Evaluating GPU Programming Models for the LUMI Supercomputer

George S. Markomanolis[1], Aksel Alpay[2], Jeffrey Young[5], Michael Klemm[3], Nicholas Malaya[3], Aniello Esposito[4], Jussi Heikonen[1], Sergei Bastrakov[6], Alexander Debus[6], Thomas Kluge[6], Klaus Steiniger[6], Jan Stephan[6,7(✉)], Rene Widera[6], and Michael Bussmann[6,7]

[1] CSC - IT Center for Science Ltd., Espoo, Finland
{georgios.markomanolis,jussi.heikonen}@csc.fi
[2] Heidelberg University, Heidelberg, Germany
aksel.alpay@uni-heidelberg.de
[3] Advanced Micro Devices Inc, Santa Clara, USA
{michael.klemm,nicholas.malaya}@amd.com
[4] Hewlett Packard Enterprise, Spring, USA
aniello.esposito@hpe.com
[5] Georgia Institute of Technology, Atlanta, USA
jyoung9@gatech.edu
[6] Center for Advanced Systems Understanding, Görlitz, Germany
[7] Helmholtz-Zentrum Dresden-Rossendorf, Dresden, Germany
{s.bastrakov,a.debus,t.kluge,k.steiniger,
j.stephan,r.widera,m.bussmann}@hzdr.de

Abstract. It is common in the HPC community that the achieved performance with just CPUs is limited for many computational cases. The EuroHPC pre-exascale and the coming exascale systems are mainly focused on accelerators, and some of the largest upcoming supercomputers such as LUMI and Frontier will be powered by AMD Instinct™ accelerators. However, these new systems create many challenges for developers who are not familiar with the new ecosystem or with the required programming models that can be used to program for heterogeneous architectures. In this paper, we present some of the more well-known programming models to program for current and future GPU systems. We then measure the performance of each approach using a benchmark and a mini-app, test with various compilers, and tune the codes where necessary. Finally, we compare the performance, where possible, between the NVIDIA Volta (V100), Ampere (A100) GPUs, and the AMD MI100 GPU.

Keywords: GPU · Programming models · HIP · CUDA · OpenMP · hipSYCL · Kokkos · Alpaka

1 Introduction

Europe has procured a number of supercomputers through the EuroHPC Joint Undertaking (JU) organization. In this work, we focus on the LUMI [1] super-

D. K. Panda and M. Sullivan (Eds.): SCFA 2022, LNCS 13214, pp. 79–101, 2022.
https://doi.org/10.1007/978-3-031-10419-0_6

computer which is being installed in Finland by Hewlett-Packard Enterprise and is run by a consortium of ten European countries. LUMI will have both a CPU and a GPU partition, where the CPU partition performance is only a few petaflops, the AMD Instinct^{TM} GPUs provide almost 0.5 EFLOPS across 2560 nodes with 64 core AMD Trento CPU and four AMD MI250X GPUs, using similar technology as the Frontier system [2].

There are a few parallel older programming models, however, with the arrival of the GPUs, other programming models had to be created, such as Compute Unified Device Architecture (CUDA) [3], OpenCL [4], or directive-based programming models such as OpenMP [5] and OpenACC [6]. Meanwhile, even more programming models have emerged, some of which are more widely known than others. For some of them, there is a significant learning curve, and others are to be used by HPC domain experts.

When a scientist prepares an application to be ported to a GPU architecture, or to move from NVIDIA GPUs to AMD GPUs, the effort often depends on the used programming model. With such a wide variety of available programming models, it sometimes is not straightforward which one to use. In this paper, we explore the porting procedure for the LUMI supercomputer, discuss the applicable programming models, and present some results of various benchmarks and performance comparisons across AMD and NVIDIA GPUs.

The main contributions of this work are as follows:

- We present a porting diagram that illustrates how the LUMI users could port their application in various scenarios.
- To our knowledge, this is one of the first comparisons between NVIDIA V100/A100, and AMD MI100.
- We evaluate the performance of many programming models such as HIP, CUDA, OpenMP Offloading, hipSYCL, Kokkos, and Alpaka while optimizing when possible.
- We present results of the BabelStream with Alpaka backend for the first time.
- We present how to tune some of the kernels.

2 Related Work

For many programming models, there are studies that evaluate these for CPUs or GPUs. In [7] the authors study OpenMP offload on NVIDIA V100 with a few mini-apps and various compilers, observe performance variations, and provide some OpenMP optimization techniques. In [8], the authors present the compute-bound mini-app miniBUDE and evaluate various programming models, including offload to GPUs. In [9], the authors present a performance analysis of CUDA, OpenACC, and OpenMP programming models on V100 GPU where they illustrate how it is easier to use OpenMP offloading and OpenACC compared to CUDA, and they measure the performance. Deakin et al. in [10] evaluate the performance of benchmarks in SYCL and comparing them with an OpenCL version. They use three applications for this purpose. The authors in [11] present a performance portability study on different CPUs/GPUs, using programming

models and codes to investigate the performance portability. However, not many of them supported an AMD GPU at that time.

Compared to the current related work, we use some new GPUs, and especially the AMD MI100 and evaluate the programming models and tune them based on its hardware specifications. Furthermore, we try to use the most recent versions of the programming models where possible. Finally, we provide box plots in some cases to identify variations.

3 Programming Models

In this section, we present a few programming models that we plan to use on LUMI, and later we describe in which situation to use them.

3.1 HIP

The Radeon Open Compute (ROCm) platform [12,13] includes programming models to develop codes for AMD GPUs. Among those is the Heterogeneous-compute Interface for Portability (HIP) [14]. HIP is a C++ API and kernel language to create portable applications for the AMD ROCm platform as well as NVIDIA GPUs using the same source code. It is open source, it provides an API to port your code, and the syntax is very similar to CUDA. It supports a large subset of the CUDA runtime functionality and has almost no negative performance impact over coding directly in CUDA. HIP includes features such as C++11 lambdas, namespaces, classes, templates, etc. The HIPify [15] tools convert CUDA code to HIP. Of course, tuning will be required for each specific GPU.

Table 1 exemplifies some similarities between CUDA and HIP. For most cases, replacing the "cuda" in the function name with "hip" as well as for the arguments is enough to translate the API. However, not all the CUDA API is supported in HIP yet. Executing a GPU kernel is similar as you can see in the corresponding table but there is also a HIP API called hipLaunchKernelGGL.

