Best Practice Guide LUMI

Jussi Heikonen, CSC, Finland

Georgios Markomanolis, CSC, Finland

Cristian-Vasile Achim, CSC, Finland

Ezhilmathi Krishnasamy, University of Luxembourg, Luxembourg

Abdulrahman Azab, University of Oslo, Norway

Pedro Ojeda-May, HPC2N, Umeå University, Sweden

Michele Martone, LRZ, Germany

Marcin Krotkiewski, University of Oslo, Norway

Hicham Agueny, University of Bergen, Norway

Maria Guadalupe Barrios Sazo, University of Oslo, Norway

Ole Widar Saastad (Editor), University of Oslo, Norway

12 January 2023

Table of Contents

1. Introduction

The pre-exascale system LUMI is hosted by CSC IT Center for Science [1] and is located at Kajaani, Finland. While LUMI is an acronym (Large Unified Modern Infrastructure) it is also the Finish word for snow.

At the time of writing the LUMI [2] system ranks as number 3 on the Top500 list [3].

Figure 1. The LUMI pre-exascale supercomputer

LUMI [2] is well described in detail by the LUMI consortium [4]. A short summary is presented below.

The system was delivered by Hewlett Packard Enterprise (HPE) and is an HPE Cray EX supercomputer. The committed Linpack (HPL) performance of the GPU partition of LUMI is 375 Petaflops/s coming from AMD Instinct™ MI250X GPUs.

The system has various partitions of which LUMI-C (x86 partition) and LUMI-G (accelerator partition) are the most important. Other partitions, include storage and other services see [6].

As LUMI is powered by GPUs many codes may need to be adapted to take advantage of the accelerators. As the GPUs are from AMD the software tools are somewhat different from NVIDIA based systems. These and related issues are discussed in the LUMI documentation [5].

Access to LUMI is granted by members of the consortium. LUMI access is documented in [7].

2. Architecture

2.1. Introduction

LUMI contains a number of partitions with different hardware and purpose. The most common ones are :

- LUMI-C : the CPU partition
- LUMI-G : the GPU partition
- LUMI-P : the LUSTRE storage partition
- LUMI-F : the flash storage partition

Other partitions exist but are not covered in this guide.

2.2. LUMI-C: CPU partition

Each compute node in LUMI-C has two 64-core AMD Zen3 CPUs (EPYC 7763, 2.45 GHz, boost up to 3.5 GHz) and at least 256 GiB of memory. The nodes in LUMI-C are connected with a 200 Gbits/s HPE-Cray Slingshot-11 interconnect. Please consult the PRACE Best Practice Guide on Modern Processors for in depth information on AMD processors [21] chapter 4.

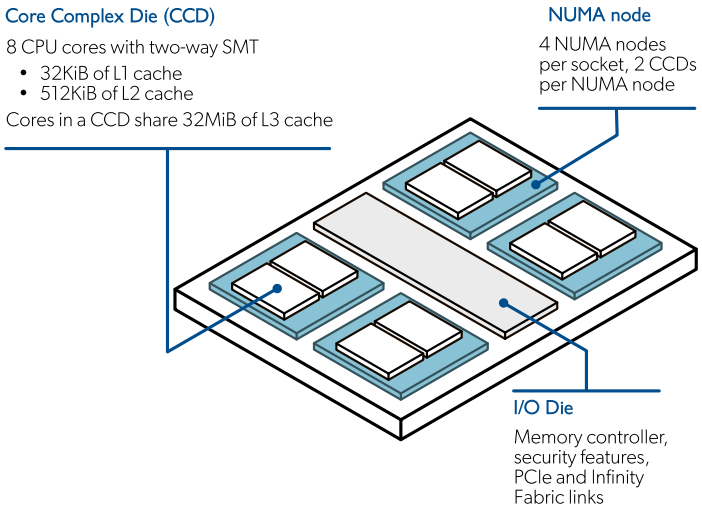

Figure 2. AMD Milan Overview

The EPYC 7763 CPU cores support two-way simultaneous multithreading (SMT) allowing up to 256 threads per node and 256-bit AVX2 vector instructions for a maximum throughput of 16 double precision floating point operations per clock cycle. Each CPU has 8 core complex dies (CCDs) that each contain 8 cores. The nodes are configured in 4 NUMA zones with 2 CCDs each.

Each core has a 32 kiB instruction cache, a 32 kiB L1 cache, a 512 kiB L2 cache, and a shared 32 MiB L3 cache. The L3 cache is shared between the 8 cores in a CCD for a total of 256 MiB of L3 cache in a CPU. Each node has also at least 256 GiB of shared memory (up to 1024 GiB on some nodes) accessible to all 8 CCDs using 8 memory channels with a peak theoretical bandwidths of 204.8 GB/s per socket. The two CPUs within a node are connected by 4 bi-directional links with a peak bandwidth of 256 GB/s.

The cores in a modern processors are very complex, there is a number of executional units within each one: decoding, load/store, integer, floating-point, and vector units, just to mention a few. For some of these even more than one. To take advantage of this, the core has a mode where it can run multiple streams of instructions, simultaneous multi threading (SMT) [42] or Hyper Threading (HT) [43]. In principle more than one stream can run simultaneously if they do not use overlapping executional units. However, in most cases they are competing for the resources.

In practical terms, the SMT manifests itself by providing a set of virtual cores. In the case of selecting SMT-2 mode twice as many and for SMT-4 four times. Seen from the operating system's point of view (which is also the user's view), it looks like we have 2x or 4x number of cores to use. Most tools and queries to the OS report the number of virtual cores. It takes a closer look to reveal if the cores are virtual or not[1].

The mentioned larger number of cores is somewhat deceiving as the executional units are the same. For diverse load with a wide spectrum of different applications and processes this can be beneficial. It can also help to hide both memory and I/O wait as the other instruction streams can continue without any context switching. This has been less beneficial for HPC workloads where the instruction streams are normally equal as MPI jobs tend to run multiple copies of the same executable and OpenMP created multiple identical threads from a parallised loop. Hence the instructions streams tend to request the same executional units of which there a fixed amount of regardless of how many virtual cores there are. The best practice for HPC is setting SMT to off or just schedule one thread per core.

The system is a non-uniform memory access (NUMA) architecture where memory access time differs depending on which NUMA bank is being accessed from which CPU. The following test is performed using the Stream benchmark [54] locking the execution to the single core number 0 core and a specified NUMA bank using numactl. NUMA banks 0 to 3 are attached to the first socket while NUMA banks 4-7 are attached to the second socket. Each NUMA node has 64 GiB of memory.

The record is recorded using the command:

```
for j in $(seq 0 7); do numactl -m $j --physcpubind=0 ./stream; done
```

Table 1. Single core NUMA memory bandwidth

NUMA node	Copy [GB/s]	Scale [GB/s]	Add [GB/s]	Triad [GB/s]
0	42.2	42.9	41.1	41.6
1	42.2	43.0	40.9	41.3
2	41.4	42.3	39.9	40.4
3	41.1	42.1	39.7	40.1
4	29.8	29.7	26.8	26.9
5	29.7	29.6	26.6	26.8
6	30.0	29.8	27.0	27.1
7	28.3	28.2	25.2	25.3

The numbers above clearly quantify why processor binding, memory and processor placement are important. They also show the penalty for accessing NUMA banks from another socket.

More information is available in the LUMI documentation [5] and in the AMD product specification [8].

2.3. LUMI-G: GPU partition

Each compute node in LUMI-G has one 64-core AMD Trento CPU, four AMD Instinct MI250X GPUs, and 512 GiB of memory. The nodes in LUMI-G are connected with a 200 Gbits/s HPE-Cray Slingshot-11 interconnect.

[1] The command lscpu emit processor info, look for «Thread(s) per core:»

The MI250X GPUs are based on the AMD CDNA 2 architecture [9] and offer a theoretical peak performance of 95.7 TFLOPS (double precision) per GPU. Each GPU consists of two graphics compute dies (GCDs) with 110 compute units each for a total of 220 compute units per GPU. Since each compute unit has 64 stream processors, one GPU has a total of 14080 stream processors.

Each GPU has a 128 GiB HBM2e memory with a peak memory bandwidth of 3.2 TB/s. Within the GPU, an in-package Infinity Fabric interface delivers 400 GB/s theoretical peak bi-directional bandwidth between the GCDs. Furthermore, each node in LUMI-G contains 4 x 200 Gbits/s network adapters.

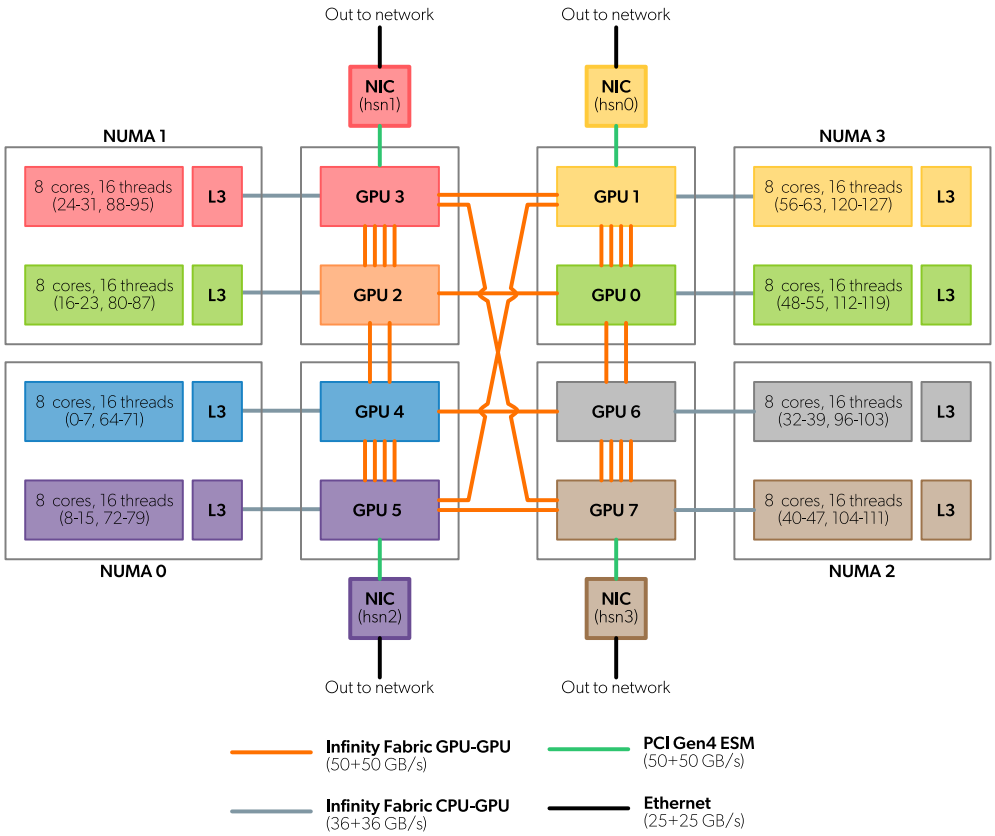

Figure 3. LUMI-G node architecture

Some measured bandwidth numbers are available from tests at Jülich research center in Germany [12].

More detailed information is available in the LUMI documentation [5], in the AMD MI250™ product specification [10], and in the AMD CDNA 2 Architecture white paper [11].

2.4. LUMI-P: LUSTRE storage

The main parallel disk system on LUMI, called LUMI-P, is a Cray ClusterStor E1000 Lustre solution, with 80 PB in total. It is split in 4 different parallel filesystems for safety, so that one user can not harm the filesystem for all the users as the projects are scattered among the different filesystems. For the initial aggregated filesystem, the peak performance was 960 GB/s for one file per MPI process but for each of the four new filesystems it should be close to 240 GB/s. Each filesystem has one metadata module and 8 data storage modules. Each data storage achieves

30 GB/s and is constituted by 212 HDD of 16 TB each and there are 2 storage servers for every module. Thus, we have 32 OSTs per filesystem. There are 2 storage servers per metadata module with 24 NVMes of 3.2 TB each.

The default striping is 1 which is good for many small files but as we increase the file size we can start increasing the striping. One rule of thumb is that we can use the square root of the file size in GB for number of OSTs, thus for a 100 GB file we could use 10 OSTs. For example, if a directory is called examples, we define the number of the OSTs as follows: as:

```
lfs setstripe -c 10 examples
```

Moreover, another restriction is the interconnect, and it depends on how many processes and how many interconnect ports are used for the I/O. The maximum speed per LUMI node is 100 GB/s, and to achieve this performance, we need to use enough OSTs. Considering that the peak is 240 GB/s for 32 OSTs, we need to use at least 14 OSTs for 100 GB/s, always for one file per MPI process. In practice scaling is not ideal and there are other jobs running. The actual numbers will differ. Some testing is needed to explore the performance space.

2.4.1. Performance in practice

Performance for a normal user for a single instance of the well-known I/O test Iozone [13] is shown in the table below. The commands for the tests (Cached, Memory mapped, Synchronously or Direct) are as follows:

```
./iozone -r 64 -s 16g -i 0 -i 1 -i 2 -f ./iozone.tmp
./iozone -B -r 64 -s 16g -i 0 -i 1 -i 2 -f ./iozone.tmp
./iozone -o -r 64 -s 16g -i 0 -i 1 -i 2 -f ./iozone.tmp
./iozone -I -r 64 -s 16g -i 0 -i 1 -i 2 -f ./iozone.tmp
```

Table 2. IOZone performance

Access	Sequential		Random	
File access type	Write [MB/s]	Read [MB/s]	Write [MB/s]	Read [MB/s]
Cached	1581	10512	534	7299
Memory mapped	521	3143	340	2661
Synchronously	27	9021	21	6622
Direct	31	54	23	4

The testfile is far less (only 16 GB) than what is normally used for storage benchmarking (rule of thumb is 2x installed memory), but illustrates what a user can expect for a normal run with a reasonably sized file with LUMI-P. The performance differs greatly between cached, synchronously or direct. There are multiple methods to arrange IO when writing applications. Choosing an optimal access with your application is important. The standard cached IO is by far most common and are normally the default.

2.5. LUMI-F: LUSTRE Flash storage

The flash file system, called LUMI-F is also based on Lustre and is about 7 PB of NVMe SSDs. Using all the SSDs, the file system the file system can achieve an aggregate bandwidth of about 1740 GB/s with one file per MPI rank. The data OST are constituted by 2 nodes each, with 24 NVMe storage elements of 15.36 TB and can perform 60 GB/s each.

This filesystem should be used for codes with many small files with irregular access patterns and/or a lot of metadata operations as they would perform significantly faster than on LUMI-P. Also, as we have a peak of around 240 GB/s on each filesystem on LUMI-P, it would benefit for large files with fast I/O operations.

LUMI-F is of limited size. Users should plan usage in order not to waste this limited resource. LUMI-F is not a place for long term storage.

