deprecated in 2022). In principle, also the core of PyTorch [51], libtorch, can be used as a form of programming model. No compatibility libraries were included either, like the *libompx* project [7] which prototypes implementing vendor-agnostic pSTL-like algorithms.

*Performance Evaluation.* A second important limitation is the lack of evaluation of performance. As shown above, many models exist and can target the various GPU types, partly even through different backends. Assessing the level of support, as done here, is at times already challenging; partly even dedicated test suites exist (for example for OpenACC [24, 33] or OpenMP [24, 49]). Judging the performance fairly is even more involved, as a representative selection of micro-benchmarks would need to be ported to the models. For sub-sets of the presented models, performance comparisons do exist. For example by Hammond [18], who compares many NVIDIA-GPU-compatible programming models (and even various implementation routes). Frequently, application-specific use-cases are evaluated on two or more models/devices. An example is by Lin et al. [38], comparing performance of a physics simulation between Kokkos, SYCL, and OpenMP. Closest to an performance overview certainly gives the BabelStream project [12], although only for a STREAM-like algorithm; an example of a recent performance-comparing publication is [11].

*Topicality.* Parts of the field are rapidly evolving. For example, the support for C++ standard parallelism on AMD GPUs made great progress in the past year, and now multiple venues exist. Much of the support is driven by the community, especially for the AMD platform, and it can be hard to assess the current status. In addition, proper documentation sometimes does not exists (yet) and one needs to review the source code. At times, some features are not even advertised in the documentation (like the pSTL support on NVIDIA/AMD GPUs through DPC++). The downside of this evolving field are unmaintained models. For example, it is unclear if GPUFORT [3] is still *officially* supported by AMD. This paper can hence only be seen current at the time of submission.

[4] Although GitHub *Stars* are inherently a flawed metric, RAJA has about one-third as many stars as Kokkos

*Individual Category Discussions.* Of course, some assessments are subject to discussion. For example the ***some support*** category, which is mostly used for *incomplete* support in Figure 1. OpenACC C++ support on NVIDIA GPUs (7) was rated complete, while OpenMP C++ support showed some ambivalence (9). Here, the assessment was made because NVIDIA is upfront in acknowledging that some features of OpenMP for GPU offloading are still missing. Further ambivalence in rating can be seen for Python on NVIDIA GPUs, were the pick-up of the Open Source community was acknowledged through the added ***non-vendor support*** category. C++ standard language parallelism at AMD has most ambivalence, a result of the rapidly changing support through multiple venues – and currently no vendor-supported, advertised solution (which roc-stdpar [6] might become). The double-rating of CUDA C++ on Intel GPUs honors the research project chipStar [56], besides the CUDA-to-SYCL conversion tool [31] by Intel. Finally, C++ standard parallelism for Intel GPUs has ambivalence, as all pSTL functionality currently resides in a custom namespace.

Although due care has been taken when compiling the presented data, there might still be unexplored venues, or changed status. The presented descriptions of this paper reside in a GitHub repository and are open for collaboration through issues or pull requests [20].

## 6 CONCLUSION

This paper presented a methodology to categorize the support of programming models on HPC GPU devices, assessing the level of support and the provider (vendor or third-party). The results for a number of selected models on GPUs of three vendors (AMD, Intel, NVIDIA) were presented in Figure 1, accompanied by extensive descriptions in section 4. The limitations of the method and some key caveats of the presentation were discussed in section 5.

The support for NVIDIA GPUs can be considered most comprehensive, founded in their long-time prevalence in the field.CUDA is possibly the most famous GPU programming model, and both other vendors (AMD, Intel) provide tools for converting CUDA C/C++ to their *native* model (HIP, SYCL). AMD designed HIP closely to mimic CUDA-like programming and enable it other platforms. And, indeed, NVIDIA and AMD GPUs can be used from the same source code, and recently also Intel GPUs with chipStar. SYCL is an entirely different programming model compared to CUDA or HIP, but it also supports all three GPU platform; either by the work by Intel or the community (Open SYCL). While OpenACC can be used on NVIDIA and AMD GPUs, support for Intel GPUs does not exist. OpenMP, on the other hand, is supported on all three platforms – and even for both C++ and Fortran. Standard language parallelism appears to be the model with the fastest change at the moment, with multiple new projects in progress for all three platforms. Kokkos and Alpaka both provide higher-level abstractions and support all three platform. Python, a somewhat outlier in the list, is also well-supported by all three platforms.