With the HIP translation tool, a common approach is to semi-automatically port a CUDA code to HIP. For example, to convert a CUDA file called *example.cc*, the command `hipify-perl --inplace example.cc` performs the translation of CUDA API calls to HIP API calls (a backup of the original code is kept as `example.cc.hip`. There is also `hipconvertinplace-perl.sh` to translate all source files of a directory as well as a version of the HIPify tool that is based on the clang compiler. For more details about porting codes there are a few sources such as [16,17].

For CUDA Fortran codes, it is required to do the following steps (further details are available at [17]):

- Port CUDA Fortran code to HIP kernels in C++. The `hipfort` API helps to invoke the HIP API from Fortran.
- Wrap the kernel launch in function with C calling convention.
- Call the launch function from Fortran through the Fortran 2003 C bindings.

Table 1. Convert CUDA code to HIP

CUDA	HIP	Description
cudaMemcpy	hipMemcpy	Copy data between two different memory locations
cudaMalloc	hipMalloc	Allocates a memory pointer on the device
cudaFree	hipFree	Deallocate memory from the GPU
kernel_name <<< gridsize, blocksize, shared_mem_size, stream >>>(arg0, arg1, ...);	kernel_name <<< gridsize, blocksize, shared_mem_size, stream >>>(arg0, arg1, ...);	Execute a GPU kernel

3.2 The OpenMP Application Programming Interface

The OpenMP API supports offloading computation to accelerator devices since version 4.0 and has since then refined and extended the features continuously [18]. The OpenMP API supports a variety of *target* directives that control the transfer of data (if needed), transfer of control flow, as well as parallelism on the target device. OpenMP also offers low-level API interfaces for memory allocation and data transfers similar to the interfaces of the CUDA and HIP programming models.

This is a very basic example of an OpenMP offload region, running code on a GPU:

```
#pragma omp target teams distribute parallel for simd \
            map(to:A[:N]) map(from:B[:N]) \
            num_teams(x) thread_limit(y)
for (int i = 0; i < N; ++i) {
    B[i] = expression(A[i], i);
}
```

In the example above, the `target` construct transfers the control flow from the host device to the default target device (the host thread will await completion of the offload region). The `map` clauses are used to specify the data that is needed for execution as well as the direction of the data flow. If the host and accelerator have distinct memories, the OpenMP implementation will perform an actual transfer. If host and device have a shared memory (emulation), the `map` clauses do not issue an actual data transfer.

Since the OpenMP API does not only support GPU-like architectures as target devices, it has been a design decision by the OpenMP Language Committee to separate offload directives and parallelism from each other. Through this decision programmers can use the best matching OpenMP directives to

create parallelism for a specific target architecture. Also, the OpenMP API supports a more descriptive approach via the `loop` construct instead of the `teams distribute parallel for` construct.

The `teams distribute` directive then partitions the loop iteration space across the available warps or wavefronts, while the `parallel for simd` constructs can parallelize the partitioned loop for the available GPU threads. Another approach is to map `parallel for` to a single GPU thread and use `simd` to create parallelism within the warp/wavefront. OpenMP explicitly allows for this flexibility in laying out the execution on the GPU, such that implementations can pick the best possible strategy.

Many compilers now have (partial) support for version 5.0 and version 5.1 of the OpenMP API. In this work, we use only OpenMP offloading as we benchmark GPU accelerators. For AMD GPUs, we rely on the AMD OpenMP compiler (AOMP).

3.3 SYCL

SYCL [19] is an open standard for heterogeneous programming. It is developed and maintained by the Khronos Group. Unlike other heterogeneous programming models, SYCL does not introduce any custom syntax extensions or pragmas. Instead, expresses heterogeneous data parallelism with pure C++. The latest SYCL version is SYCL 2020, which relies on C++17. Originally, SYCL was intended as a higher-level single-source model for OpenCL. This means that in contrast to OpenCL, host and device code reside in the same source file in SYCL, and are processed together by the SYCL compiler. Starting with SYCL 2020, a generalized backend architecture was introduced that allows for other backends apart from OpenCL. Backends used by current SYCL implementations include OpenCL, Level Zero, CUDA, HIP and others.

While a more task-oriented model is available as well, SYCL currently strongly focuses on data parallel kernels. The execution of these kernels is organized by a task graph that is maintained by the SYCL runtime. There are two memory management models in SYCL: the buffer-accessor model and the unified shared memory (USM) model.

In the buffer-accessor model, the SYCL runtime handles data transfers automatically according to data access specifications given by the programmer. These are also used by the SYCL runtime to automatically construct a task graph for the execution of kernels. In the pointer-based USM model, the programmer is responsible for correctly inserting dependencies between kernels and making sure that data is available on the device when necessary. While the buffer-accessor model may introduce overheads due to the evaluation of the access specifications and calculatation of kernel dependencies, if the scheduler receives detailed information that can be used to optimize the task graph execution.

The execution model in SYCL is largely inherited from OpenCL. Parallel work items are grouped into work groups, and synchronization is only possible within a work group. Starting with SYCL 2020, work groups are additionally subdivided into subgroups that are typically mapped to SIMD units. On GPUs,

a SYCL work group usually corresponds to a thread block from HIP or a team in the OpenMP model. As such, the SYCL work-group size is a tuning parameter as in those other models. In SYCL, multiple methods exist to invoke kernels. In the simplest method, `parallel_for`, the work groups are not exposed and, on GPUs, a SYCL implementation automatically selects an appropriate work group size. In the more complex `nd_range` model, the user is responsible for choosing an appropriate work group size.

There are multiple implementations of SYCL. The most well-known implementations include ComputeCpp [20], DPC++ [21], hipSYCL [22] and triSYCL [23]. In this work, we will be using hipSYCL as it has mature support both for the GPUs investigated in this work. hipSYCL consists of a multi-backend runtime with support for CPUs and GPUs from AMD, NVIDIA and Intel, the SYCL kernel and runtime header library, as well as a compiler component with a unified compiler driver called *syclcc*. This compiler component is designed to integrate with existing compiler toolchains. For example, when compiling for NVIDIA and AMD GPUs, hipSYCL acts as an additional layer on top of CUDA and HIP. During compilation, hipSYCL loads an additional clang plugin that extends clang's native HIP and CUDA support with support for SYCL-specific constructs, such as automatic kernel detection and outlining. This design not only allows a user to mix-and-match CUDA or HIP kernel code with SYCL code even within one kernel, it also allows using vendor-supported toolchains with hipSYCL since e.g. AMD's official ROCm HIP compiler uses the same clang HIP toolchain. Consequently, hipSYCL can be deployed on top of the AMD HIP compiler.