3. Application Programming

3.1. Porting code to GPU accelerators

Developing applications requires knowledge of the architecture and software stack in addition to parallel programming environment and models. While this guide is not a replacement for local documentation it tries do convey information that has been tested and a verified as a good foundation. Each application requires special attention and only in-depth knowledge of the current application can tailor it to the used system.

As LUMI contains a fair share of GPU accelerators, many applications will want to exploit the huge processor power that comes with the GPUs. In most cases this means adapting legacy codes to acceleration. While this guide is not a guide about porting legacy codes it does nevertheless discuss some issues before embarking on adapting codes for acceleration.

Porting scientific codes to take advantage of an attached accelerator like a GPU (Graphics Processing Unit) is not a trivial task. As already given in the name it still is a unit designed to do processing for displaying graphics. Althrough newer models have matrix units tailored for machine learning and lack ray tracing units. However, accelerators like AMD GPUs (but also NVIDIA H100, A100 etc) which have an optimised/adapted design for scientific computation (NVLink, more amount of computing units, more modules to do graphics).

The highly parallel nature of this kind of processor is very different from a common general purpose CPU. The main difference from CPU is the core approach of the GPU processing [14]. Terminology has been inspired from weaving, as similarities to the huge number of threads in the warp [15] (just pause for a moment and think about a break, test or jump in a program is comparable to a knot in one of the threads in a loom, some issues can be overcome by masking instructions with mask excluding division by zero etc).

For selected algorithms and codes written for GPUs excellent performance can be achieved, approaching the quoted performance numbers. The GPU's internal memory bandwidth is superior to the CPU memory bandwidth and is partially responsible for the huge performance within the accelerator.

For some scientific applications a massive number of vector operations with a lot of floating-point operations per unit of data, large performance gains can be experienced. This is typical for well structured codes with loops with constant stride and regular memory access patterns. For other types of applications where vector type operations or structured code are less predominant the performance gain might be small or even a slowdown, hence the GPUs are only an option for a subset of scientific codes.

Care must be taken during the initial analysis phase to verify that the application selected for accelerator porting is suitable. This commercial blog entry about selecting codes contain an overview of issues to address [16], or another from Better Scientific Software [17].

A good introduction about performance and memory bandwidth is given in Chapter 3 in the Best Practice Guide Modern Accelerators [22].

Memory usage and access is often a major issue with large scale scientific applications. Recording a roofline model of the application will in most cases show that the application is memory bound. The accelerator has far superior memory bandwidth compared to the CPU. However, the memory bandwidth between host memory and device memory is limited. This is well-known with CPUs, where it is handled by the cache hardware. With accelerators the scales are different and data movement is mostly under manual control (Unified memory [25] leaves the control of this to the libraries). The guide above gives a nice insight into the usage of the «roofline Model», both on memory bandwidth and compute capacity.

Cache coherent shared memory. The Infinity fabric facilitate all memory, both main and device memory as shared memory with full cache coherence. This represent a major step forward in ease of use and performance. Synchronisation and data transfer can now rely on the memory system, thus making it simpler and more efficient. Far less effort is needed from the programmer who no longer need to administrate and control data placement.

The old wisdom that do as much work as you possibly can while data is in the cache applies even more with accelerators. How much work can be done by the accelerator with the data that has been already transferred? In

addition we know the compute unit is very fast and with a very high bandwidth comes the question: can we now do more computation ? With the data residing in the accelerator memory, is it possible to do finer grid, shorter time steps? Or to introduce a new science model that intrinsically requires more computation ? Given the high peak performance of the accelerator this question should be given consideration.

The floating-point precision is also an issue: can some calculations be done in 32 bit precision and the final reduction in 64 bit precision, or even all computations in 32 bit precision ? The matrix core units are limited to 16 and 32 bits precision, (MI250X support 64 bit data types in the matrix engine [26]). The performance is impressive: one 4x4 matrix multiplication per clock cycle.

Many tools and libraries allow the accelerator to be addressed without fundamentally changing the code. Compiler directives is in widespread use (OpenACC, OpenMP), accelerated library functions, and language features like modern fortran with do concurrent (which some compilers can translate to accelerated code). External functions for common tasks like linear algebra and Fourier transform are commonly available for accelerators. This is fairly trivial to implement and should always be considered.

Porting the legacy code in order to have the compiler help to offload loops or even part of the code is another option. Offloading loops can be done using compiler directives of which two are currently in widespread use, OpenACC and OpenMP. The latter is well-known for parallelising loops over both vector units (SIMD-OpenMP) and multi-threads using multiple cores. There are different philosophies behind these variants for offloading. OpenACC does parallelisation hiding many details away, while OpenMP allows developers to have more control. It is worth reviewing the documentation before selecting one or the other.

Both OpenACC and OpenMP offer offloading of loops and mechanisms to do data transfer, all potentially without changes to the program source, but just via pragma directives inserted as comments in the source code. This ensures portability with old and future versions of compilers and systems, because the directives can be ignored or utilized, based on a compiler switch.

Coverage of OpenACC and OpenMP (offload) in depth is beyond the scope of this guide. There are multiple sites on the web who offer documentation and training of these standards OpenACC [57] OpenMP [59].

3.1.1. Hardware limitations

Today, the GPU is an attached processor which sits on the motherboard connected to the rest of the node via a link with superior performance compared to PCIe called Infinity fabric [18]. Each link accommodates a bandwidth of 100 GB/s in each direction. Using several links and bidirectional transport high bandwidth rates can be achieved, for LUMI 400 GB/s [19]. While data still has to be moved from main CPU memory to the device GPU memory the higher performing fabric mean less time is wasted on data movement.

Because moving data to and from an accelerator memory is a bottleneck, it is of major importance to do as much work a possible on the accelerator when data is present in accelerator memory. Any reuse of data must be done as moving data back to and from the main memory is so costly (also with respect to the programmers effort to control it). Large GPU kernels doing as much computation in the data as possible are of help. This might initiate a code rewrite or even change of algorithm.

Limitation of main versus device memory, the main memory of a compute node is in the range of 1 to 8 GB per CPU core, or in the 128 GB to multi TB range for nodes. The accelerator memory is generally far less, in the order of 40-128 GB. Most of the time, only a fraction of the main memory. This limits the ability to fit entire models or complete data structures in the accelerator memory. Multiple cards can be used, but each memory is still separated. Taking advantage of the shared memory option makes it possible to run very large data sets with the GPU, but accessing data residing in other memory banks is less efficient and performance might suffer.

3.1.2. Task based approach

Another approach in OpenMP is task-based offloading. It offers the possibility of offloading so-called tasks. This usually requires writing some more code than with the original loops, and hence requires some development effort. Given that the amount of computation within each task and its associated data is expected to be greater than for the loop-based approach, the potential for less movement of data and more computation is far higher for task based offload than loop offloading. However, it incurs in the cost of re-factoring the code and extra development, and

all the issues usually related with code changes. The latter might be more problematic than one thinks, as the program codes tend to live for decades and any deviation from a streamlined approach (which the compiler easily «understands») might cause problems down the line in the future.

This will be discussed and addressed in later sections, starting at Section 3.2, “GPU programming models”.

3.1.3. Language considerations

While CUDA [63] is regarded as some kind of standard for low level GPU programming, it's a framework from NVIDIA which only run on hardware from this vendor. An alternative to CUDA offered by AMD is HIP, with additional tools like HIPify to convert CUDA to HIP code, for more on this see Section 3.2.4.

While OpenCL [34] remains some kind of standard for portable GPU programming it is not easy to use nor very user friendly and have never seen widespread use in HPC for scientific applications. It is a language that offers portability combined with outstanding performance between different GPUs. Portability is important when codes are to be moved between HPC systems. For more on OpenCL see Section 3.2.6.

SYCL [35] have gained more and more popularity over OpenCL. SYCL is covered in Section 3.2.7.1.

Current compilers support the common HPC languages Fortran, C and C++, along with OpenMP [59] and OpenACC [57] directives. Some of the codes in dusty decks run today are many decades old and will continue to run for decades to come. Hence the reluctance by many to do major rewrites just to accommodate for today’s attached accelerators. Taking the chance of predicting the future one can speculate that the attached accelerators may become part of the processor just like the arithmetic units for floating-point and vector units became parts of the CPU, and therefore future compilers will maybe parse legacy code and generate suitable instructions to exploit GPUs. Compiler directives that do not require source code changes are far less intrusive in the source code, and therefore much more popular.

For new development free from the restrains of the legacy languages there is a large selection of languages. One language in particular, Julia [30], has set out to replace Fortran and has the potential to succeed.

Julia also strives to support GPUs more closely. The initiative JuliaGPU [31] is one such approach. See Section 3.2.5 for more on Julia and GPUs.

3.2. GPU programming models

The Best Practice Guide for Modern-Accelerators [22] is also a good source of information to GPU programming.

3.2.1. OpenACC offload

In this section, we provide a short introduction to OpenACC. We consider both C/C++ and Fortran programming languages here.

OpenACC is a user-driven directive-based performance-portable parallel programming model. It is designed for scientists and engineers interested in porting their codes to a wide variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a lower-level model. The OpenACC specification supports the C, C++, and Fortran programming languages, and multiple hardware architectures, including: X86 and POWER CPUs, NVIDIA GPUs, AMD GPUs, and Xeon Phis (the latter was a high core count x86 processor which is now discontinued, for reference see [23]). The below table provides the compiler command for the different programming languages.

Please note that in the LUMI environment OpenACC is currently supported for Fortran only.

3.2.1.1. Compilers

OpenACC is supported by commercial, Open Source and academic compilers, e.g.

- Commercial Compilers:
  - Hewlett Packard Enterprise (HPE) Cray

  - NVIDIA HPC SDK
  - AMD (Siemens)[2]
- GNU Compiler Collection (GCC)
- LLVM (development)[36]
- Academic Compilers
  - Omni compiler project
  - openARC, Oak Ridge National Laboratory
  - OpenUH, University of Houston, Stony Brook University
  - ROSEACC, LLNL/University of Delaware

However, we focus here only using the NVIDIA HPC SDK, HPE Cray, AMD and GNU compilers.

3.2.1.2. Directive and Clauses

The listings below show the basic syntax in OpenACC for C/C++ and Fortran. To be able to use the OpenACC API in the application, please use: **#include "openacc.h"** for C/C++ and **use openacc** for Fortran; these two concepts are shown in the example below.

```
/*-- C/C++ --*/
#include "openacc.h"
#pragma acc <directive> [clauses [[,] clause] . . .] new-line
```

```
/*-- Fortran --*/
use openacc
!$acc <directive> [clauses [[,] clause] . . .]
```

A **pragma** instructs the compiler to run the region in parallel by a team of threads, for example. **acc** instructs the compiler to use the OpenACC directive definitions.

A directive is an instruction to the compiler on how the parallel region's code block can be executed, or it can be a directive instructing data placement. In OpenACC, three types of directives are available:

- *Compute directives*
- *Data management directives*
- *Synchronization directives*

A clause is an argument to the directives, which instructs the compiler with more information on how to implement the directives. Three types of clauses exist in OpenACC:

- *Data handling*
- *Work distribution*
- *Control flow*

[2]Siemens work with AMD on a GNU compiler which aim to integrate OpenACC standard for AMD GPUs

3.2.1.3. Compute Constructs and Data Environment

The OpenACC paradigm belongs to the intersection of three categories, which are: Incremental, Single Source, and Low Learning Curve. That is, it is easy to learn, easy to maintain (both serial and parallel code), and remains easily portable to a different architectures. Figure 4 shows the OpenACC implementation cycle, where a serial code can be converted to better parallel code.

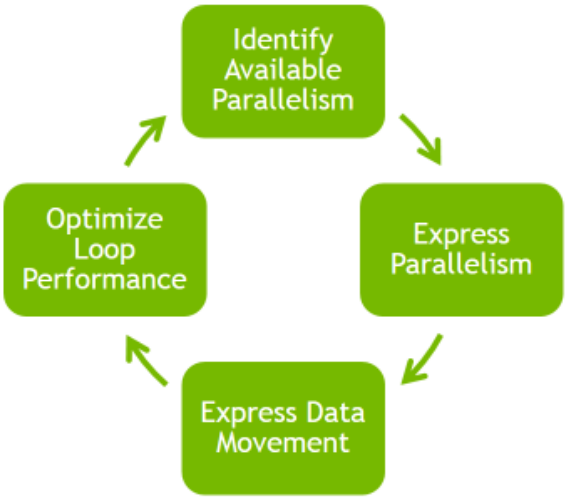

Figure 4. OpenACC implementation cycle

The OpenACC has four compute constructs; these are: *parallel*, *kernels*, *loop*, and *routine*, which distribute the work among the parallel threads in a particular region where OpenACC constructs are defined. For the data handling (memory transfer between CPU (host) to GPU(device), OpenACC provides the following clauses:

- *copy(list)*: Allocates memory on GPU and copies data from the host to GPU when entering region and copies data to the host when exiting the region
- *copyin(list)*: Allocates memory on GPU and copies data from the host to GPU when entering the region
- *copyout(list)*: Allocates memory on GPU and copies data to the host when exiting the region
- *create(list)*: Allocates memory on GPU but does not copy
- *present(list)*: Data is already present on GPU from another containing data region
- *deviceptr(list)*: The variable is a device pointer (e.g. CUDA) and can be used directly on the device

3.2.1.4. Programming in OpenACC

We start with a few simple examples. Serial code is compiled with OpenACC compiler and output is shown below; as you can see here, the loop is neither parallelised nor vectorised.

```
/*-- Hello_World.c --*/
void Print_Hello_World()
{
  for(int i = 0; i < 5; i++)
    {
      printf("Hello World!\n");
    }
}
```

```
/*-- Compilation output --*/
Print_Hello_World:
      6, Loop not vectorized/parallelized: contains call
main:
     14, Print_Hello_World inlined, size=5 (inline) file Hello_World.c (4)
           6, Loop not vectorized/parallelized: contains call
```

OpenACC code is compiled with an OpenACC compiler and output is shown below; as you can see here, loop is parallelized (loop is vectorized) with default thread blocks (provided by the compiler).

```
/*-- Hello_World_OpenACC.c --*/
void Print_Hello_World()
{
#pragma acc kernels
  for(int i = 0; i < 5; i++)
    {
      printf("Hello World!\n");
    }
}

/*-- Compilation output --*/
Print_Hello_World:
      8, Loop is parallelizable
         Generating Tesla code
          8, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

Similarly, we can see the example in Fortran.