While the C++ support appears to be well on the way to good compatibility and portability, the situation looks severely different for Fortran. The only natively supported programming model on all three platforms is OpenMP.

A key component in the ecosystem is the LLVM toolchain. The compilers of AMD, Intel, and NVIDIA are all based on LLVM infrastructure and partly take great effort in upstreaming their changes. Notable are also the open licenses attached to many components, even the key ecosystem compilers (AMD, Intel). Through LLVM, many third-party/community projects are enabled, which now add valuable contributions to the ecosystem (for example Open SYCL).

Not assessed in this work was the performance of programming models. It is a hard task, but might become a future venue for extension of the presented material. Of course, the landscape of Figure 1 evolves swiftly; the progress is tracked in a GitHub repository [20], open for suggestions.

## ACKNOWLEDGMENTS

A previous version of this work was shown in a presentation at a workshop [22]. As it led to many questions and partly heated discussions, the author decided to create a stand-alone version of the comparison and publish it on GitHub [20] with an accompanying blogpost [21], now utilizing source data in YAML form with conversion to HTML and TeX. As interest continued, the material was updated and significantly extended to become this paper. The goal is a living overview of the evolving field, with snapshots in paper form at regular intervals.

The author would like to acknowledge the frequent discussions with his colleague of Jülich Supercomputing Centre, trying to understand the level of support and determine also corner-cases and unpublished routes of GPU support.

## REFERENCES

[1] Aksel Alpay, Bálint Soproni, Holger Wünsche, and Vincent Heuveline. 2022. Exploring the possibility of a hipSYCL-based implementation of oneAPI. In *International Workshop on OpenCL*. ACM. https://doi.org/10.1145/3529538.3530005
[2] AMD. 2023. *AOMP*. https://github.com/ROCm-Developer-Tools/aomp
[3] AMD. 2023. *GPUFORT*. https://github.com/ROCmSoftwarePlatform/gpufort
[4] AMD. 2023. *HIP*. https://rocm.docs.amd.com/projects/HIP/en/latest/
[5] AMD. 2023. *hipfort*. https://rocm.docs.amd.com/projects/hipfort/en/latest/
[6] AMD. 2023. *roc-stdpar*. https://github.com/ROCmSoftwarePlatform/roc-stdpar
[7] Libompx Authors and Contributors. [n. d.]. *Libompx*. https://github.com/markdewing/libompx/tree/add_catch
[8] Pierre Carbonelle. 2023. *PopularitY of Programming Language*. https://pypl.github.io/PYPL.html
[9] Valentin Clement and Jeffrey S. Vetter. 2021. Flacc: Towards OpenACC support for Fortran in the LLVM Ecosystem. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 12–19. https://doi.org/10.1109/LLVMHPC54804.2021.00007
[10] TOP500 Compilers. 2023. *TOP500 List*. https://www.top500.org/lists/top500/2023/06/
[11] Tom J Deakin, Andrei Poenaru, Tom Lin, and Simon N Mcintosh-Smith. 2021. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *Proceedings of P3HPC 2020 (Proceedings of P3HPC 2020: International Workshop on Performance, Portability, and Productivity in HPC, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis)*. Institute of Electrical and Electronics Engineers (IEEE), United States, 1–13. https://doi.org/10.1109/P3HPC51967.2020.00006 Publisher Copyright: © 2020 IEEE..
[12] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262. https://doi.org/10.1504/IJCSE.2018.095847 arXiv:https://www.inderscienceonline.com/doi/pdf/10.1504/IJCSE.2018.095847
[13] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2018. CLACC: Translating OpenACC to OpenMP in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 18–29. https://doi.org/10.1109/LLVM-HPC.2018.8639349
[14] GCC Developers. 2023. *GCC OpenMP*. https://gcc.gnu.org/wiki/openmp

[15] LLVM/Clang Developers. 2023. *Clang OpenMP*. https://clang.llvm.org/docs/OpenMPSupport.html

[16] GCC. 2023. *GCC OpenACC*. https://gcc.gnu.org/wiki/OpenACC

[17] Khronos Group. 2023. *SYCL*. https://www.khronos.org/sycl/

[18] Jeff Hammond. 2022. Shifting through the Gears of GPU Programming: Understanding Performance and Portability Trade-offs. https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41620/ GTC Digital Spring Conference.