3.4 OpenACC

OpenACC is a directive programming model for the GPUs that has evolved significantly since its beginning. Initially, there were two options for OpenACC support on LUMI. First, the HPE/Cray compiler supports only Fortran and OpenACC version 2.7, with potential for up to v3.1 until end of 2022. Second, the GNU compiler [24], which is not a contractual agreement. Thus, our guidance is not recommending OpenACC without also mentioning these caveats.

For illustration, the following OpenACC directive uses a few clauses. The `gang` clause corresponds to the thread blocks, while the `worker` clause is the warp or wavefront, and `vector` is the threads:

```
#pragma acc parallel loop  \
            copyin(A[:N]) copyout(B[:N]) \
            vector_length() gang worker num_workers()
            ...
```

As GCC with offload to AMD MI100 GPUs is not focus on performance this moment, but more to functionality, we do not report OpenACC results. We mention though that GCC v10.3, v11.1, and later have fixed an issue that GPU memory was cleaned too often and as a result the performance on NVIDIA GPUs is improved by almost 30% for all BabelStream kernels except the `dot`

kernel for which the performance remained similar. Moreover, in the future we plan to explore a research project called *clacc* [25,26] that provides OpenACC support for Clang and LLVM. This will allow for simplified porting of OpenACC codes to the OpenMP API (amongst other benefits).

3.5 Alpaka

The Abstraction Library for Parallel Kernel Acceleration (alpaka) [27] is implemented as a header-only C++14 abstraction library for accelerator development and portability. Originally developed to support large-scale scientific applications like PIConGPU [28], alpaka enables an accelerator-agnostic implementation of hierarchical redundant parallelism, that is, the API allows a user to specify data and task parallelism at multiple levels of compute and memory for a particular platform. Currently, alpaka provides support for backends for OpenMP, (C++) threads, Intel Threading Building Blocks, CUDA, HIP, and SYCL for FPGA along with new backends for directives in development.

Alpaka code can be used to express hierarchical parallelism for both CPU-style and GPU devices. In addition to grids, blocks, and threads, alpaka also provides an element construct that represents an n-dimensional set of inputs that is amenable to vectorization by the compiler. This extra level of parallelism is key to achieve good performance when attempting to map GPU-style kernels to a CPU architectures that offer SIMD instructions as part of their instruction set architecture.

In addition to the optimized kernels via alpaka, users can also use the C++ User interface for the Platform independent Library Alpaka (cupla) [29] to port CUDA code to use the alpaka library. Cupla codes have a very similar syntax to regular CUDA kernels and can include calls to the CUDA API for data allocation and movement. While cupla introduces some host-side API call overhead compared to pure alpaka, it provides a suitable path to map existing codes to alpaka's supported backends.

3.6 Kokkos

The Kokkos [30] C++ Performance Portability Ecosystem is a framework for writing modern C++ applications with portability across a variety of hardware. It is part of the Exascale Computing Project (ECP) and is used by many HPC users and packages. It supports several backends, such as CUDA, HIP, SYCL, and OpenMP offloading to target various accelerators, including NVIDIA and AMD GPUs.

The Kokkos abstraction layer maps C++ source code to the specific instructions required for the backend during build time. When compiling the source code, the binary will be built for the declared backends:

- Serial backend, for serial code on a host device.
- Host-parallel backend, which executes in parallel on the host device (OpenMP API, etc.).
- Device-parallel backend, which offloads on a device, such as a GPU.

4 Choosing a Programming Model

Figures 1 and 2 present the porting diagrams of potential codes targeting the LUMI system. Initially, developers make a decision based on whether the code is already able to use a GPU or not. If not (see Fig. 1), there is an option for the developer to try various programming models such as SYCL, Alpaka, or Kokkos if the application's programming language is supported. Expert developers could port their code directly to HIP, identifying the kernels and preparing them similarly to CUDA. If the code does not have OpenMP directives, a tool such as Cray Reveal could be used to port the code to the OpenMP API. This procedure can be more productive for Fortran applications, and it could be expanded to the rest main programming languages later. Then the developer can manually port the OpenMP CPU code to user OpenMP `target` directives.

Then, a standard software tuning cycle can kick in. If the performance is not as expected and desired, then developers profile and tune the OpenMP directives, especially avoiding unnecessary data transfers. This cycle repeats until the problem is solved and the code works as expected. Otherwise, some OpenMP offload regions can be ported to HIP to expose more control over kernel execution. It should be mentioned that at the time of writing, the OpenMP implementation of AMD is composable with the HIP API, but requires to keep OpenMP code and HIP code in separate compilation units. If the code is in C/C++, then profile, identify the kernels, and port to HIP. If the code is CUDA Fortran, then it is required to use a Fortran interface for GPU kernels, called hipfort [31], to port the code to HIP. The developers could also use OpenACC instead of the OpenMP API once the compilers are available, but it depends on whether the applications are already using OpenMP directives or not, and what preference for the programming model is.

If the application is already ported to a GPU (see Fig. 2) there is a possibility to use the programming models such as SYCL, Alpaka, etc. If the initial application is developed in OpenACC, there are three options that are divided into sub-categories. First of all the Cray compilers are supporting only Fortran codes. LUMI is an HPE supercomputer, which means there is contractual engagement on the availability of some programming models. On the other side, the GCC efforts were mentioned in the OpenACC section before. Finally, the research projects such as Clacc, and Flacc which provides OpenACC support for Flang, are not yet in the final state. If the performance with any of the previous solutions is not good, then port the OpenACC calls to OpenMP to investigate the performance and tune the code.

If the GPU code is written in CUDA, and if it is C/C++ code it could be ported with the HIPify tools, while if it is in CUDA Fortran, then the hipfort should be used as also described in Sect. 3.1. Finally, if the performance is not as expected, a similar software tuning cycle is used to resolve the issue.

5 Benchmarks and Applications

5.1 BabelStream

BabelStream [32,33] is a memory bound benchmark with many programming models implemented. There are five computational kernels that we are using, the add ($a[i] = b[i] + c[i]$), multiply ($a[i] = b * c[i]$), copy ($a[i] = b[i]$), triad ($a[i] = b[i] + d * c[i]$), and dot ($sum = sum + a[i] * b[i]$). The default problem size is 2^{25} FP64 operations and 100 iterations. We are evaluating BabelStream v3.4 (6fe81e1). We developed the Alpaka backend for BabelStream for which we present some results in this paper.