```
!! Serial Hello World !!
subroutine Print_Hello_World()
  integer :: i
  do i = 1, 5
     print *, "hello world"
  end do
end subroutine Print_Hello_World

/*-- Compilation output --*/
print_hello_world:
      3, Loop not vectorized/parallelized: contains call


!! OpenACC Hello World!!
subroutine Print_Hello_World()
  integer :: i
  !$acc kernels
  do i = 1, 5
     print *, "hello world"
  end do
  !$acc end kernels
end subroutine Print_Hello_World

/*-- Compilation output --*/
print_hello_world:
      4, Loop is parallelizable
         Generating Tesla code
          4, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

3.2.1.5. Basic Vector Operations

We will now go through how to do the vector addition in C/C++ and Fortran programming languages. The listing below shows the simple vector addition function that adds the two vectors into one.

```
/*-- Vector_Addition.c --*/
float * Vector_Addition(float *a, float *b, float *c, int n)
{
  for(int i = 0; i < n; i ++)
    {
      c[i] = a[i] + b[i];
    }
  return c;
}
```

With a default declaration of arrays Fortran start with 1^{3} , not 0 as for C/C++.

```
!! Serial Vector Addition !!
module Vector_Addition_Mod
  implicit none
contains
  subroutine Vector_Addition(a, b, c, n)
    !! Input vectors
    real, intent(in), dimension(:) :: a
    real, intent(in), dimension(:) :: b
    real, intent(out), dimension(:) :: c
    integer :: i, n
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
  end subroutine Vector_Addition
end module Vector_Addition_Mod
```

Now, we need to use the OpenACC directives to make this function run in parallel. To achieve that, we need to add OpenACC compute directives, for example, *parallel* or *kernels*. Since this function involves a loop; when there is a loop, it is better to use the loop clause along with the compute directive. See the example below for the OpenACC enabled Vector_Addition function. We can also notice that we have used the *copyin* and *copyout* clauses; these are needed to define the data transfer between CPU and GPU. The compiler will be able to deal with the simplest cases automatically even without corresponding clauses. However, to avoid any possibility of a mistake, one should use the *copyin* and *copyout* clauses. For more complex scenarios this needs more attention.

```
/*-- OpenACC Vector Addition --*/
void Vector_Addition(float *a, float *b, float *restrict c, int n)
{
#pragma acc kernels loop copyin(a[:n], b[0:n]) copyout(c[0:n])
  for(int i = 0; i < n; i ++)
    {
      c[i] = a[i] + b[i];
    }
}
```

```
!! OpenACC Vector Addition !!
module Vector_Addition_Mod
```

[3] declaring *real::x(0:n-1)* make fortran vector start at 0

```
  implicit none
contains
  subroutine Vector_Addition(a, b, c, n)
    !! Input vectors
    real, intent(in), dimension(:) :: a
    real, intent(in), dimension(:) :: b
    real, intent(out), dimension(:) :: c
    integer :: i, n
    !$acc kernels loop copyin(a(1:n), b(1:n)) copyout(c(1:n))
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
    !$acc end kernels
  end subroutine Vector_Addition
end module Vector_Addition_Mod
```

The OpenACC compiler needs to avoid the pointer aliasing to do that; we need to use the *restrict* type qualifier with the pointer variable. This is essentially needed where we need to update the array values, where the pointer points to.

As we can notice here in the above example, we have been using only the kernels compute directive. This is because the *kernels* compute directive is safer; furthermore, the compiler executes code below sequentially if there are any dependencies in the multiple loops. This demonstrate that there are multiple ways of doing things, some trial and error must be expected.

3.2.2. OpenMP offload

OpenMP API offloading supports C/C++ and Fortran code to be executed on the accelerators. OpenMP has a different specification versions, which have evolved over the years since 1997 (the year the OpenMP specification was released). Since 2013, OpenMP has started to support accelerator offloading computation [70]. And the latest OpenMP specification, 5.2, was released in Nov 2021.

OpenMP offloading is supported by many compilers, such as AMD, ARM, GNU, HPE, IBM, Intel, NVIDIA HPC compilers, etc. Most of these compilers can support both C/C++ and Fortran programming languages.

The OpenMP programming model is based on directives, and these directives have many clauses to support parallelism in the SIMD architecture. Similarly to support accelerators (for SIMT architecture), OpenMP API introduced the "Device Directive". Mainly they can be categorised into three; they are device control, work sharing and data mapping. The below listing shows the example code block for the target construct.

```
/*-- C/C++ --*/
#pragma omp target [clause[[,]clause]...]new-line
    structured-block
```

```
/*-- Fortran --*/
!$omp target [clause[[,]clause]...]
    structured-block
!$omp end target
```

The following simple example shows how target constructs execute the printing call on the device in C/C++ and Fortran programming language. When the host thread reaches the target directive, a single thread will execute the code block within the target construct on the device

```
/*-- C/C++ --*/
#pragma omp target
{
   printf("Hello from device\n");
}
```

```
/*-- Fortran --*/
  !$omp target
  print *, "Hello from device"
  !$omp end target
```

However, the main idea of using the accelerator is to use more threads. For this reason, we need to create more threads. OpenMP API provides a "teams" clause. This can be used along with "target". The below example shows the simple example of using teams along with the target. Here a single host thread will create three worker threads in the device.

```
/*-- C/C++ --*/
#pragma omp target teams num_threads(3)
{
   printf("Hello from device\n");
}
```

```
/*-- Fortran --*/
  !$omp target teams num_teams(3)
  print *, "Hello from device"
  !$omp end target teams
```

The above example of a target with teams provides coarse-grain parallelism. But in some cases, we need to have fine-grained parallelism. In order to achieve that, we need to add the "parallel" and "distribute" directives, the below example shows the simple case for C/C++ and Fortran.

```
/*-- C/C++ --*/
#pragma omp target teams distribute parallel for
for (int i=0; i < 6; i++)
printf("Hello from id %d\n",i);
```

```
/*-- Fortran --*/
!$omp target teams distribute parallel do
do i = 1, 6
   print *, "hello from id \n", i
enddo
!$omp end target teams distribute parallel do
```

Furthermore, SIMD clauses can be added to parallelise the loops further. However, we need to make sure that the compiler supports this; for example, NVIDIA HPC SDK does not support SIMD on the device [69]. It will simply ignore it[4] . Figure 5 shows a visual representation of loop parallelisation on the device.

[4]While not relevant for AMD, it's interesting to note as all GPUs have SIMD capacity [27].

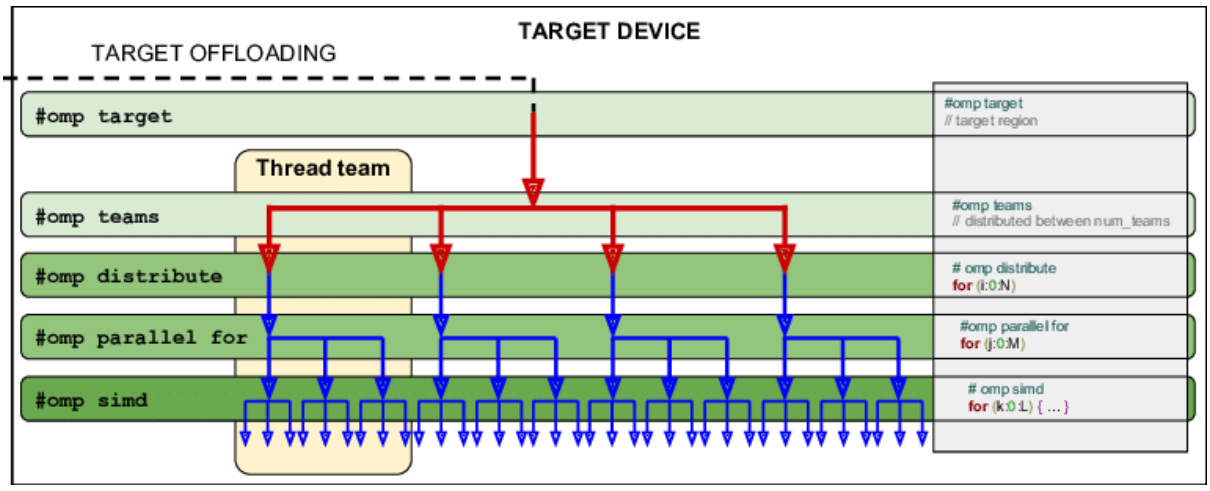

Figure 5. OpenMP offloaing loop parallelisation with SIMD;source [71]

OpenMP API specification from 5.0 onwards provides simplified memory and loop handling. In addition, similar to OpenACC, OpenMP also started to support the unified memory concept; however as far as we see, HPE compiler does not support that in OpenMP API Fortran yet[5]. This can be implemented as follows:

```
/*-- C/C++ --*/
#pragma omp requires clause[[[,]clause]...]new-line
#pragma omp requires unified_shared_memory
```

```
/*-- Fortran --*/
!$omp requires clause[[[,]clause]...]
!$omp requires unified_shared_memory
```

Moreover, a new loop construct is being introduced, which will replace the lengthier clause declaration on the directives. See the below listing for the unified memory and loop construct.

```
/*-- C/C++ --*/
#pragma omp target teams loop // collapse(n)<-for nested loop
for (int i=0; i < N; i++)
/*-- computation --*/
```

```
/*-- Fortran --*/
!$omp target teams loop !! collapse(n)<-for nested loop
do i = 1, N
!! compuation !!
enddo
!$omp end target teams loop !! collapse
```

3.2.3. HIP

HIP offers a C++ API for programming kernels, as well as a runtime library. HIP can be used to create applications which are functionally portable on either the AMD ROCm platform, or the NVIDIA GPUs, using the same source

[5]Please consult LUMI local documentation [68] for relevant up to date information.

code. The API is distributed as open source and its vocabulary is quite similar to CUDA. It supports a big subset of CUDA runtime functionality. It has almost no performance impact over coding directly in CUDA mode, and support features such as C++11 lambdas, namespaces, classes, templates etc. The HIPIFY tools can be used to convert CUDA codes to HIP. Of course, tuning will be required for each specific GPU. Figure [6] is presented the approach of HIP with regards to specific GPUs. The HIP API is similar to CUDA. For programmers who are familiar with CUDA, it will be easy to work with HIP.

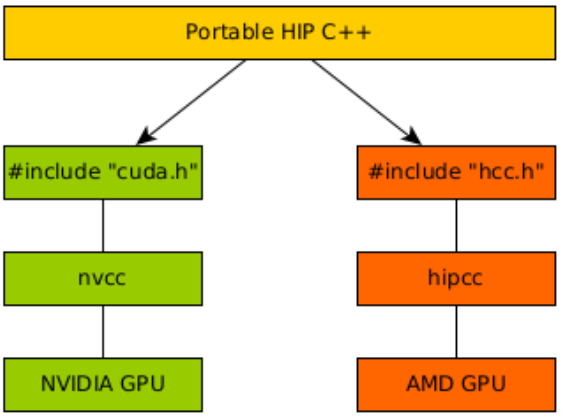

Figure 6. Diagram of HIP utilization on different GPUs

In the Section 3.2.3.1, “SAXPY” , we show some HIP calls that are used to develop a Single Precision A*X+Y (saxpy) code (see appendix Section 2, “Terms” for info on BLAS prefix).

3.2.3.1. SAXPY

The kernel is similar to CUDA ones: similarly, contextual variables are used to locate the GPU threads, and have the same names as in CUDA. What changes is the API include header, "hip/hip_runtime.h".

```
include "hip/hip_runtime.h"
#include <stdio.h>

__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

While for the main code, we can see some differences in Table 3:

Table 3. SAXPY code from CUDA to HIP

CUDA	HIP	What is does
cudaMalloc	hipMalloc	Allocates a memory pointer on GPU
cudaMemcpy	hipMemcpy	Copy data between CPU and GPU
cudaFree	hipFree	Deallocates memory from the GPU

The execution of the kernel is similar but there are some differences. For HIP the hipLaunchKernelGGL is used, and same kernel name and arguments are used.

Table 4. Execute a kernel from CUDA and HIP

CUDA	HIP
kernel_name <<<gridsize, blocksize, shared_mem_size, stream>>> (arg0, arg1, ...);	hipLaunchKernelGGL(kernel_name, gridsize, blocksize, shared_mem_size, stream, arg0, arg1, ...);

```
int main(void)
{
  int N = 1 << 15;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  hipMalloc(&d_x, N*sizeof(float));
  hipMalloc(&d_y',' N*sizeof(float));
  ...
  hipMemcpy(d_x, x, N*sizeof(float), hipMemcpyHostToDevice);
  hipMemcpy(d_y, y, N*sizeof(float), hipMemcpyHostToDevice);

  hipLaunchKernelGGL(saxpy, dim3((N+255)/256), dim3(256),
                     0, 0, N, 2.0f, d_x, d_y);

  hipMemcpy(y, d_y, N*sizeof(float), hipMemcpyDeviceToHost);
  ...
  hipFree(d_x);
  hipFree(d_y);
  ...
}
```

3.2.3.2. Terminology

Terminology-wise, CUDA and HIP are very similar, the main thread block and grid terminology is the same. The only difference is that in CUDA 32 threads executing at once is referred to as a warp, whereas in the HIP nomenclature the threads executing at the same time is referred to as a wavefront. How many threads are included in a wavefront is determined by what hardware is used, when running the code on AMD hardware the wavefront consists of 64 threads, however when running HIP code on NVIDIA hardware it is only 32 threads. In order to execute GPU kernels, we use special variables whose purpose is to identify the thread on the grid.

3.2.3.3. API (Runtime API)

As you can see in the Table 3, the differences in the API are straight-forward, basically the cuda-prefixed calls are converted to hip-prefixed calls. Of course there could be a bit more complicated cases but most of the codes can be converted as described. There is a difference on the used header file. Some examples of the API are presented below

- Device Management

  hipSetDevice(): Declare the device that should be used for hip API calls from this thread
  hipGetDevice(): Return the device id that was used calling from the host thread.
  hipDeviceSynchronize(): Waits on all active streams on current device

- Memory Management

hipMalloc(): Allocate memory on the default accelerator
hipMemcpy(): Memory transfer from host to device, device to host, device to device and host to host
hipMemcpyAsync(): Transfer asynchronously data. The memory should be pinned declared
hipFree(): Free allocated memory
hipHostMalloc(): Allocate pinned memory on the host

- Streams

hipStreamCreate(): Create an asynchronous stream
hipStreamSynchronize(): Wait for all commands in a stream to be completed
hipStreamWaitEvent(): Make a stream to wait for an event
hipStreamDestroy(): Deallocates a stream. If some commands are still executing on the specified stream, some may complete execution before the queue is deleted.

- Events

hipEventCreate(): Create an event
hipEventRecord(): Record an event for a specified stream
hipEventElapsedTime(): Elapsed time between two events

- Device Kernels

__global__: Functions that will be executed on the device and called from the host.
They should return void.

__device__: Functions that will be executed on the device and are called from the device.
hipLaunchKernelGGL(): The call to execute GPU kernels

- Device Code

threadIdx, blockIdx, blockDim, __shared__: Coordinate index functions and declare shared memory
Hundreds math functions covering entire CUDA math library

- Error Handling

hipGetLastError(): Return last error by any HIP runtime API
hipGetErrorString(): Return text message to explain an error

3.2.3.4. Libraries

AMD has ported many well-known libraries to HIP/ROCm. When a library's name starts with *roc*, then the library needs and works only on AMD GPU; if it starts with the name hip, then the AMD hipcc wrapper will understand if required to use nvcc for NVIDIA GPUs or hcc for AMD GPUs. You should use the libraries when appropriate as they are optimized for specific hardware. Table 5 showcases some libraries that are available, for the complete list please check [28].

Table 5. Libraries for CUDA, HIP, and ROCm

CUDA	HIP	ROCm	Description
cudaBLAS	hipBLAS	rocBLAS	Basic Linear Algebra Subroutines
cudaFFT	hipFFT	rocFFT	Fast Fourier Transfer Library
cuSPARSE	hipSPARSE	rocSPARSE	Sparse BLAS + SPMV
cuRAND	hipRAND	rocRAND	Random Number Generator Library
NCCL		RCCL	Communications Primitives Library based on the MPI equivalents
CUB	hipCUB	rocPRIM	Low Level Optimized Parallel Primitives
cuDNN	*[a]	MIOpen	Deep Learning Solver Library
AMG-X	*	rocALUTION	Sparse iterative solvers and pre-conditioners with Geometric and Algebraic Multi Grid

[a]The ROCm versions of the libraries can be installed with HIP support.

3.2.3.5. Memory Allocation

3.2.3.5.1. Host Memory

Continuing on the memory allocation topic. The *hipHostMalloc* allocates pinned host memory which is accessible to all the GPUs in the compute node. This helps for faster *HostToDevice* and *DeviceToHost* data transfers. Moreover, the GPU can access the host memory over the CPU/GPU interconnect without requiring to copy the data. A developer should be careful as accessing the interconnect is significantly slower than the GPU's memory. The *hipHostMalloc* can also use memory flags. The *hipHostMallocPortable* defines the memory declared to be considered as pinned memory by all contexts, not just the one which performed the allocation. The *hipHostMallocMapped* maps the allocation into the address space for the current device. The device pointer of the memory can be obtained by *hipHostGetDevicePointer()*.

3.2.3.5.2. Coherency

Two coherency options are provided for host memory:

- Coherent memory: The developer can synchronize while the kernel is running and an operation is accessible by CPU and GPU. However, a coherent memory can not be cached by the GPU, thus there could be performance penalties. The type of memory can be selected with environment variable:

```
export HIP_HOST_COHERENT=1
```

- Non-coherent memory: On the other hand, memory can be cached by the GPU but there is no support of synchronization while the kernel is running. The usual synchronizations at the end of the kernel or copy command can be used. This approach could provide better performance.

```
export HIP_HOST_COHERENT=0
```

3.2.4. HIP Porting tools

As mentioned, the Hipify tools can be used to convert CUDA code to HIP. It is possible that not all of the code is converted, and manual modifications may be required. One tool is called hipify-perl and it is a text-based replacement, as it also adds the appropriate headers. The second tool is called hipify-clang and it is a source-to-source translator that uses the clang compiler. When a code is ported on a CUDA machine it is easy to port part of the code to investigate the results and continue. HIP should have similar performance to CUDA running on relevant hardware.

3.2.4.1. Hipify-perl

Hipify-perl is an easy to use tool; it can convert for example a single file, or scan a directory and it will attempt to hipify the CUDA code. Its main technique is string replacement, the *cuda* string is replaced with *hip* and also the appropriate headers are added. You should always check the correctness of the translation. The tool prints output of the translation for each file and a summary at the end. For example we have a file called mult.cu, there are few approaches to convert it to HIP, not all of them are analysed here:

```
module load hip
hipify-perl mult.cu > mult.cpp
```

Then the mult.cpp is the hipified version of mult.c.hip. The hipify-perl adds also the hipblas header as it can be seen below. It can include headers from supported HIP libraries if there is the corresponding CUDA call in the initial code. The original CUDA code is the following:

The headers are changed and all the cuda* calls have been converted to hip* calls as we can see in Table [6].

Table 6. Hipify CUDA code to HIP

CUDA

```
...
#include <cuda_runtime.h>
#include "cublas_v2.h"
...
if (cudaSuccess != cudaMalloc((void **) &a_dev, sizeof(*a) * n * n) ||
      cudaSuccess != cudaMalloc((void **) &b_dev, sizeof(*b) * n * n) ||
      cudaSuccess != cudaMalloc((void **) &c_dev, sizeof(*c) * n * n)) {
...
    cudaFree(a_dev); cudaFree(b_dev); cudaFree(c_dev);

...
```

HIP

```
...
#include <hip/hip_runtime.h>
#include "hipblas.h"
...
if (hipSuccess != hipMalloc((void **) &a_dev, sizeof(*a) * n * n) ||
      hipSuccess != hipMalloc((void **) &b_dev, sizeof(*b) * n * n) ||
      hipSuccess != hipMalloc((void **) &c_dev, sizeof(*c) * n * n)) {
...
    hipFree(a_dev); hipFree(b_dev); hipFree(c_dev);
...
```

By executing the following command:

```
hipify-perl --inplace mult.cu
```

Then the original file mult.cu is saved in mult.cu.prehip and the new mult.cu file will be the hipified file. This way you can compile your code with the same filenames. If you want to change the original source code, modify the mult.cu.prehip and then execute the hipify-perl script again.

The script *hipconvertinplace-perl.sh* executes the *hipify-perl* with *-inplace* and *-print-stats* options for the whole directory that is used as argument. For example, to hipify a directory called src:

```
hipconvertinplace-perl.sh src
```

This command will hipify all the appropriate files and it will print statistics for each file, including a summary.

```
Output:

info: TOTAL-converted 105 CUDA->HIP refs ( ... memory:42 ... library:31 ...
include_cuda_main_header:4 type:2 literal:0 numeric_literal:24 ...)
  warn:0 LOC:1577
  kernels (0 total) :

  hipFree 36
  HIPBLAS_STATUS_SUCCESS 12
  hipSuccess 8
  hipMalloc 6
  HIPBLAS_OP_N 4
  hipDeviceSynchronize 2
  hip_runtime 2
```

From the output, it is illustrated that 105 total CUDA calls were converted to HIP ones and it is mentioned what type of calls. There are 0 warnings, it is important to check if there is any warning as maybe a call was not able to be converted. Total lines are 1577, 36 calls were converted to hipFree and you can observe the rest of the API calls. It is also possible to just examine the directory to check if there would be an error in the case of a conversion. From inside the directory we can execute the script *hipexamine-perl.sh*

```
hipexamine-perl.sh
  info: converted 1 CUDA->HIP refs ( error:0 init:0 ... library:1 ...
  numeric_literal:0 define:0 extern_shared:0 kernel_launch:0 )
  warn:0 LOC:155 in './matMul.cu'
  info: converted 52 CUDA->HIP refs ( error:0 init:0 ... device:1 ...
  memory:21 ... library:15 ... include_cuda_main_header:2 type:1 ...
  literal:0 numeric_literal:12)
  warn:0 LOC:689 in './matMulAB.c'

  info: TOTAL-converted 53 CUDA->HIP refs ( error:0 init:0 ...
  device:1 ... memory:21 ... library:18 ... include_cuda_main_header:2 ...
  type:1 literal:0 numeric_literal:12)
  warn:0 LOC:844
  kernels (0 total) :

  hipFree 54
  HIPBLAS_STATUS_SUCCESS 18
  hipSuccess 12
  hipMalloc 9
  HIPBLAS_OP_N 6
  hipDeviceSynchronize 3
  hip_runtime 3
```

The files with CUDA code print a summary of the calls and in which category they belong. At the end, the summary shows for example how many hipFree calls would be converted among all the other HIP calls.

3.2.4.2. Hipify-clang

Hipify-clang is clang based tool to translate CUDA codes to HIP. Sometimes you need to declare appropriate options if a header can not be found or something similar -I, -D, -cuda-path, etc. Clang is responsible for the CUDA support. All the CUDA code should beforehand be correct in order to be translated. If we execute:

```
hipify-clang matMulAB.cu -I/appl/opt/rocm/rocm-4.0.0/llvm/include/
```

Then a file called matMulAB.cu.hip will be created which is hipified. If you add the flag -print-stats, then we have the following:

```
hipify-clang -print-stats mult.c -I/appl/opt/rocm/rocm-4.0.0/llvm/include/
```

```
[HIPIFY] info: file 'matMulAB.cu' statistics:
  CONVERTED refs count: 51
  UNCONVERTED refs count: 0
  CONVERSION %: 100
  REPLACED bytes: 615
  TOTAL bytes: 23540
  CHANGED lines of code: 27
  TOTAL lines of code: 689
  CODE CHANGED (in bytes) %: 3
  CODE CHANGED (in lines) %: 4
  TIME ELAPSED s: 0.38
[HIPIFY] info: CONVERTED refs by type:
  device: 1
  memory: 21
  library: 13
  include_cuda_main_header: 2
  type: 1
  literal: 1
  numeric_literal: 12
[HIPIFY] info: CONVERTED refs by API:
  CUDA RT API: 28
  cuBLAS API: 23
[HIPIFY] info: CONVERTED refs by names:
  CUBLAS_OP_N: 2
  CUBLAS_STATUS_SUCCESS: 6
  cublasCreate: 1
  cublasDestroy: 7
  cublasGetMatrix: 1
  cublasHandle_t: 1
  cublasSetMatrix: 3
  cublasSgemm: 2
  cublas_v2.h: 1
  cudaDeviceSynchronize: 1
  cudaFree: 18
  cudaMalloc: 3
  cudaSuccess: 4
  cuda_runtime.h: 1
```

In the output of the print-stats is presented how many calls were converted, if there is any unconverted, a lot of information about the changed lines of codes and a list of the changed calls.

3.2.4.3. Hipify C/C++ code

We consider two examples: SAXPY with kernel and SAXPY with cuBLAS. SAXPY is the single precision code for a * X + Y.

3.2.4.3.1. SAXPY

The file saxpycuda.cpp includes saxpy code in CUDA with memory copy and execute a kernel. We hipify the code:

```
hipify-perl --inplace --print-stats csaxpycuda.cpp

  info: converted 14 CUDA->HIP refs ( error:0 init:0 ... memory:7 ...
device_function:3 ... numeric_literal:3 ... kernel_launch:1 )
  warn:0 LOC:40 in 'csaxpycuda.cpp'
  hipMemcpy 3
  hipFree 2
  hipMemcpyHostToDevice 2
  hipMalloc 2
  hipLaunchKernelGGL 1
  hipMemcpyDeviceToHost 1
```

There were 14 calls converted to HIP. Moreover, the hip_runtime header is included. The kernel remains exactly the same as it is supported from both GPU ecosystems. In Table [7] we can see the SAXPY code before and after the conversion from CUDA to HIP.

Table 7. Convert SAXPY from CUDA code to HIP

CUDA

```
#include <stdio.h>

__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<15;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

```
  float maxError = 0.0f;
  for (int i = 0; i  N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

HIP

```
#include "hip/hip_runtime.h"
#include <stdio.h>

__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<15;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  hipMalloc(&d_x, N*sizeof(float));
  hipMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  hipMemcpy(d_x, x, N*sizeof(float), hipMemcpyHostToDevice);
  hipMemcpy(d_y, y, N*sizeof(float), hipMemcpyHostToDevice);

  hipLaunchKernelGGL(saxpy, dim3((N+255)/256), dim3(256), 0, 0, N, \
  2.0f, d_x, d_y);

  hipMemcpy(y, d_y, N*sizeof(float), hipMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  hipFree(d_x);
  hipFree(d_y);
  free(x);
  free(y);
}
```

Compilation: *hipcc -o saxpy_hip saxpy.cpp*.

3.2.4.3.2. SAXPY with cuBLAS

We have a file called saxpy.cpp with CUDA code, we execute:

```
hipify-perl --inplace --print-stats saxpy.cpp

  info: converted 12 CUDA->HIP refs ( error:0 init:0 version:0 ...
  library:6 ... include_cuda_main_header:1 type:1 )
  warn:0 LOC:35 in 'saxpy.cpp'
  hipFree 2
  hipMalloc 2
```

From the report, it emerges that 12 total calls were hipified, 4 from memory, 6 from library, etc. The cuBLAS calls were converted and the corresponding header is included. In Table [8] we can see the SAXPY code before and after the conversion from CUDA with cuBLAS to HIP.

Table 8. Convert SAXPY with cuBLAS from CUDA code to HIP

CUDA

```
#include <iostream>
#include "cublas_v2.h"
using namespace std;
const int N = 1 << 15;

int main(){
        float *a_h, *b_h;
        a_h = new float[N];
        b_h = new float[N];
        float *a_d, *b_d;
        float maxError = 0.0f;
        const float s = 2.0f;
        for(int i = 0; i < N; i++){
                a_h[i] = 1.0f ;
                b_h[i] = 2.0f ;
        }
        cublasHandle_t handle;
        cublasCreate(&handle);
        cudaMalloc(&a_d, sizeof(float) * N);
        cudaMalloc(&b_d, sizeof(float) * N);
        cublasSetVector( N, sizeof(float), a_h, 1, a_d, 1);
        cublasSetVector( N, sizeof(float), b_h, 1, b_d, 1);
        cublasSaxpy(handle, N, &s, a_d, 1, b_d, 1);
        cublasGetVector( N, sizeof(float), b_d, 1, b_h, 1);
        for (int i = 0; i < N; i++)
                maxError = max(maxError, abs(b_h[i]-4.0f));
        printf("Max error: %f\n", maxError);
        cudaFree( a_d);
        cudaFree( b_d);
        cublasDestroy(handle);
        delete[] a_h;
        delete[] b_h;
        return 0;
}
```

HIP