[19] Jeff R Hammond, Tom Deakin, Jim H Cownie, and Simon N McIntosh-Smith. 2022. Benchmarking Fortran DO CONCURRENT on CPUs and GPUs Using BabelStream. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Institute of Electrical and Electronics Engineers (IEEE), United States, 1–18. https://doi.org/10.1109/PMBS56514.2022.00013 SC 2022 Workshops International Conference for High Performance Computing, Networking, Storage and Analysis ; Conference date: 13-11-2022 Through 18-11-2022.

[20] Andreas Herten. 2022. *GPU Vendor/Programming Model Compatibility Table*. https://github.com/AndiH/gpu-lang-compat

[21] Andreas Herten. 2022. GPU Vendor/Programming Model Compatibility Table. (2022). https://doi.org/10.34732/XDVBLG-R1BVIF

[22] Andreas Herten and Kaveh Haghighi Mood. 2022. Many Ways to GPUs - A GPU Introduction. https://juser.fz-juelich.de/record/916369 The content is also available at https://www.nat-esm.de/services/documentation.

[23] HPE. 2023. *HPE Cray Programming Environment*. https://www.hpe.com/psnow/doc/a50002303enw

[24] Thomas Huber, Swaroop Pophale, Nolan Baker, Michael Carr, Nikhil Rao, Jaydon Reap, Kristina Holsapple, Joshua Hoke Davis, Tobias Burnus, Seyong Lee, David E. Bernholdt, and Sunita Chandrasekaran. 2022. ECP SOLLVE: Validation and Verification Testsuite Status Update and Compiler Insight for OpenMP. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 123–135. https://doi.org/10.1109/P3HPC56579.2022.00017

[25] Intel. 2023. *Data Parallel Control*. https://github.com/IntelPython/dpctl

[26] Intel. 2023. *Data Parallel Extension for Numpy*. https://github.com/IntelPython/dpnp

[27] Intel. 2023. *Data-parallel Extension to Numba*. https://github.com/IntelPython/numba-dpex

[28] Intel. 2023. *Intel Application Migration Tool for OpenACC to OpenMP API*. https://github.com/intel/intel-application-migration-tool-for-openacc-to-openmp

[29] Intel. 2023. *oneAPI*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html

[30] Intel. 2023. *oneDPL*. https://oneapi-src.github.io/oneDPL/index.html

[31] Intel. 2023. *SYCLomatic*. https://github.com/oneapi-src/SYCLomatic

[32] Intel and Contributors. 2023. *oneAPI DPC++ Compiler*. https://github.com/intel/llvm LLVM-fork with DPC++ support by Intel..

[33] Aaron Jarmusch and Sunita Chandrasekaran. [n. d.]. *OpenACC Verification and Validation Testsuite*. https://crpl.cis.udel.edu/oaccvv/

[34] Aaron Jarmusch, Aaron Liu, Christian Munley, Daniel Horta, Vaidhyanathan Ravichandran, Joel Denny, Kyle Friedline, and Sunita Chandrasekaran. 2022. Analysis of Validating and Verifying OpenACC Compilers 3.0 and Above. In *2022 Workshop on Accelerator Programming Using Directives (WACCPD)*. 1–10. https://doi.org/10.1109/WACCPD56842.2022.00006

[35] Hartmut Kaiser, Mikael Simberg, Bryce Adelstein Lelbach, Thomas Heller, Agustin Berge, John Biddiscombe, Auriane Reverdell, Anton Bikineev, Grant Mercer, Andreas Schaefer, Kevin Huck, Adrian Lemoine, Taeguk Kwon, Jeroen Habraken, Matthew Anderson, Steven R. Brandt, Marcin Copik, Srinivas Yadav, Martin Stumpf, Daniel Bourgeois, Akhil Nair, Denis Blank, Giannis Gonidelis, Rebecca Stobaugh, Nikunj Gupta, Shoshana Jakobovits, Vinay Amatya, Lars Viklund, Patrick Diehl, and Zahra Khatami. 2023. *STEllAR-GROUP/hpx: HPX V1.9.1: The C++ Standards Library for Parallelism and Concurrency*. https://doi.org/10.5281/zenodo.5185328