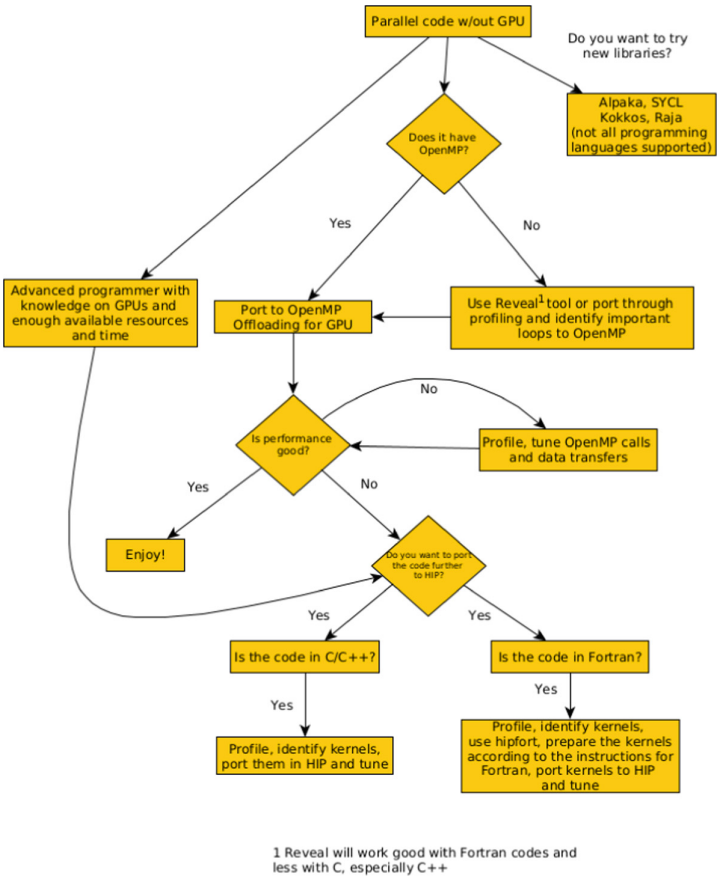

Fig. 1. Diagram for porting CPU applications to LUMI

5.2 MiniBUDE

We use also the mini-app called miniBUDE for the Bristol University Docking Engine (BUDE) [34], a kernel of a drug discovery application that is compute bound and provides performance results in single precision. BUDE is designed for in silico molecular docking. In the computationally intensive virtual screening, molecules of drug candidates, known as ligands, are bonded to target protein molecule. BUDE predicts the binding energy of the ligand with the target, however, there are many ways this bonding could happen, and a variety of positions and rotations of the ligand relative to protein, known as poses, are explored. And for each pose, a number of properties are evaluated. We are evaluating the version with commit 1af5b39.

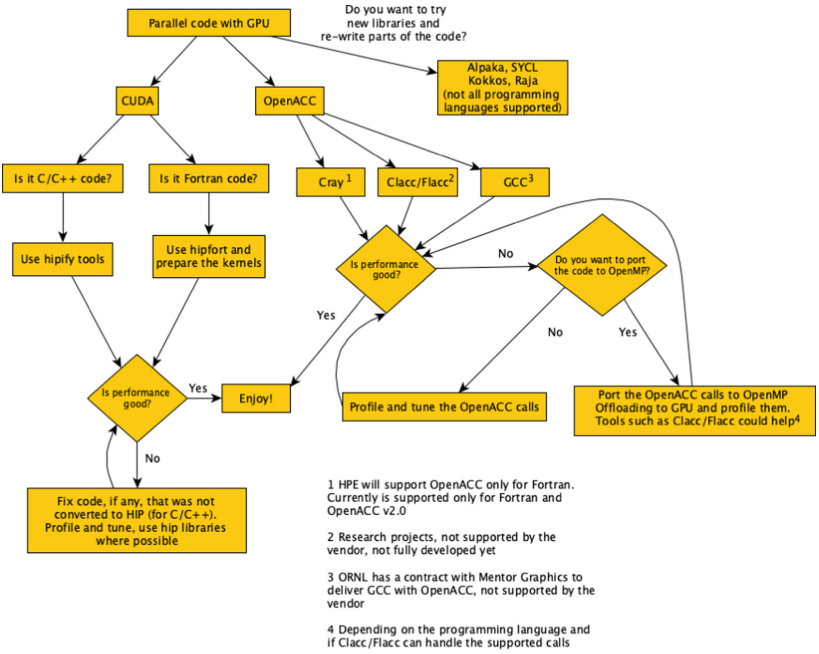

Fig. 2. Diagram for porting GPU applications to LUMI

Both BabelStream and miniBUDE support many programming models such as HIP, CUDA, OpenMP Offloading, SYCL, OpenACC, Kokkos.

6 Methodology

6.1 Compilation

For the compilation, we used the provided instructions from the benchmark and application. For the miniBUDE, we had some concerns about our installation

that we will discuss later as we could not achieve the expected performance but we had no issue with the BabelStream.

6.2 Execution and Tuning

We save the data from ten executions (in the same submission script, so using the same compute nodes). We then visualize the results in a box plot to determine variations and to ensure sure that our observations are correct. As our runs do not include MPI, we try to investigate if binding the processes helps in some cases or trying to have a process as close to the GPU as possible, but for our cases, we did not observe any significant improvement. We tune where possible by adjusting the number of the thread blocks to be a multiple of the number of compute units for the MI100 or streaming multiprocessors for V100/A100 GPUs.

7 Results

7.1 Configuration

We plan to utilize a single GPU, as we do not want to interfere with MPI performance evaluation in this work. CSC provides two supercomputers called Puhti and Mahti. The AMD Accelerator Cloud is a remote, heterogeneous system provided by AMD. Technical details about the GPUs specifications are presented in Table 2.

The **Puhti** [35] supercomputer at CSC, is constituted by 682 CPUs and 80 GPU nodes. Each GPU node has two Intel Xeon Gold 6230 processors with 20 cores each, and four NVIDIA V100 with 32GB HBM2 memory each. The interconnect is based on a dual-rail Mellanox HDR100 fabric.

The **Mahti** [36] supercomputer has 1404 CPUs and 24 GPU nodes. Each GPU node has two AMD EPYC^{TM} 7H12 Processors ("Rome") with 64 cores each, four NVIDIA A100 with 40 GB HBM2 memory for each one, and a total of 512 GB of memory.