```
#include <iostream>
#include "hipblas.h"
using namespace std;
const int N = 1 << 15;

int main(){
        float *a_h, *b_h;
        a_h = new float[N];
        b_h = new float[N];
        float *a_d, *b_d;
        float maxError = 0.0f;
        const float s = 2.0f;
        for(int i = 0; i &lt; N; i++){
               a_h[i] = 1.0f ;
               b_h[i] = 2.0f ;
        }
        hipblasHandle_t handle;
        hipblasCreate(&handle);
        hipMalloc(&a_d, sizeof(float) * N);
        hipMalloc(&b_d, sizeof(float) * N);
        hipblasSetVector( N, sizeof(float), a_h, 1, a_d, 1);
        hipblasSetVector( N, sizeof(float), b_h, 1, b_d, 1);
        hipblasSaxpy(handle, N, &s, a_d, 1, b_d, 1);
        hipblasGetVector( N, sizeof(float), b_d, 1, b_h, 1);
        for (int i = 0; i < N; i++)
                maxError = max(maxError, abs(b_h[i]-4.0f));
        printf("Max error: %f\n", maxError);
        hipFree( a_d);
        hipFree( b_d);
        hipblasDestroy(handle);
        delete[] a_h;
        delete[] b_h;
        return 0;
}
```

Compilation:a *hipcc -o saxpy_hip saxpy.cpp -I/path_to_hipblas/include/ -L/path_to_hipblas/lib/ -lhipblas*. If you use NVIDIA GPU, add the option *-x cu* after the hpicc.

Both codes can be executed with similar performance.

3.2.4.3.3. Benchmarks

Unfortunately this benchmark exercise was performed on NVIDIA hardware, but its main purpose is to illustrate comparable performance using CUDA versus HIP.

- MatMul OpenMP Offload

  We use a matrix multiplication example from [72] with size 2048 x 2048. All the CUDA calls were converted and the application was linked with hipBLAS among also OpenMP offload

  Results:

  - CUDA:

    matMulAB (11): 1001.2 GFLOPS 11990.1 GFLOPS maxabserr= 0.0

  - HIP:

matMulAB (11): 978.8 GFLOPS 12302.4 GFLOPS maxabserr= 0.0

The first value of GFLOPS includes the whole execution time, while the second value is only the kernel execution time. We can see that the HIP execution time is around 2.23% slower but it is a bit faster for the kernel. However, such a small different is hardly significant. With HIP we expect similar results to CUDA by using the same NVIDIA GPU.

- N-BODY Simulation

  A second application is the N-Body simulation [73] test-case AllPairs_N2. 171 CUDA calls were converted to HIP without issues, close to 1000 lines of code. The problem size is 32768 number of small particles and 2000 time steps.

  Results:

  - CUDA:

    Execution time: 68.5 seconds

  - HIP:

    Execution time: 70.1 seconds

The HIP overhead is only ~2.33%, which is hardly significant.

3.2.4.4. Hipify Fortran code

In some cases, porting Fortran with CUDA to HIP, can require more effort. In this section we mention two cases and we focus on the second case that is more complicated.

- Fortran + CUDA C/C++

  In the case that there are CUDA calls only from C/C++, then you can hipify as above the C/C++ files and compile the HIP file with hipcc and link also with hipcc

- CUDA Fortran

  In this case there is no direct HIP equivalent. It is required to use Fortran 2003 C bindings to use the appropriate pointers to access the kernel.

It is required to do the following steps:

- Write the kernels in a new file with extension .cpp
- Wrap the kernel launch in a C function
- Use Fortran 2003 C binding to call the C function
- Use hipfort, a Fortran interface for GPU Kernels

Figure [7], shows the approach that we follow.

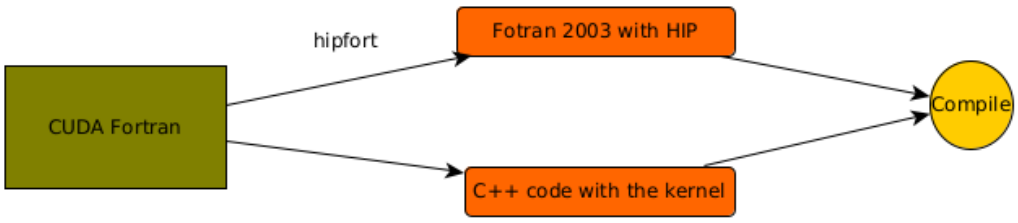

Figure 7. From CUDA Fortran to HIP

In the Table [9] we can see how the CUDA Fortran code should be prepared with Fortran 2003 code, which needs more modifications with HIP API, the hipfort, as also the Fortran interface to call the extern C routine that will execute the GPU kernel.

Table 9. Convert SAXPY CUDA Fortran to HIP

CUDA Fortran

```
module mathOps
contains
       attributes(global) subroutine saxpy(x, y, a)
          implicit none
          real :: x(:), y(:)
          real, value :: a
          integer :: i, n
          n = size(x)
          i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
         if (i <= n) y(i) = y(i) + a*x(i)
      end subroutine saxpy
end module mathOps

program testSaxpy
   use mathOps
   use cudafor
   implicit none
   integer, parameter :: N = 40000
   real :: x(N), y(N), a
   real, device :: x_d(N), y_d(N)
   type(dim3) :: grid, tBlock
   tBlock = dim3(256,1,1)
   grid = dim3(ceiling(real(N)/tBlock%x),1,1)
   x = 1.0; y = 2.0; a = 2.0
   x_d = x
   y_d = y
   call saxpy<<<grid, tBlock>>>(x_d, y_d, a)
   y = y_d
   write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

Fortran 2003 using HIP

```
program testSaxpy
  use iso_c_binding
  use hipfort
  use hipfort_check

  implicit none
  interface
     subroutine launch(y,x,b,N) bind(c)
       use iso_c_binding
       implicit none
       type(c_ptr) :: y,x
       integer, value :: N
       real, value :: b
     end subroutine
  end interface
```

```
  type(c_ptr) :: x_d = c_null_ptr
  type(c_ptr) :: y_d = c_null_ptr
  integer, parameter :: N = 40000
  integer, parameter :: bytes_per_element = 4
  integer(c_size_t), parameter :: Nbytes = N*bytes_per_element
  real, allocatable, target, dimension(:) :: x, y


  real, parameter ::  a=2.0
  real :: x_d(N), y_d(N)

  call hipCheck(hipMalloc(x_d,Nbytes))
  call hipCheck(hipMalloc(y_d,Nbytes))

  allocate(x(N))
  allocate(y(N))

  x = 1.0; y = 2.0

  call hipCheck(hipMemcpy(x_d, c_loc(x), Nbytes, hipMemcpyHostToDevice))
  call hipCheck(hipMemcpy(y_d, c_loc(y), Nbytes, hipMemcpyHostToDevice))

  call launch(y_d, x_d, a, N)

  call hipCheck(hipDeviceSynchronize())

  call hipCheck(hipMemcpy(c_loc(y), y_d, Nbytes, hipMemcpyDeviceToHost))

  write(*,*) 'Max error: ', maxval(abs(y-4.0))

  call hipCheck(hipFree(x_d))
  call hipCheck(hipFree(y_d))

  deallocate(x)
  deallocate(y)

end program testSaxpy
```

C++ using HIP

```
#include <hip/hip_runtime.h>
#include <cstdio>

__global__ void saxpy(float *y, float *x, float a, int n)
{
    size_t i = blockDim.x * blockIdx.x  + threadIdx.x;
    if (i < n) y[i] = y[i] + a*x[i];
}


extern "C"
{
  void launch(float **dy, float **dx, float db, int N)
  {
     dim3 tBlock(256,1,1);
     dim3 grid(ceil((float)N/tBlock.x),1,1);
     hipLaunchKernelGGL((saxpy), grid, tBlock, 0, 0, *dy, *dx, db, N);
```

```
  }
}
```

3.2.5. Julia

The Julia [30] programming language is young, it was publicly announced in 2012, in comparison to the well established languages such as Fortran, C/C++, R, and Python that appeared between 1950-1990. Despite being so recent, it can already offer a rich environment, like its package manager and the considerable number of supported packages. One of the most important features of Julia is that one can write both prototype code and production code, without sacrificing performance, and without having to port code to a faster language (two-language problem).

Although Julia can be installed from source, we decided to test the precompiled binary files that are available in [32]. The sequence of commands that we followed in order to use Julia is:

```
mkdir $HOME/Julia  #this location can vary depending on the workflow
cd $HOME/Julia
wget https://julialang-s3.julialang.org/bin/linux/x64/1.7/\
julia-1.7.2-linux-x86_64.tar.gz
tar xzf julia-1.7.2-linux-x86_64.tar.gz
export PATH="$HOME/Julia/julia-1.7.2/bin":"$PATH"
```

The module *rocmod4.5.2* was loaded on the terminal to make Julia aware of the ROCm API. In the present analysis, we will work in a Julia *Environment* created with the *activate* function. The *AMDGPU.jl* package offers support for AMD GPUs. In addition to this package, we recommend to install the *BenchmarkTools.jl* package to benchmark Julia code. These packages can be installed on the Julia Read Evaluate Print Loop (REPL) command-line by typing:

```
using Pkg
Pkg.activate(".")      #create an environment in the current directory
Pkg.add("AMDGPU")
Pkg.add("BenchmarkTools")
```

According to the *AMDGPU.jl* API reference [33], both *HSAArray* and *ROCArray* are supported, which allows to define arrays that can be used in computations on the GPU side. However, in the present setup where binary files were used for Julia, only *ROCArray* was recognized by Julia. Our investigation on the unrecognized *HSAArray* points to the incompatibility of the required library *libhsakmt.so* with dependencies of the Linux kernel from the version 3.19. In the Table 10 we can see a matrix-matrix multiplication code for both CPU and GPU. These following lines are common to both codes and should be executed before the lines in Table 10:

```
using Pkg
Pkg.activate(".")      #activate the environment in the current directory
using BenchmarkTools
```

In addition to the previous initialisation lines, we recommend to set the number of OpenMP threads to 1 (*export OMP_NUM_THREADS=1*) because by default several threads are spawned during linear algebra operations. Notice that Julia uses its own threading scheme through the environment variable *JULIA_NUM_THREADS* that is not related to the one of OpenMP.

Table 10. Matrix-matrix multiplication in Julia

CPU	GPU

`N = 2^7`
`A = rand(N,N)`
`B = rand(N,N)`
 	
`@benchmark A*B evals=1000`
	`using AMDGPU`
	`N = 2^7`
	`A = ROCArray(rand(N,N))`
	`B = ROCArray(rand(N,N))`
	`@benchmark A*B evals=1000`

Compared to CUDA or HIP code, the Julia code in Table 10 for matrix-matrix multiplication looks more transparent to the programmer because explicit memory allocation and movement (CPU to GPU and viceversa) is not required but they are handled by ROCArrays. The benchmark was performed with the *@benchmark* macro using 1000 evaluations, the results can be seen in Figure 8. Here, we can notice the difference in performance between the CPU and GPU code measured by the execution times of 114 µs and 51.8 µs, respectively. In addition to this the allocated memory on the CPU is also provided.

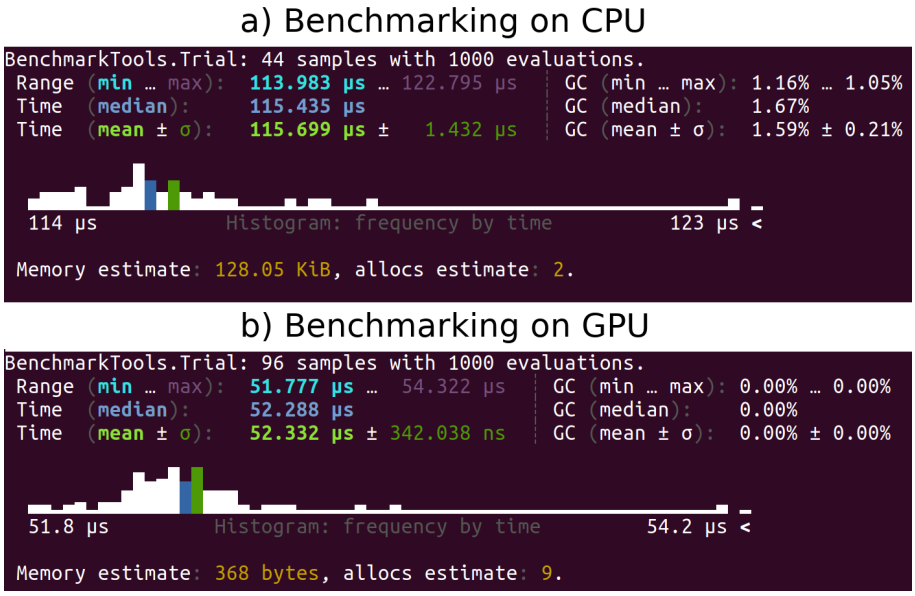

Figure 8. Benchmark results of the matrix-matrix multiplication calculation on both a) CPU (single core) and b) GPU.

```
using Pkg
Pkg.activate(".")
using AMDGPU
using BenchmarkTools
using Test

N = 2^27
a = 2.0
x = rand(Float64, N)
y = rand(Float64, N)
z_cpu = a.*x + y

x_d = ROCArray(x)
y_d = ROCArray(y)
z_d = similar(x_d)

function saxpy!(z, a, x, y)
```

```
     i = workitemIdx().x + (workgroupIdx().x -1)*workgroupDim().x
     if i <= length(x)
        z[i] = a*x[i] + y[i]
     end
     return
end

@roc groupsize=256 gridsize=N saxpy!(z_d, a, x_d, y_d)
wait(@roc groupsize=256 gridsize=N saxpy!(z_d, a, x_d, y_d))

z = Array(z_d)

@test isapprox(z, z_cpu)
```

Monitoring utilisation and progress show good utilisation of the GPU.

```
$watch -n 1 rocm-smi

======================= ROCm System Management Interface =======================
================================= Concise Info =================================
GPU  Temp   AvgPwr  SCLK     MCLK     Fan  Perf  PwrCap  VRAM%  GPU%
0    71.0c  290.0W  1472Mhz  1200Mhz  0%   auto  290.0W   10%   99%
================================================================================
WARNING:                One or more commands failed
============================== End of ROCm SMI Log ==============================
```

3.2.6. OpenCL

3.2.6.1. Introduction

OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform developed by Khronos [34] and is designed for low-level heterogeneous programming. The implementation of OpenCL covers a large spectrum of applications in HPC and AI [34]. It has the advantage of boosting deployment on a wide range of hardware vendors, which are NVIDIA, AMD, Intel, ARM and Adreno.

OpenCL offers new functionalities and optional features for accelerating applications and libraries, and for generating codes for compilers, and for writing portable applications supported on various GPU architectures [38]. Some of these OpenCL features include [34]:

- Runtime framework for accelerated applications: This feature enables offloading computation to various heterogeneous processors: CPUs, GPUs, DSPs, FPGAs and tensor processors.
- Platform layer API: It enables to select and initialise compute devices.
- Runtime API: This feature enables to build and execute programs written on C and C++ kernel languages.
- Explicit control of accelerated applications: It enables to control which devices the program is executed on, and the data location in a memory of a system. In addition, it allows identifying the dependency of different operations.

Furthermore, OpenCL offers a unified specification, which makes it easier for developers to navigate between various versions [37]. The OpenCL specification also introduces the asynchronous data locality, which enables developers to properly utilise the characteristic features of processors, and thus improving the speed of applications.

In this section, we guide readers to get started with OpenCL. A special focus is placed on how to access OpenCL API from a Fortran program. This will be illustrated via a Fortran example, in which we highlight some of the OpenCL features defined above.

3.2.6.2. Case study: Example

In the following, we provide an example of a Fortran program that makes use of OpenCL. The main module file that contains all OpenCL APIs to be invoked from a Fortran program is publicly available on GitHub [39]. We thus refer readers to the GitHub repository for additional files that demonstrate the functionality of module libraries, mainly, the Focal module [39]. This module will be briefly introduced in this section.