[36] Andreas Kloeckner, Gert Wohlgemuth, Gregory Lee, Tomasz Rybak, Alex Nitz, David Chiang, Stan Seibert, Martin Bergtholdt, Thomas Unterthiner, Graham Markall, Mit Kotak, Vincent Favre-Nicolin, Bogdan Opanchuk, Bruce Merry, Nicolas Pinto, Fabrizio Milo, Thomas Collignon, Florian Rathgeber, Simon Perkins, Vladimir Rutsky, Bryan Catanzaro, Alex Park, Freddie Witherden, Lev E. Givon, Luke Pfister, Marcus Brubaker, RA ZA, Loic Hausammann, and Christoph Gohlke. 2023. *PyCUDA*. https://doi.org/10.5281/zenodo.8121901

[37] Siu Kwan Lam, stuartarchibald, Antoine Pitrou, Mark Florisson, Stan Seibert, Graham Markall, esc, Todd A. Anderson, Guilherme Leobas, rjenc29, Michael Collison, luk-f a, Jay Bourque, Aaron Meurer, Kaustubh, Travis E. Oliphant, Nick Riasanovsky, Michael Wang, densmirn, njwhite, Ethan Pronovost, Ehsan Totoni, Eric Wieser, Stefan Seefeld, Hernan Grecco, Andre Masella, Pearu Peterson, Isaac Virshup, Matty G, and Itamar Turner-Trauring. 2023. *numba/numba: Version 0.57.1*. https://doi.org/10.5281/zenodo.8087361

[38] Meifeng Lin, Zhihua Dong, Tianle Wang, Mohammad Atif, Meghna Battacharya, Kyle Knoepfel, Charles Leggett, Brett Viren, and Haiwang Yu. 2023. Portable Programming Model Exploration for LArTPC Simulation in a Heterogeneous Computing Environment: OpenMP vs. SYCL. (4 2023). https://www.osti.gov/biblio/1973454

[39] LLVM/Flang. 2023. *Flang*. https://flang.llvm.org/

[40] George S. Markomanolis, Aksel Alpay, Jeffrey Young, Michael Klemm, Nicholas Malaya, Aniello Esposito, Jussi Heikonen, Sergei Bastrakov, Alexander Debus, Thomas Kluge, Klaus Steiniger, Jan Stephan, Rene Widera, and Michael Bussmann. 2022. Evaluating GPU Programming Models for the LUMI Supercomputer. In *Supercomputing Frontiers*, Dhabaleswar K. Panda and Michael Sullivan (Eds.). Springer International Publishing, Cham, 79–101.

[41] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, and M. Bussmann. 2017. Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library. arXiv:1706.10086 https://arxiv.org/abs/1706.10086

[42] Microsoft. 2023. *C++ AMP*. https://learn.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-cpp-accelerated-massive-parallelism?view=msvc-170

[43] NVIDIA. 2023. *CUDA Fortran*. https://developer.nvidia.com/cuda-fortran

[44] NVIDIA. 2023. *CUDA Python*. https://nvidia.github.io/cuda-python/index.html

[45] NVIDIA. 2023. *CUDA Toolkit*. https://developer.nvidia.com/cuda-toolkit

[46] NVIDIA. 2023. *cuNumeric*. https://developer.nvidia.com/cunumeric

[47] NVIDIA. 2023. *NVIDIA HPC SDK*. https://developer.nvidia.com/hpc-sdk

[48] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf

[49] Swaroop Pophale, Felipe Cabarcas, and Sunita Chandrasekaran. [n. d.]. *OpenMP Validation and Verification Testsuite*. https://crpl.cis.udel.edu/ompvvsollve

[50] ECP Exascale Computing Project. 2022. OpenMP Roadmap for Accelerators Across DOE Pre-Exascale/Exascale Machines. https://www.openmp.org/wp-content/uploads/2022_ECP_Community_BoF_Days-OpenMP_RoadMap_BoF.pdf

[51] PyTorch Authors and Contributors. 2023. *PyTorch C++ Interface*. https://pytorch.org/cppdocs/frontend.html

[52] RAJA Authors and Contributors. [n. d.]. *RAJA Performance Portability Layer*. https://github.com/LLNL/RAJA

[53] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[54] TIOBE. 2023. *TIOBE Index*. https://www.tiobe.com/tiobe-index/

[55] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[56] Jisheng Zhao, Colleen Bertoni, Jeffrey Young, Kevin Harms, Vivek Sarkar, and Brice Videau. 2023. HIPLZ: Enabling Performance Portability for Exascale Systems. In *Euro-Par 2022: Parallel Processing Workshops*, Jeremy Singer, Yehia Elkhatib, Dora Blanco Heras, Patrick Diehl, Nick Brown, and Aleksandar Ilic (Eds.). Springer Nature Switzerland, Cham, 197–210.