Table 2. List of utilized GPU architectures and specifications

Vendor	Model	HBM Memory (GB)	MemoryBandwidth (GB/s)	Threads	Peak FP64 (TFLOPS)	Peak FP32 (TFLOPS)
NVIDIA	V100	32	900	5,120	7.8	15.7
NVIDIA	A100	40	1,555	6,912	9.7	19.5
AMD	MI100	32	1,200	7,680	11.5	23.1

The **AMD Accelerator Cloud** offers different GPU options. We used a node with two AMD EPYC 7742 Processors and four AMD Instinct MI100 accelerators.

In Table 3 we mention the compilers/software for each system that participated in our study and their versions.

7.2 BabelStream

In this subsection we present the results from the BabelStream and do comparisons between GPUs.

Versions and Generic Tuning

We mention some software versions and generic tuning that applies in most of the kernels below. If a kernel has a different tuning, it will be mentioned in the corresponding kernel. About HIP, we decrease the number of threads per block to 256 instead of 1024, and achieve on average a performance improvement of up to 28% than using the default number of threads. The AMD MI100 has 120 compute units, thus when the blocks of threads are a multiple of 120, are usually more efficient for this GPU because we hide the latency cost. It is known that the AOMP is under heavy development to achieve better performance. We work with one of the latest AOMP versions instead of building the LLVM from the provided AMD GitHub repository through ROCm, as the AOMP is closer to production. Moreover, according to our tests, all the kernels perform around 5% better between AOMP 13.x and AOMP 12.x except the *dot* kernel that it is improved with a factor of 2.7. One of the reasons is also that AOMP 13.x creates automatically two times more block of threads compared to version 12.x, including the AOMP performance improvements and the utilization of LLVM 13. A table which displays the range of the percentage of speedup of MI100 over V100 or the slowness of MI100 over A100 will be presented for each kernel. For all the experiments, hipSYCL uses HIP as backend for AMD GPUs and CUDA for NVIDIA GPUs. Finally, we developed the Alpaka backend for BabelStream [37].

Table 3. List of compilers

Compiler/Software	Version	System
AOMP	13.0-4-4	AMD accelerator cloud
LLVM	13.0.0	AMD accelerator cloud
ROCM/HIP	4.2	AMD accelerator cloud
NVIDIA HPC SDK	21.7	Puhti, Mahti
GCC	devel/omp/gcc-10 (6b88ea4)	Puhti
hipSYCL	0.9.1 (c759aac1d)	Puhti, Mahti, AMD accelerator cloud
Kokkos	3.4.1	Puhti, Mahti, AMD accelerator cloud
Alpaka (cupla)	commit 287deace	Puhti, Mahti, AMD accelerator cloud

Copy Kernel

Figure 3 demonstrates the results from the Copy kernel of the BabelStream across many programming models. On the x-axis are the names of the programming models and on the y-axis the bandwidth in GB/s is depicted. However, as we plot the boxplots, we split the y-axis in order to be able to visualize the plots more clearly. On each y-axis range, there are results only from a specific GPU whose name is mentioned on the right y-axis. In Table 4, we present in the three first rows the peak performance (in %) based on the best programming model and the last two rows demonstrate the percentage range of the differences, either slower for MI100 vs A100 or faster for MI100 vs V100. We observe that for all the GPUs, the HIP/CUDA programming models achieve the highest performance. Although the OpenMP performance seems to be close to the CUDA's one, thanks to efficient NVHPC compiler, the AMD OpenMP based on LLVM is not performing similar to HIP for this pattern. The default Kokkos implementation in BabelStream does not provide tuning options, however, we observe that for MI100 its results are close to not tuned HIP. Finally, hipSYCL has a variation on A100 which is less than 2%, and Alpaka achieves a performance similar to HIP for the MI100.

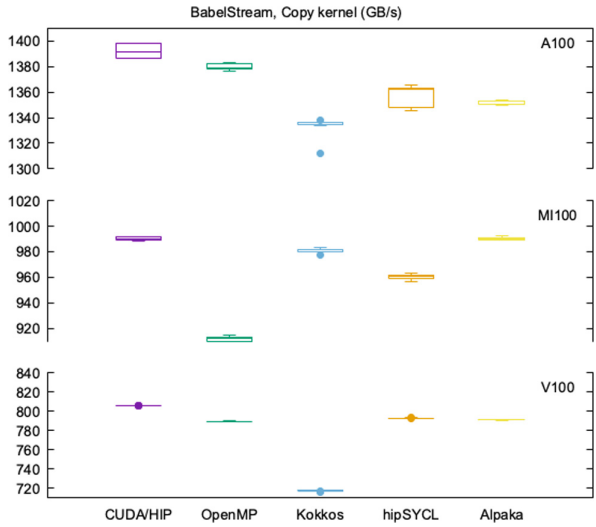

Fig. 3. Results of BabelStream for copy kernel across the programming models on AMD MI100, and NVIDIA V100/A100

Table 4. Copy kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	98.5	96.3	97.53	97.3
MI100	100	92.6	99	97.8	99.99
V100	100	97.95	89.01	98.4	98.2
MI100 slower than A100	28.5–29.25	33–34	25.29–26.79	28.74–29.79	26.14–26.78
MI100 faster than V100	22.5–23	14.9–15.9	36.2–37.1	20.67–21.42	25.75–25.93

Multiply Kernel

We plot the results for the Multiply kernel in Fig. 4 and present the peak performance and the comparison in the Table 5. For MI100, most of the programming models achieve 96.63% and above except OpenMP, while for A100 all the programming models perform close to the peak, however, for V100, the not tuned Kokkos underperforms. For MI100, most programming models perform 21.6–37.8% faster than V100, except OpenMP, and similarly, MI100 is 25–31% slower than A100. For some cases there is variation up to 4.5% on A100 where for the moment we have not identified a specific reason as for all the cases we use a dedicate single node. However, it could be considered as execution variation.