In the example below, we present some of the basic functions invoked when interfacing with OpenCL from a Fortran program. This example is described in five major steps, which consist of: (i) using the Focal module library, (ii) initialising OpenCL with Focal library, (iii) initialising a device memory, (iv) data locality in host-device memory, and (v) launching device kernels.

- Use of the Focal module library

The Fortran program shown below uses the Focal module [39]. The Focal module is a modern Fortran abstraction layer for OpenCL. It provides an accessible Fortran interface to the OpenCL runtime API using the CLFortran module library [40], which is also a Fortran interface to OpenCL written with pure Fortran and designed to accelerate Fortran applications with the use of OpenCL.

```
Program myProgram
  use Focal
  implicit none
  ....
end program myProgram
```

- Initialisation of OpenCL with Focal

A starting point here is to initialise OpenCL using the function **fclInit** [41]. This function contains arguments, mainly: **vendor**, **type** and **sortBy**, as shown in the code below. These arguments offer flexibility in choosing a vendor device (e.g. NVIDIA, AMD, …), the type of device (e.g. GPU) and sorting available devices and selecting the one to be used. Here it is possible to sort devices based on the number of cores (**sortBy='cores'**), or by global memory (**sortBy='memory'**), or by maximum clock speed (**sortBy='clock'**). A subsequent step is to create a command queue using the function **fclCreateCommandQ** to perform operations on OpenCL objects [41].

```
! Define device object
   type(fclDevice) :: device
   .....
! Select device with most
   device = fclInit(vendor='nvidia, amd', type='gpu', sortBy='cores')

! Create command queue
   call fclSetDefaultCommandQ(fclCreateCommandQ(device))
```

- Initialisation of device memory

Initialising device memory is done by calling the function **fclInitBuffer**, in which the argument array_d is defined as a derived type of a device buffer [41].

```
   integer,  parameter   ::  n=1000
   type(fclDeviceFloat) :: array_d

! Initialise device arrays
   call fclInitBuffer(array_d,n)
```

- Data locality in Host-device memory

Here we describe the syntax of transferring data between host-memory and a device-memory. As shown below the syntax is simple and easy to use.

```
   use iso_c_binding, only: c_float
   integer,  parameter   ::  n=1000
!Device array data
   type(fclDeviceFloat) :: array_d
!Host array data
    Real(c_float) :: array_h(n)
    .....
! Copy data from a host to a device
   array_d = array_h
! Copy data from a device to a host
   array_h = array_d
```

- Launch of device kernels

We show below an example based on computing the sum of two vectors labeled array1 and array2. Here launching a device kernel requires first loading a file called sum.cl [39], which contains OpenCL kernel. The program is then compiled to obtain a kernel object, as illustrated in the code:

```
! Launch kernel
! Device arrays
   type(fclDeviceFloat) :: array1_d, array2_d

! Set global work size equal to array length and launch kernel
   sumKernel%global_work_size(1) = n
   call sumKernel%launch(n,array1_d,array2_d)
```

Note that the file “sum.cl” defined in the code above contains the following OpenCL kernel:

```
__kernel void sum(const int n, const __global float * v1,
                  __global float * v2)
{
  int i = get_global_id(0);
  if(i &lt; n) v2[i] += v1[i];
}
```

3.2.6.2.1. Compilation process

A Fortran program that uses the Focal module is widely supported by current compilers, such as GNU (**gfortran**), Intel (**ifort**), IBM (**xlf**, **xlf_r**) and Cray (**ftn**) compilers. The syntax of the compilation process can be expressed as [41]:

```
$FC -c myprogram.f90 -I/path/to/focal/mod/ -o myprogram.o
```

where FC can be any of the compiler wrappers defined above. The path points the compiler to the Focal repository (i.e. mod/Focal.mod). Linking the generated objects (*.o) is done by specifying the path to the Focal repository and directive for Focal and OpenCL. This can be expressed as:

```
$FC myprogram.o -L/path/to/focal/lib/ -lFocal -lOpenCL -o bin/myprogram
```

For completeness, we also describe the syntax of the compilation process of OpenCL C with the GNU and Cray compilers on LUMI-G. Note that OpenCL is supported as part of ROCM. Here is a simple example:

```
ml LUMI/22.06  partition/G
ml  rocm/5.1.4
export ROCMOPENCL=${ROCM_PATH}/opencl/
# GNU:
gcc main.c -I${ROCMOPENCL}/include/ -L${ROCMOPENCL}/lib -lOpenCL
# Cray:
cc main.c -I${ROCMOPENCL}/include/ -L${ROCMOPENCL}/lib -lOpenCL
```

Modules are subject to change, consult user documentation [5] for details.

3.2.7. Higher Level Programming

3.2.7.1. SYCL

SYCL [35]is a Khronos Group C++ standard which adds support for heterogeneous data parallelism with a single-source approach. Both CPU and GPU codes are contained in the same C++ source file. The code uses standard C++ language and there is no use of language extensions, pragmas or keywords. Originally SYCL was implemented on top of OpenCL, but it was designed to target a wide range of heterogeneous platforms. It can target any devices which is supported by its back-end and it can target any number of different back-ends, just like OpenCL. Every SYCL implementation comes with the standard SYCL API, which is a standard C++ template library that provides the features needed to write a SYCL programs and the SYCL runtime needed to execute the code on the specific device. At the moment there are at least five compilers supporting SYCL : DPC++ by Intel, ComputeCpp created ComputeCpp, triSYCL from AMD and Xilinx, hipSYCL from the University of Heidelberg, and neoSYCL by NEC.

3.2.7.1.1. Anatomy of a SYCL code

In order to understand how SYCL works, one can check the simple element-wise vector addition. The code starts with including the SYCL header file:

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
```

The header includes the entire API, everything needed for developing a SYCL program. In order to reduce the amount of code the *cl::sycl* namespace is imported as well. Next, a *queue* is created to enqueue work:

```
int main(int argc, char *argv[]){
queue gpuQueue{gpu_selector()}
```

The SYCL *queue* creates a connection to a single device on which the work will be performed. As mentioned this can be any of the devices supported by the specific SYCL implementation. There are four options here. 1. Let the runtime pick the best suitable for the work; 2. Ask for a specific class of devices (CPUs or GPUs); 3. Give even more hints or take full control; 4. Examine all devices and pick one based on given criterias. In this example the argument `gpu_selector()` instructs the runtime to pick only GPUs. SYCL introduces the concept of buffers to move, control, and modify the data. The *buffer<T, dims>* class encapsulates the object that is going to be used

in the computation. The buffers take ownership of the data associate with the raw pointers. Take the example of element wise vector addition of the vectors *a*, *b* with result stored in *c*.

```
int nx=45000000;
int *a_d=new int[nx],*b_d=new int[nx],*c_d=new int[nx];

for(int i=0;i < nx;i++)
a[i]=i,b[i]=i*i;
{ // this bracket is needed to start a scope
     buffer<int, 1> a_sycl(&a, range&lt;1>(nx));
     buffer<int, 1> b_sycl(&b, range&lt;1>(nx));
     buffer<int, 1> c_sycl(&c, range&lt;1>(nx));
```

In this case 3 buffers were created, each encapsulating a *int* vector of size *nx*. The buffers are destroyed at the end of the scope defined by the brackets *{}*. Next a command group is created:

```
gpuQueue.submit([&](handler &cgh){
```

Technically this is just a single call to the *submit* member function. Its argument is a function object parameter encapsulating the command group. A command group represents a single unit a work queued to be executed. It effectively encapsulates the operations to be executed on the selected device and the related data dependencies. In a command group, it is required to set up the *accessors* as a first thing. This is the only way to access the data owned by the buffers. The accessors for the input and output vectors are created via the *get_access()* member function:

```
auto a_acc = a_sycl.get_access<access::mode::read>(cgh);
auto b_acc = b_sycl.get_access<access::mode::read>(cgh);
auto c_acc = c_sycl.get_access<access::mode::discard_write>(cgh);
```

The *get_access()* method takes an access mode argument. In the case of element-wise vector addition, for the input vectors one uses *access::mode::read* and *access::mode::discard_write* for the results. *discard_write* assumes that original data in the vector is not needed and it can be discarded. *read_write* modes can be used when the initial data in the vector is used for both input and output. Alternatively *Unified Shared Memory* can be used. The USM pointers on the CPU are are valid pointers as well on the GPU (device). The third option, *images*, offers a similar API to buffer types, with extra functionalities tailored for image processing. Finally, the actual work, an *action*, to be executed in parallel is submitted as kernel code. Kernels can be written as *lambda expressions* or as *function objects*. For our particular example as lambda, the kernel has the following form:

```
cgh.parallel_for<class vector_add>(range<1>(nx),[=] (id<> i) {
c_acc[i] = a_acc[i] + b_acc[i];
});
});
}
```

The kernel functions are executed by *work-items*. The work-items can be mapped, depending on the device, to CPU threads, SIMD lanes, GPU threads, or other type of processing elements. For the vector addition case the *parallel_for* member function indicates the device to execute in parallel *nx* instances of the operation *c_acc[i] = a_acc[i] + b_acc[i];*, where the index *i* takes the value corresponding the index of the work-item. The work-items are grouped together in *work-groups*, the size depending on targeted device. Work-groups are very similar to the *blocks* concept used in GPU programming. The work-items in a work-group can exchange data and can be synchronised among themselves. When exiting the scope the buffers are automatically synchronised and destroyed. From the above example one can notice that the parallelism is expressed in a very similar fashion to that of

the CUDA/HIP programming models. The *parallel_for* function executes in a specific queue the same *kernel*, one instance per *work-item*. This is called *data-parallelism*. SYCL provides another type of parallelism, *task-parallelism*. If more than device is present in the system, and different tasks are performed on the same or different data SYCL can be used to control the way the work is distributed between different devices, using the same code.

3.2.8. GPU-accelerated MPI applications

3.2.8.1. Introduction

GPU acceleration of MPI (message passing interface) applications has the potential to fully utilising the capacity of multiple GPUs in distributed nodes. Such a hybrid-based approach permits to significantly improve the performance of computations on modern HPC systems, and explore exascale platforms, such as the supercomputer LUMI-G.

In this section we guide readers, who are familiar with MPI, in implementing a hybrid model in which the MPI communication framework is combined with either OpenACC or OpenMP application programming interfaces (APIs). A special focus will be on performing point-to-point (e.g. **MPI_Send** and **MPI_Recv**) and collective operations (e.g. **MPI_Allreduce**) between a pair of GPUs . This concept is referred to as GPU-aware MPI. Here the GPU-awareness approach can simply mean how to make a GPU device memory aware or not aware of the existence of an MPI library. This concept will be presented in the context of both the hybrid MPI-OpenACC and MPI-OpenMP models. Its implementation has the advantage of reducing the computing time caused by transferring data via the host-memory during heterogeneous communications, thus rendering HPC applications efficient. For reference, a scenario with the non-GPU-aware MPI will also be addressed.

3.2.8.2. GPU-aware MPI approach

In the following, we first introduce how to establish a connection between each MPI rank and a specific GPU device. Here, we are in the situation in which a host and a device have a distinct memory (i.e. non-shared memory device [55]). This is followed by a presentation of the GPU-non- aware MPI and GPU-aware MPI. Subsequently, the performance analysis of a mini-application, in which the GPU- awareness approach is implemented, is discussed.

3.2.8.2.1. Assigning MPI ranks to GPU devices

Managing multiple GPU devices on different nodes requires as a first step assigning each MPI rank to a specific GPU device. Here, one needs to determine which MPI processes run on the same node, such that each can be assigned to the nearest GPU device (cf. Fig. 9). This permits to minimise latency. This procedure can be done by splitting the world communicator into sub-groups of communicators (or sub-communicators) using the routine **MPI_COMM_S PLIT_TYPE()**.

```
call MPI_COMM_SPLIT_TYPE(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, &
                         MPI_INFO_NULL, host_comm,ierr)
call MPI_COMM_RANK(host_comm, myDevice,ierr)
```

Here each sub-communicator contains a list of processes in which each one is indicated by a rank. These processes have a shared-memory region defined by the argument **MPI_COMM_TYPE_SHARED** (see ref. [56] for more details). Calling the routine **MPI_COMM_SPLIT_TYPE()** returns a sub-communicator (labeled "host_comm") in which each MPI rank can be assigned to a specific GPU device. This can be done according to which directive-based model is implemented. In the OpenACC API [58], the function call **acc_set_device_num(myDevice , acc_get_device_type())** is used to set a specific device in which the data are directed to, while this is done in the OpenMP API [60] by calling the function **omp_set_default_device(myDevice)**. Note that the function **acc_get_device_type()** returns the device type to be used in the OpenACC specification, and its counterpart in the OpenMP API is **omp_get_device_num ()**.

On the other hand, one can check the number of devices available on the host via the functions **acc_get_num_devices(acc_get_device_type())** and **omp_get_num_devices(omp_get_device_num)**, respectively, for the OpenACC and OpenMP APIs.

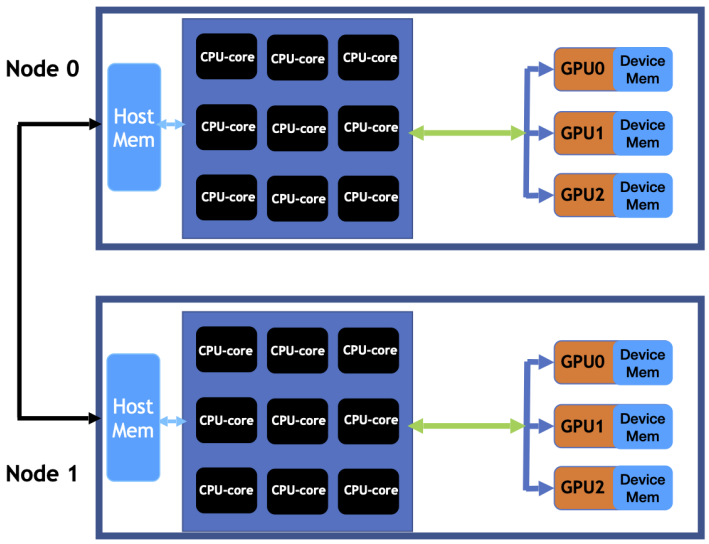

Figure 9. Schematic representation of two compute nodes, each is connected to three GPU devices. Host Memory and Device Memory represent the host-memory and the device-memory, respectively.

3.2.8.2.2. Non-GPU-aware MPI with OpenACC and OpenMP APIs

After having covered how to assign each MPI rank to a specific device, we now address the concept of the non-GPU-aware MPI. In this approach calling an MPI routine from an OpenACC or OpenMP API requires updating the data in a host before and after an MPI call. In this scenario, the data are copied back and forth between a host and a device before and after each MPI call. In the hybrid MPI-OpenACC model, the procedure is defined by specifying the directive **update host()** for copying the data from a device to a host before an MPI call; and by the directive **update device()** specified after an MPI call for copying the data back to a device. To illustrate the concept of MPI-non-GPU-aware MPI, we show below an example of an implementation that involves the MPI functions **MPI_Send()** and **MPI_Recv()**

```
!Structured data
!$acc data copyin(f)
!copy data from GPU to CPU
!$acc update host(f)

!transfer the data at the boundaries to the neighbouring MPI-process
!send f(:,nyp) from myid-1 to be stored in f(:,0) in myid+1
         if(myid.lt.nproc-1) then
          call MPI_Send(f(:,nyp), (nx+2), MPI_DOUBLE_PRECISION, myid+1, tag1, &
                        MPI_COMM_WORLD, ierr)
         endif

!receive f(:,0) from myid-1
         if(myid.gt.0) then
          call MPI_Recv(f(:,0), (nx+2), MPI_DOUBLE_PRECISION, myid-1, &
                       tag1, MPI_COMM_WORLD, status, ierr)
         endif
!update data in the host: copy from CPU to GPU
!$acc update device(f)
!$acc end data
```

A similar concept can be implemented in the hybrid MPI-OpenMP model. Here, updating the data in a host can be done by specifying the OpenMP directives **update device() from()** and **update device() to()**, respectively, for copying the data from a device to a host and back to the device as shown below.