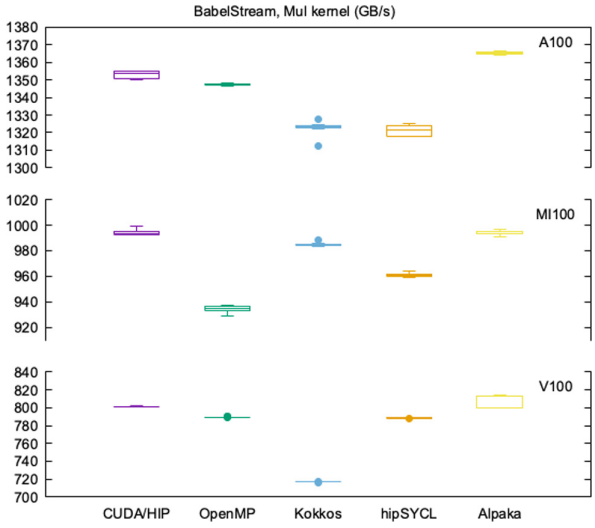

Fig. 4. Results of BabelStream for mul kernel across the programming models on AMD MI100 and NVIDIA V100/A100

Table 5. Multiply kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.59	97.75	97.63	99.99
MI100	100	93.98	99.07	96.63	99.95
V100	100	98.54	89.52	98.39	99.90
MI100 slower than A100	26.16–26.78	30.4–30.98	24.92–25.79	26.99–27.57	26.96–27.44
MI100 faster than V100	23.8–24.74	17.8–18.6	37.0–37.8	21.6–22.22	21.74–24.52

Add Kernel

We plot the results for the Add kernel in Fig. 5 and present the peak performance and the comparison in the Table 6. The performance of Alpaka programming model is quite close to HIP/CUDA for all the devices with hipSYCL following, and the OpenMP is less efficient on the MI100 compared to the rest GPUs. Finally, Kokkos, seems to be between Alpaka and hipSYCL, regarding the performance.

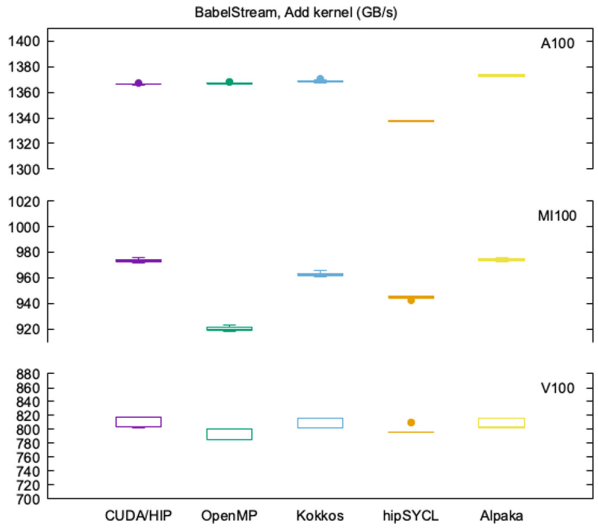

Fig. 5. Results of BabelStream for add kernel across programming models on AMD MI100 and NVIDIA V100/100

Table 6. Add kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.80	97.6	97.70	99.99
MI100	100	94.52	98.92	97.06	99.99
V100	100	97.91	99.90	98.93	99.93
MI100 slower than A100	28.55–28.86	32.46–32.86	29.44–29.78	29.27–29.54	28.91–29.19
MI100 faster than V100	18.94–21.55	14.74–17.58	17.76–20.08	16.55–18.88	19.20–21.60

Triad Kernel

We plot the results for the Triad kernel in Fig. 6 and present the peak performance and the comparison in the Table 7. For this kernel Alpaka performs equally to HIP/CUDA, with following Kokkos, and then hipSYCL and OpenMP.

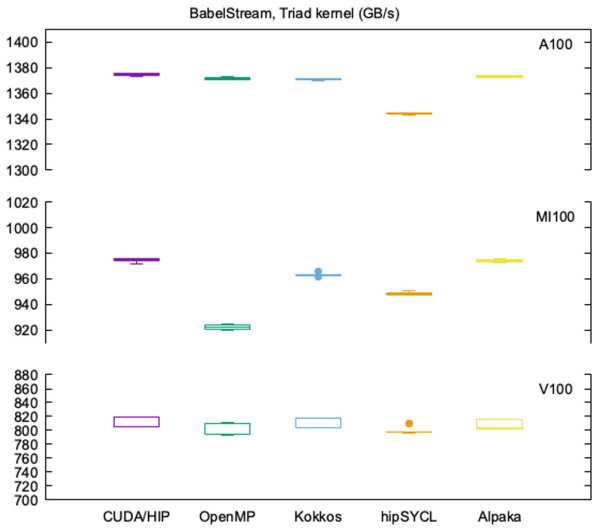

Fig. 6. Results of BabelStream for triad kernel across programming models on AMD MI100 and NVIDIA V100/A100

Dot Kernel

We plot the results for the Dot kernel in Fig. 7 and present the peak performance and the comparison in the Table 8. In the first segment of V100, we have also a

Table 7. Triad kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	99.78	99.72	97.77	99.89
MI100	100	94.62	98.83	97.30	99.94
V100	100	98.82	99.86	98.80	99.69
MI100 slower than A100	28.93–29.31	32.6–32.81	29.52–29.87	29.21–29.57	28.91–29.19
MI100 faster than V100	18.63–21.25	13.47–16.40	17.79–20.13	17.00–19.25	19.02–21.61

plot of MI100 for hipSYCL as they were too close these values. MI100 GPU is around 2.69–14.62% faster than NVIDIA V100 except for OpenMP, and 28.00–69.69% slower than A100. The OpenMP on MI100 does not perform efficiently for the reduction pattern, and Kokkos is not optimized but it is quite close to non-optimized HIP version. For HIP/CUDA; We define 216 blocks of threads for the A100, as it has 108 streaming multiprocessors, and improved the *dot* kernel performance by 8–10%. For hipSYCL; we utilize 960 blocks of threads and 256 threads per block to achieve around 8% faster than the default values. For Alpaka; we are able to tune also the *dot* kernel with 720 blocks of threads for MI100 and improve its performance by 28% comparing to the default settings.

Table 8. Dot kernel results, percentage peak and comparison (%)

	HIP/CUDA	OpenMP	Kokkos	hipSYCL	Alpaka
A100	100	96.77	84.43	94.63	98.52
MI100	100	42.16	74.40	89.53	99.99
V100	100	96.5	72.87	96.95	99.38
MI100 slower than A100	29.09–29.84	68.7–69.69	37.19–38.43	32.67–33.65	28.00–28.62
MI100 faster than V100	10.80–11.86	–51.06 - (–51.513)	12.60–14.62	2.69–3.38	11.80–12.59

Summary

We can observe that based on the hardware performance AMD MI100 performs faster than NVIDIA V100 and slower than NVIDIA A100. The peak bandwidth percentage for the OpenMP programming model is 42.16%–94.68% for MI100 while it is at least 96% for the NVIDIA GPUs which demonstrates that AOMP needs further development. For Kokkos, the range is 74–99% for MI100, 72.87–99% for the NVIDIA GPUs, where the non-optimized version has lower

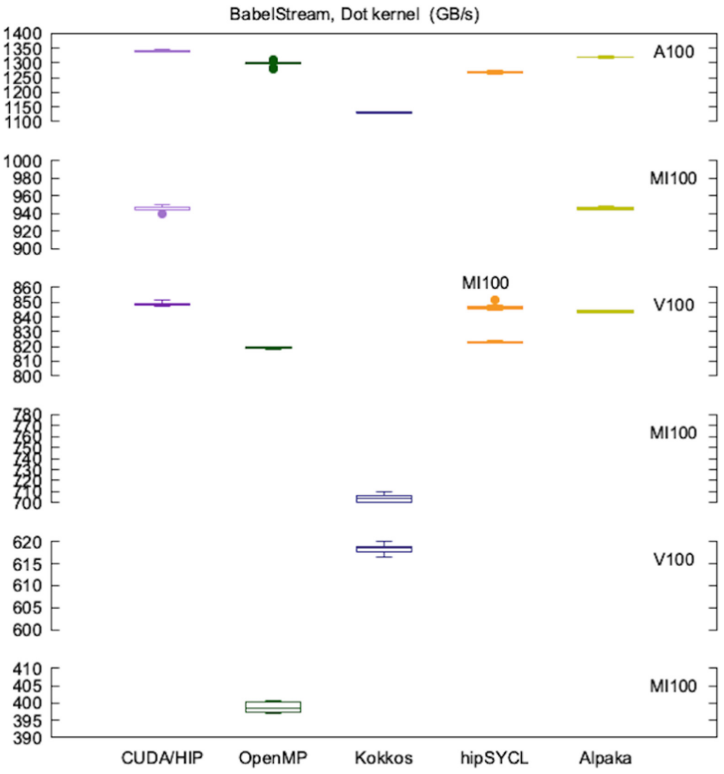

Fig. 7. Results of BabelStream with Kokkos HIP backend on AMD MI100 and NVIDIA V100/A100

performance mainly on MI100 and V100. hipSYCL achieves at least 96% of the HIP/CUDA performance except for MI100 and *dot* kernel that achieves 89.53%. Finally, Alpaka achieves at least 97.2% for all the cases and demonstrates its performance. The OpenMP compiler performs better for NVIDIA GPUs regarding *dot* kernel, however, the version that we used for AMD GPUs, is not the final product yet. The Kokkos results regarding the *dot* kernel are not optimized, we did not modify the execution policy and, we tried to use the default code and change only specific values, thus the low percentage. Also this demonstrates that some tuning are not so straight forward for Kokkos. Overall, the Alpaka performance is quite close to HIP, followed by hipSYCL and Kokkos, while OpenMP can perform slower depending on the kernel. We have to mention that all the remaining programming models except the OpenMP utilize HIP or CUDA as backend. The OpenMP Offloading for AMD GPUs has a potential to improve in the future as it is under development.

Finally, we should mention that we can observe that various programming models could have similar performance on the same GPU with some variation except OpenMP for some cases and Kokkos because is not optimized for some specific

cases. Overall, the tuning is not difficult if the developer is aware of the architecture and the programming model. Also, the utilization of each programming model depends on the experience of the developer, and the programming language as it was presented in the porting workflow.

7.3 MiniBUDE

Large problem sizes for miniBUDE are required to be able to saturate the GPUs. For every experiment, we execute 8 iterations with 983040 poses. We calculate this number by tuning for AMD MI100, however, this value achieves peak performance on the NVIDIA GPUs also with minimal variance of 1–2% and we decided to use the same workload for all the devices. The miniBUDE provides in the output the single precision GFLOP/s, and we observe in the Fig. 8 that the AMD MI100 GPU achieves a performance close to A100 by 2% and around 26% over V100. As the benchmark does not use tensor cores or other features, the peak performance is based on the FP32 capabilities of the GPUs. Thus, AMD MI100 is on average 1.25 times faster than NVIDIA V100, and 0.018 times slower than NVIDIA A100 for single precision using miniBUDE. For the moment, the other programming models do not perform very well, and we are still investigating the reasons. The code varies a bit between the programming models and the performance is significantly worse, thus we can not identify yet why both hipSYCL and Kokkos perform lower than HIP while using HIP for backend. Moreover, the Alpaka version for miniBUDE is under preparation. Regarding single precision, we tested also the mixbench [38,39] benchmark and the MI100 was 1.16 times faster than A100, achieving both close to their peak performance.

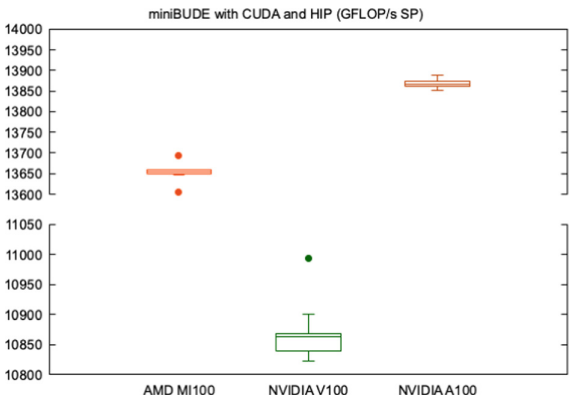

Fig. 8. Results of miniBUDE on various GPUs for HIP and CUDA

8 Conclusion and Future Work

In this paper, we present a methodology for porting applications to LUMI supercomputer, an AMD GPU-based system. As we expect many users to utilize LUMI, we are getting ready for a variety of porting scenarios. We benchmark various programming models to understand how they perform, how efficient they are, and which ones to propose to our future users. Thus, we do a performance comparison between AMD MI100, NVIDIA V100, and A100. We utilize a benchmark and a mini-app, which are memory and compute-bound respectively. We illustrate how various programming models perform on these GPUs and what techniques can improve the performance for specific cases. We discuss the lack of performance on some aspects of OpenMP, how to tune some programming models based on the targeted hardware and we verify the results. Moreover, the single precision mini-app demonstrates how similar performance to NVIDIA A100 has the AMD MI100 when not utilizing tensor cores. Overall, HIP/CUDA perform quite good and most of the programming models are quite close, depending on the kernel pattern. Depending on your experience, the programming language, and the kernel, you could leverage many of the programming models and always compare with the peak performance. Finally, programming models such as Alpaka and hipSYCL could be utilized as they support many backends, are portable and for many kernels they provide similar performance to HIP. All the scripts and the results are provided in [40] for reproducibility purposes.