```
!Structured data
!$omp target data device(myDevice) map(to:f)
!copy data from GPU to CPU
!$omp target update device(myDevice) from(f)

!transfer the data at the boundaries to the neighbouring MPI-process
!send f(:,nyp) from myid-1 to be stored in f(:,0) in myid+1
         if(myid.lt.nproc-1) then
          call MPI_Send(f(:,nyp), (nx+2)*1, MPI_DOUBLE_PRECISION, myid+1, &
                tag1, MPI_COMM_WORLD, ierr)
         endif

!receive f(:,0) from myid-1
         if(myid.gt.0) then
          call MPI_Recv(f(:,0), (nx+2)*1, MPI_DOUBLE_PRECISION, myid-1, &
                      tag1, MPI_COMM_WORLD, status,ierr)
         endif

!update data in the host: copy from CPU to GPU
!$omp target update device(myDevice) to(f)
!$omp end target data
```

Despite the simplicity of implementing the non-GPU-aware MPI approach, it suffers from a low performance caused by an explicit transfer of data between a host and a device before and after calling an MPI routine. This constitutes a bottleneck in GPU-programming. Furthermore, the approach is synchronous, which does not allow overlapping between MPI-based computation and OpenACC/OpenMP operations. An alternative to this approach is to implement the GPU-aware MPI as described in the next section.

3.2.8.2.3. GPU-aware MPI with OpenACC and OpenMP APIs

The concept of the GPU-aware MPI relies on the possibility of moving data that resides in a GPU device memory without necessarily using CPU-host memory as an intermediate buffer.

This approach enables an MPI library to directly access a GPU device memory, which in turn permits to transfer data from one GPU to another GPU within a specific node. Such an approach has the advantage of reducing the communication time between different MPI processes, as the interconnect of GPU-GPU is faster than the one of CPU-GPU. As an example, in the LUMI-G partition, the Infinity Fabric links of CPU-GPU has a peak of 36 GB/s on one direction (e.g. host-to-device), while it is 200 GB/s for the links GPU-GPU on AMD MI250X GPUs [10]. This characteristic feature of this kind of GPU indicates the possibility of significantly improving the performance when the GPU-aware MPI library is enabled.

In our example discussed in the Section 3.2.8.2.2, the data associated to each MPI process reside in a GPU device, as they have already been copied there. In the non-GPU-aware MPI approach, this data must be updated on a CPU-host and copied back to a GPU device. In the GPU-aware MPI, however, this data can be communicated between a pair of GPUs without necessarily passing through a CPU-host memory. This approach is supported by recent versions of MPI libraries such as OpenMPI [61] and MPICH [62]. Here when a pointer to a GPU device is passed to an MPI call, the MPI library automatically sets up a GPU memory for processing data.

In the hybrid MPI-OpenACC model, the concept is defined by combining the directive **host_data** together with the clause **use_device(list_array)**. This combination enables the access to the arrays listed in the clause **use_device(list_array)** from the host [58] . The list of arrays, which are already present in a GPU device memory, are directly passed to an MPI routine without the need of a staging host-memory for copying the data.

In our example, the GPU-aware MPI support with OpenACC is illustrated in connection with the MPI operations **MPI_Send** and **MPI_Recv** and the operation **MPI_Allreduce** as shown below. Note that not all MPI functions are supported by the GPU-awareness approach (see [64] for more details).

In this example, the data stored in the array f(:,:) are present in GPUs and are passed directly to the **MPI_Send()** and **MPI_Recv()** functions. By enabling the GPU-aware MPI, the operations **MPI_Send** and **MPI_Recv** are performed between GPUs without passing through a CPU-host. A similar picture occurs in connection with the **MPI_Allreduce()** function, in which the **MPI_Allreduce** operation is performed between a pair of GPUs (see .e.g. [65] for further details about a design of the **MPI_Allreduce** operation in GPU clusters).

```
!Unstructured data
!$acc enter data copyin(f)

!Performing MPI_send and MPI_Recv between GPUs without passing through
!the host
!$acc host_data use_device(f)

!transfer the data at the boundaries to the neighbouring MPI-process
!send f(:,nyp) from myid-1 to be stored in f(:,0) in myid+1
         if(myid.lt.nproc-1) then
          call MPI_Send(f(:,nyp), (nx+2), MPI_DOUBLE_PRECISION, myid+1, &
                 tag1, MPI_COMM_WORLD, ierr)
         endif

!receive f(:,0) from myid-1
         if(myid.gt.0) then
          call MPI_Recv(f(:,0),(nx+2),MPI_DOUBLE_PRECISION,myid-1, &
                        tag1, MPI_COMM_WORLD, status, ierr)
         endif

!$acc end host_data

!max_err is copied back to the CPU-host after performing the
!reduction operation

!$acc enter data copyin(max_err)
!Performing MPI_Allreduce between GPUs without passing through the host
!$acc host_data use_device(max_err)
         call MPI_ALLREDUCE(MPI_IN_PLACE, max_err, 1, &
              MPI_DOUBLE_PRECISION, MPI_MAX, MPI_COMM_WORLD, ierr)
!$acc end host_data
!$acc exit data copyout(max_err)
!$acc exit data copyout(f)
```

The same concept can be adopted in the hybrid MPI-OpenMP model. The GPU-aware MPI support with OpenMP can be implemented by specifying the directive **target data use_device_ptr(ptr-list)**. Here each array defined in the clause **use_device_ptr()** is a pointer to an object that is accessible on a GPU device [60]. The implementation in the MPI-OpenMP picture, in which the MPI functions **MPI_Send()** and **MPI_Recv()** can be performed between a pair of GPUs, is shown below.

```
!Unstructured data
!$omp target enter data device(myDevice) map(to:f)

!Performing MPI_send and MPI_Recv between GPUs without passing
!through the host
!$omp target data use_device_ptr(f)
```

```
!transfer the data at the boundaries to the neighbouring MPI-process
!send f(:,nyp) from myid-1 to be stored in f(:,0) in myid+1
         if(myid.lt.nproc-1) then
         call MPI_Send(f(:,nyp), (nx+2)*1, MPI_DOUBLE_PRECISION, myid+1, &
                tag1, MPI_COMM_WORLD, ierr)
         endif

!receive f(:,0) from myid-1
         if(myid.gt.0) then
          call MPI_Recv(f(:,0),(nx+2)*1,MPI_DOUBLE_PRECISION,myid-1, &
                        tag1, MPI_COMM_WORLD, status,ierr)
         endif

!$omp end target data
```

Similarly for the **MPI_Allreduce** operation; this is shown below.

```
!max_err is copied back to the CPU-host after performing
! the reduction operation

!$omp target enter data device(myDevice) map(to:max_err)
!Performing MPI_Allreduce between GPUs without passing through the host
!$omp target data use_device_ptr(max_err)
         call MPI_ALLREDUCE(MPI_IN_PLACE, max_err, 1, &
              MPI_DOUBLE_PRECISION, MPI_MAX, MPI_COMM_WORLD, ierr )
!$omp end target data
!$omp target exit data map(from:max_err)
!$omp target exit data map(from:f)
```

By comparing the syntax of the hybrid MPI-OpenACC API with that of the MPI-OpenMP API, one can see that the porting procedure from one API to another is straightforward.

3.2.8.3. Performance analysis

We carry out experiments based on solving the two-dimensional Laplace equation in a uniform grid iteratively and implementing the hybrid MPI-OpenACC and MPI-OpenMP APIs. The source code can be downloaded from the GitHub repository [66]. The compilation process is described in Section 3.2.8.4. Carrying out such experiments will help illustrating the benefit of implementing the GPU-aware MPI library. Our computational tests are performed on the supercomputer LUMI-G, in which each compute node is connected with 4xAMD MI250X GPUs via the Infinity Fabric Link, see Section 2.3, “LUMI-G: GPU partition”

We first begin with investigating the benefit of implementing the GPU-aware MPI with the OpenACC API. This is shown in Fig. 10, in which the computations are performed on 4 GPUs (black curve). For reference, the computations based on pure MPI are also shown (blue curve). Interesting enough, one can see clearly that the computing time (in second) is reduced by a factor of 10 when the GPU-aware support is enabled compared to the case of the non-GPU-aware MPI (green curve). Moreover, the comparison shows a further decrease of the computing time by a factor of about 10.

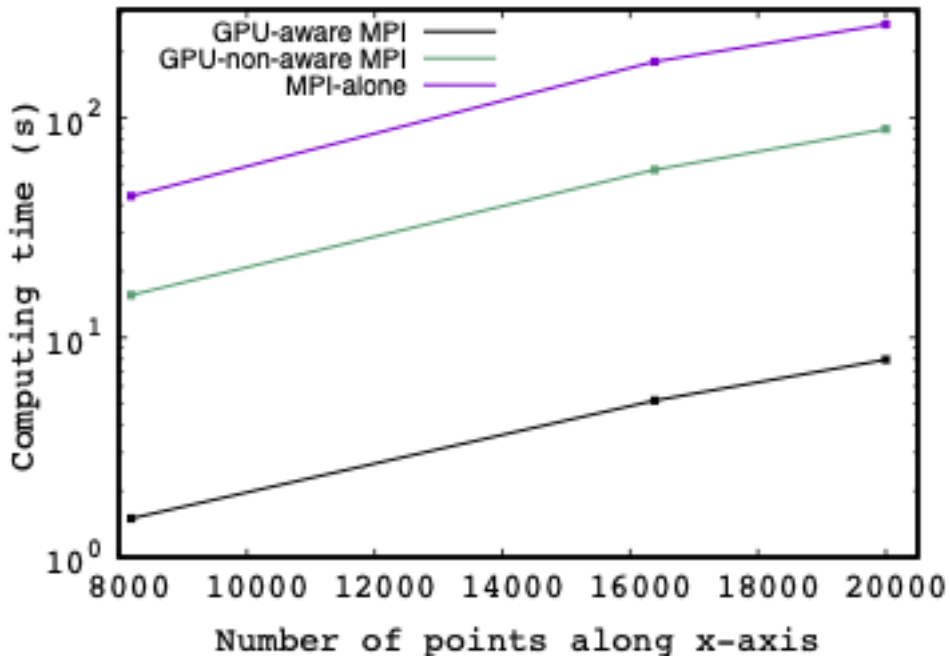

Figure 10. Computing time (given in second) as a function of the number of points nx along the x-axis. The computations are based on the MPI-OpenACC model and MPI-alone (see text for further description). The legend indicates different approaches. The square symbols are used for guidance

For completeness, we present in Table 11, “Comparison of the computing time (given in second) between the hybrid MPI-OpenACC and MPI-OpenMP APIs at three different grids increased in size. The comparison is performed for both GPU-aware MPI and non-GPU-aware MPI.” a comparison of the computing time (in second) between the hybrid MPI-OpenACC and MPI-OpenMP APIs. The comparison is evaluated for both the GPU-aware MPI and the non-GPU-aware MPI and is shown at different sizes of the spatial 2D-grid (nx,ny). The comparison is summarised in the table below and shows a roughly similar performance between these two hybrid models. These results indicate, on the one hand, the benefit of implementing the GPU-awareness approach independently of the GPU directive-based model; and on the other hand, they highlight the similarity in the performance of the MPI-OpenACC and MPI-OpenMP APIs.

Table 11. Comparison of the computing time (given in second) between the hybrid MPI-OpenACC and MPI-OpenMP APIs at three different grids increased in size. The comparison is performed for both GPU-aware MPI and non-GPU-aware MPI.

Hybrid models / 2D-grid (nx,ny)	8192x8192	16384x16384	20000x20000
MPI-OpenACC with GPU-aware MPI	1.46 (secs)	5.11 (secs)	7.92 (secs)
MPI-OpenMP with GPU-aware MPI	1.35 (secs)	4.72 (secs)	7.09 (secs)
MPI-OpenACC with GPU-Non-aware MPI	14.90 (secs)	58.03 (secs)	86.25 (secs)
MPI-OpenMP with GPU-Non-aware MPI	14.84 (secs)	61.21 (secs)	85.58 (secs)

3.2.8.4. Compilation process

In this section we summarise the compilation process of a hybrid application that combines MPI with OpenACC/OpenMP API. Here it is possible to incorporate both OpenACC and OpenMP directives in the same code, such that the directives are defined using the preprocessor directive **#ifdef** (or also **#if defined**). A code example is provided in [66]. This procedure, in general, enables compiling the same code ported to multiple programming

models. In this context, different options can be used to specify preprocessing of source files, and that according to the used HPC system, as described below.

Here we consider hybrid MPI-OpenACC and MPI-OpenMP applications written in Fortran 90. The compilation process on LUMI-G using a Cray compiler of the wrapper **ftn** is described in the following (see the LUMI documentation [68] for further details about the programming environment available on LUMI):