For future work, we plan to identify what the issue with some programming models and miniBUDE is. We want to analyze the OpenACC performance from the GCC and Cray Fortran compiler, amongst tuning further the programming models, and to test the new functionalities from the ROCm platform such as Heterogeneous Memory Management (HMM). We envision evaluating multi-GPU benchmarks and their scalability across multiple nodes. By using LUMI we will be able to use the MI250X GPU and compare it with the current GPU generation. Finally, we are interested in benchmarking I/O from the GPUs memory as we have already hipified Elbencho [41] benchmark and a few applications could have significant I/O bottlenecks but we plan to benchmark them when LUMI is available as its architecture is much different to the available systems.

Acknowledgement. We want to thank CSC - IT Center for Science Ltd. for the access to Puhti and Mahti supercomputers. Tomas Tobias from Siemens for the discussions about GCC and LLVM. Finally, thank to Simon McIntosh-Smith and Wei-Chen Lin from University of Bristol for providing the necessary files to create new input problem sizes for miniBUDE.

This work was partly funded by the Center for Advanced Systems Understanding (CASUS) that is financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

References

1. CSC LUMI supercomputer. https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/
2. Frontier web page. https://www.olcf.ornl.gov/frontier/
3. NVIDIA. CUDA. https://developer.nvidia.com/about-cuda
4. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. In: Computing in Science & Engineering, vol. 12, no. 3, pp. 66–73, May-June 2010. https://doi.org/10.1109/MCSE.2010.69
5. OpenMP Architecture Review Board. OpenMP Application Programming Interface, version 4.0. https://openmp.org/40pdf
6. OpenACC Specification 3.0. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf
7. Davis, J.H., Daley, C., Pophale, S., Huber, T., Chandrasekaran, S., Wright, N.J.: Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In: Bhalachandra, S., Wienke, S., Chandrasekaran, S., Juckeland, G. (eds.) WACCPD 2020. LNCS, vol. 12655, pp. 25–44. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-74224-9_2
8. Poenaru, A., Lin, W.-C., McIntosh-Smith, S.: A performance analysis of modern parallel programming models using a compute-bound application. In: 36th International Conference, ISC High Performance 2021, Frankfurt, Germany (2021)
9. Khalilov, M., Timoveev, A.: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In: Journal of Physics: Conference Series, vol. 1740 (2021)
10. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of HPC-style SYCL applications. In: Proceedings of the International Workshop on OpenCL (2020)
11. Deakin, T., et al.: Performance portability across diverse computer architectures. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–13 (2019). https://doi.org/10.1109/P3HPC49587.2019.00006
12. AMD. ROCm Platform. https://github.com/RadeonOpenCompute/ROCm
13. AMD. ROCm Documentation. https://rocmdocs.amd.com/en/latest/
14. AMD. HIP. https://github.com/ROCm-Developer-Tools/HIP
15. AMD. HIPify Tools. https://github.com/ROCm-Developer-Tools/HIPIFY
16. AMD. HIP Porting Guide. https://github.com/RadeonOpenCompute/ROCm_Documentation/blob/master/Programming_Guides/HIP-porting-guide.rst
17. CSC. Porting GPU Codes to HIP. https://github.com/csc-training/hip
18. de Supinski, B.R., et al.: The ongoing evolution of OpenMP. In: Proceedings of the IEEE, vol. 106, no. 11, pp. 2004–2019, November 2018
19. Khronos Group. SYCL 2020 Specification. https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf
20. Codeplay Software. ComputeCpp. https://www.codeplay.com/solutions/ecosystem/
21. Intel Corporation. SYCL* Compiler and Runtimes. https://github.com/intel/llvm
22. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: the architecture, current state and future direction of hipSYCL. In: Proceedings of the International Workshop on OpenCL (IWOCL 2020), Association for Computing Machinery, New York, Article vol. 8, no. 1 (2020). https://github.com/illuhad/hipSYCL
23. triSYCL. https://github.com/trisycl/trisycl

24. ORNL and Mentor Graphics. https://www.olcf.ornl.gov/2020/09/03/oak-ridge-leadership-computing-facility-fosters-gcc-compiler-development-with-mentor-contract/
25. Denny, J.E., Lee, S. and Vetter, J.S.: Clacc: translating OpenACC to OpenMP in clang. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC. LLVM-HPC), Dallas, TX, USA (2018)
26. Clacc. https://github.com/llvm-doe-org/llvm-project/tree/clacc/main
27. Zenker, E., et al.: Alpaka-an abstraction library for parallel kernel acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 631–640, May 2016
28. Bussmann, M., et al.: Radiative signature of the relativistic kelvin-helmholtz instability. In: SC 2013: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2013)
29. René, W., Sergei, B., Simeon, E., Jeffrey, K., Jan, S.: Cupla - C++ User interface for the Platform Independent Library alpaka. https://rodare.hzdr.de/record/1103
30. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parall. Distrib. Comput. **74**, 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
31. AMD. hipfort. https://github.com/ROCmSoftwarePlatform/hipfort
32. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: GPU-STREAM v2.0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: Paper presented at P^MA Workshop at ISC High Performance, Frankfurt, Germany (2016). https://doi.org/10.1007/978-3-319-46079-6_34
33. Tom, D., Simon, M.-S.: BabelStream. https://github.com/UoB-HPC/BabelStream
34. miniBUDE. https://github.com/UoB-HPC/miniBUDE/
35. CSC. Puhti Supercomputer. https://docs.csc.fi/computing/systems-puhti/
36. CSC. Mahti Supercomputer. https://docs.csc.fi/computing/systems-mahti/
37. CUPLA BabelStream Fork, v3.4-alpaka release https://github.com/jyoung3131/BabelStream/releases/tag/v3.4-alpaka
38. Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. J. Parall. Distrib. Comput. **107**, 37–56 (2017)
39. Mixbench. https://github.com/ekondis/mixbench
40. Reproduce the results of the paper Evaluating GPU Programming Models for the LUMI Supercomputer. https://zenodo.org/record/6307447
41. Elbencho. https://github.com/breuner/elbencho