```
$ ftn -eZ -D_OPENACC/D_OPENMP -hacc/homp -o source.exe source_code.f90
```

where we use the conditional compilation with the macros **_OPENACC** and **_OPENMP**. This is enabled in the Cray compiler by specifying the option **-eZ** followed by either **-D_OPENACC** to enable OpenACC directives or **-D_OPENMP** to enable OpenMP directives. The flags **hacc** and **homp** enable the OpenACC and OpenMP directives in the hybrid MPI-OpenACC and MPI-OpenMP applications, respectively.

Note that the GPU-aware support in MPICH library is enabled by setting the environment **export MPICH_GPU_SUPPORT_ENABLED=1** on Cray before running a hybrid application.

3.2.8.5. Conclusion

In conclusion, we have presented an overview of the GPU-hybrid programming by integrating GPU directive-based models, specifically OpenACC and OpenMP APIs, with the MPI library. The approach adopted here allows, in general, to utilise multiple GPU devices not only within a single GPU node but it extends to multiple GPU partitions. It thus allow a richer and more flexible optimisation of the communications between the MPI ranks. In particular, we have addressed both non-GPU-aware MPI and GPU-aware MPI approaches. The latter approach has the advantage of enabling a direct interaction between an MPI library and a GPU device memory. In other words, it permits performing MPI operations between a pair of GPUs, thus reducing the computing time caused by the data locality. We have carried out experiments on the LUMI-G by developing a mini-application based on solving the 2D-Laplace equation. Here we have observed an increase of the performance by a factor of 10 when implementing the GPU-aware MPI scheme and by almost a factor of 30 in comparison with the MPI alone.

3.2.9. Bifrost case study

3.2.9.1. Introduction

This section tells the story of the work to take advantage of the compute power of the GPUs. It is a case study included to illustrate the effort needed to take advantage of the GPU compute units.

Bifrost [44] is a stellar atmosphere simulation code. Bifrost is written in Fortran and uses MPI for distributed-parallel execution.

The application is memory bandwidth-bound. Analysis using tools that generate a roofline model clearly indicate that Bifrost is in the memory bound region. A nice illustration using a very different approach can be found in the PRACE Best Practice Guide «Modern-Processors-Accelerators» [21] page 94.

Bifrost showed very good performance on Intel Knights Landing systems, see the «Knights Landing» PRACE BPG [23] page 36. A similar code to Bifrost is run on the Fugaku supercomputer [24] with very good results. This system is known to have very high memory bandwidth (1 TB/s, all HBM2 memory). In addition Bifrost showed very good performance using the Apple M1 processor.

Profiling has shown that there are no obvious «hot» sections where a lot of compute time is spent. The load profile is relatively flat. This makes adaptation to GPU somewhat harder, as the major part of the data structures needs to be accessible for the major part of execution. It is not a simple matter to move data back and forth as the device memory is far smaller than main memory.

3.2.9.2. Adapting to GPU accelerators

Bifrost has been continuously developed and changed significantly since the first OpenACC version created with assistance from NVIDIA. I took the current development (CPU only) branch and created a new «develop-gpu-OpenACC» branch which ports the MHD solver of Bifrost into GPUs. The past version served as a reference, but

the new version is kept up to date with ongoing developments and is compatible with both CPUs and GPUs. The approach we take is to move all the data to the GPU(s) and keep it there while doing the calculations, which reduces the time spent in data movement. Some of the functions were refactored to utilize the parallelisation for GPUs.

Around a year ago we weren't sure about the different compilers and support for AMD GPUs, so we decided to make a parallel branch «develop-gpu-OpenMP». We converted the data motion to have the data in the device and kernel directives into target regions. For example, in the case of three nested loops we use: **!$omp target teams distribute parallel do collapse(3)** Having this in place, core developers attended the PRACE Autumn School in Finland where they had testing hardware and software in preparation for LUMI-G. From this experience, using Cray compilers, we noticed we were able to compile, run and get the correct results with our OpenMP implementation. However, this simple version did not achieve any better performance.

During the early access platform with MI100s, using the Cray compilers and our OpenMP version, we decided to tackle a larger setup. In the past we had been doing ideal magnetohydrodynamics (MHD) experiments in 2D and 3D, so this time we used a larger setup which is a benchmark for different solar physics codes. This benchmark had a 512x512x128 grid but we did not see any increased performance, although we are aware improvements on optimizing the OpenMP implementation are feasible. Recently, with the MI250X in LUMI the Cray compilers support OpenACC really well, we tested our updated OpenACC version with the main difference between the LUMI-G and systems using NVIDIA hardware/software in our local machines being the removal of async directives. Now we can get performance of approximately 2 times for 256^3 and approximately 5 times for 512^3 grids, for one GPU compared to the CPU in one node of LUMI-G. Multi-GPU runs do not seem to scale ideally, likely due to the problem being too small. In the remaining pilot weeks we are investigating a larger grid, and adding additional physics to prepare a problem with some scientific implication.

3.2.9.2.1. Experiences with LUMI-G

Experiments during the early access platform and pilot phases on LUMI G show that what works best for Bifrost is OpenACC with Cray compilers (ftn) for AMD MI250X GPUS.

The modules loaded were:

- CrayEnv
- PrgEnv-cray
- craype-accel-amd-gfx90a
- rocm

with the compiler flags : «-O3 -h acc_model=auto_async_none:no_fast_addr:deep_copy -eZ».

The asynchronous version that we had for NVIDIA GPUs does not work with the Cray compiler on LUMI. Attempting to debug using CRAY_ACC_DEBUG=1 did not help too much since the run would stop and crash at different places. Regardless of removing the async directives, the code performs faster than the CPU version (approximately 2 times for 256^3 and approximately 5 times for 512^3 grids, for one GPU), much faster than the OpenMP version (approximately 15 times) and faster than the async version on the NVIDIA GPU(approximately 5 times).

The following table shows running times for the blast experiment and the Whole Sun (WS) high resolution (HR) benchmark. The blast experiment is the typical 3D MHD spherical blast. The WS HR is a Hydrodynamics problem with a 512x512x128 grid. The CPU runs in LUMI were done in one CPU of the LUMI G partition node, using 64 cores (with MPI communication). The OpenMP and OpenACC times shown here are for one GPU.

Table 12. Bifrost performance (run times is seconds, lower is better)

Size	OpenMP (MI100)	OpenMP (MI250X)	OpenACC (MI250X)	CPU
Blast 64^3	4.6242	6.0258	1.0239	0.19275
Blast 256^3	133.00	163.23	7.1897	16.259
WS $HR64^3$	149.89	195.78	10.054	25.016
Blast 512^3	n.d.	n.d.	29.527	136.17

A few runs with multiple GPUs in one node are conducted, having one MPI rank per GPU. Unfortunately we don't see ideal scaling, therefore we will be running more experiments with a larger size setup and different bindings.

3.2.9.2.1.1. Conclusion

The performance achieved for one GPU is modest but if we can manage to achieve good scaling for one node then the running time could be up to 40 times faster (5 x 8 GPUs) in comparison to the 64 cores, and ideally continue the trend for multiple nodes. It is important to note that most of our directives are «acc kernels» type and there is room for optimization. Although to get larger speedups one probably will have to explore technologies closer to the hardware such as HIP/CUDA.

3.3. Optimizing structured grid applications on LUMI-C

Consider the details of the LUMI-C architecture: the non-accelerated part of the cluster consists of dual-socket compute nodes with AMD Epyc Milan CPUs, interconnected using 200 Gbits/s Slingshot 11. One CPU has 4 NUMA nodes, each NUMA node consists of 2 Core Complexes (CCX), each with 8 cores, for a total of 64 cores. The cores within the same CCX share an L3 cache. CCX's and sockets are interconnected with the Infinity Fabric. Understandably, memory access speed in this hierarchical NUMAcc architecture depends on where the process is running, and which memory domain holds the data.

During parallel computations, both the compute-node hardware and the software perform data movement and synchronization to coordinate the in-node and off-node communication between the processes and threads. Although the in-node communication is largely implicit and done in hardware (except of accelerator access), it is not much different to the off-node communication over the interconnect: PCI, Infinity, and NVLink (for NVIDIA based systems) all have latency/bandwidth characteristics similar to the interconnect, and can all exhibit congestion. To optimize the software and the runtime configuration for such complex systems it is required to understand the communication cost between all those system components, and take them into account. We look at the impact of such optimizations on the example of parallel structured grid computations.

3.3.1. Mapping MPI ranks to CPU cores

A common example of HPC applications are grid-based Partial Differential Equation (PDE) solvers, where the physical model domain is discretised using a structured grid, and the individual sub-domains are assigned to CPU cores for calculations. To satisfy the PDE across the sub-domain boundaries the Processing Elements (PEs) have to exchange the field value at the boundary with their spatial neighbours. This operation is known as the halo-exchange (HE), and is a fundamental component of parallel Finite Difference, Finite Volume and Finite Element PDE solvers.

When running a parallel PDE solver on an HPC system, one has to perform the domain decomposition (split the physical domain into disjoint sub-domains) and map the individual sub-domains to PEs (e.g. CPU cores, GPUs) for computing. It is nowadays common knowledge that to maximize the processing speed and improve software scalability on massively parallel HPC machines one should minimize the communication over the interconnect. In this particular case it is necessary to perform the mapping in such a way as to maximize the number of in-node neighbours, consequently performing parts of the MPI communication over shared memory instead of the interconnect.

In practice, for structured grids a common approach is to perform the mapping using an MPI Cartesian communicator, which conveniently resolves the Cartesian neighbourhood of the sub-domains. Consider a pure MPI application, in which one grid sub-domain is assigned to each MPI rank. A Cartesian communicator is created using `MPI_Cart_create`, which remaps the original rank numbers onto a Cartesian grid of ranks. The user specifies the rank grid dimensions, e.g. `[nx, ny, nz]` in 3D, where `np=nx*ny*nz` is the total number of ranks. The Cartesian communicator provides a conversion between the linear rank number and the Cartesian rank coordinates, and makes it trivial to find each rank's spatial neighbours for the purpose of halo exchange. However, the details of how the ranks are mapped onto the hardware topology depends on the MPI runtime. The most common MPI implementations (OpenMPI, MPICH) do not take the details of the hardware topology into account.

While the above approach has some potential to decrease the off-node communication, it is not optimal: 1) with the current MPI implementations it does not result in optimal surface-to-volume ratios for the in-node sub-domains,

and 2) it does not take the architectural details of the compute nodes into account. Just as it is beneficial to distribute the sub-domains as to maximize the number of in-node neighbours, the same holds for the sub-domains inside each NUMA node, L3 cache, and possibly on other levels of the memory hierarchy. HWCART is developed within the the GHEX project [46]. [45] - a hardware-aware Cartesian MPI communicator, which allows the user to map the MPI ranks to the memory domains in a way that minimizes the in-node (shared memory) and the off-node (interconnect) data exchanges. Consider the example of LUMI-C, which can be viewed as a 5 level hierarchy of memory domains. At the lowest level of the hierarchy are the cores. Groups of 8 cores belong to a CCX (L3 cache domain), 2 CCX modules belong to a single NUMA node, 4 NUMA nodes comprise a socket, finally there are 2 sockets on each compute node. HWCART discovers the memory domain hierarchy using the standard HWLOC library [48] and enables the user to define an arrangement of the lower-level domains into sub-grids inside the higher-level domains. For example, 8 cores inside a CCX could be arranged into a `[2, 2, 2]` grid, or a `[4, 2, 1]` grid. Two CCX units can then be arranged into a `[2, 1, 1]` grid within a NUMA node, and so on. Assuming we run a 3D code on 2048 compute nodes, one of the possible HWCART topology definitions is shown in Figure 11. By testing different rank mappings one can find the most optimal layout that optimizes the given application run time.

```
    int topo[3*NLEVELS] = {
      4,  2,  1,   // core grid within each CCX / L3 cache domain
      2,  1,  1,   // l3cache grid
      1,  2,  2,   // numa grid
      1,  1,  2,   // socket grid
      8, 16, 16    // compute node grid
  };
```

Figure 11. An example multi-level HWCART topology definition for a cluster built of 2-socket AMD Epyc 3 compute nodes. In total, 262144 ranks are mapped onto a 64^3 Cartesian rank grid. Each compute node runs 128 ranks arranged into 8x4x4 grids.

3.3.2. Using HWCART

HWCART currently provides a C/C++ and a Fortran API, and requires only minimal changes (fewer than 20 lines of code) to existing codes based on a native `MPI_Cart`. Below is a simplified Fortran example that constructs an MPI-compatible Cartesian communicator with the memory hierarchy-aware rank-to-core mapping. The complete examples can be found on HWCART website [45].

```
! number of memory domain levels to consider
integer, parameter :: NLEVELS=5

! memory domains
integer, dimension(:), target :: domain(NLEVELS) = [   &amp;
  HWCART_MD_CORE,    &
  HWCART_MD_L3CACHE, &
  HWCART_MD_NUMA,    &
  HWCART_MD_SOCKET,  &
  HWCART_MD_NODE]

! hierarchical rank grid structure of the memory domains
integer, dimension(:,:), target :: topo(3,NLEVELS) = reshape([ &
  4, 2, 1, & ! core grid
  2, 1, 1, & ! l3cache grid
  1, 2, 2, & ! numa grid
  1, 1, 2, & ! socket grid
  1, 1, 1],& ! node grid
  shape(topo))
```

The above is an example arrangement of 64 ranks within a single compute node, suitable for the LUMI-C CPU architecture. Using HWCART is almost identical to using the MPI Cartesian communicator:

```
! initialize MPI
call MPI_Init(ierr);

! initialize HWCART and discover the topology
ierr = hwcart_init(hwcart_topo)

! Create the HWCART communicator using the discovered topology
! and the previously defined domain hierarchy
! returns: hwcart_comm, an MPI communicator.
ierr = hwcart_create(hwcart_topo, MPI_COMM_WORLD, domain, topo,
                     cart_order, hwcart_comm)

! Convert a HWCART communicator to a native MPI_Cart communicator.
! All standard MPI_Cart functions (MPI_Cart_shift, MPI_Cart_sub,
! MPI_Cart_rank, etc.) can be used on mpicart_comm.
ierr = hwcart2mpicart(hwcart_comm, topo, periodic, cart_order, &
                      mpicart_comm)

! obtain MPI_Cart Cartesian coordinates
call MPI_Comm_rank(mpicart_comm, mpicart_rank, ierr);
call MPI_Cart_coords(mpicart_comm, mpicart_rank, 3, mpicart_coord, ierr);

! cleanup
call MPI_Comm_free(mpicart_comm, ierr)
call hwcart_comm_free(hwcart_comm)
call hwcart_topo_free(hwcart_topo)
call MPI_Finalize(ierr);
```

The `mpicart_comm` communicator created above can be used in any MPI function that operates on a Cartesian communicator. The difference is that the rank-to-core mapping in that communicator is memory hierarchy-aware, as defined by the user.

3.3.3. Performance

Figure 12 demonstrates the effect of using HWCART compared to the standard `MPI_Cart` in single-node and 27-node tests, for a fully periodic 3D grid of 128^3 double precision values per rank. For benchmarking purposes we used GHEXBENCH [47]. The MPI implementation is a pure MPI code that exchanges the halo in 3 dimensions. In addition, the GHEX variant uses the algorithms and optimizations developed within the GHEX project [46]. In particular, XPMEM was used in the GHEX code to completely avoid MPI communication between ranks running on the same compute node: ranks, just as threads, could access each other's data structures directly, through a pointer. In the single-node tests, 128 ranks were arranged into a periodic `[8, 4, 4]` rank grid. In the 27-node runs the global rank grid dimensions were `[24, 12, 12]`.

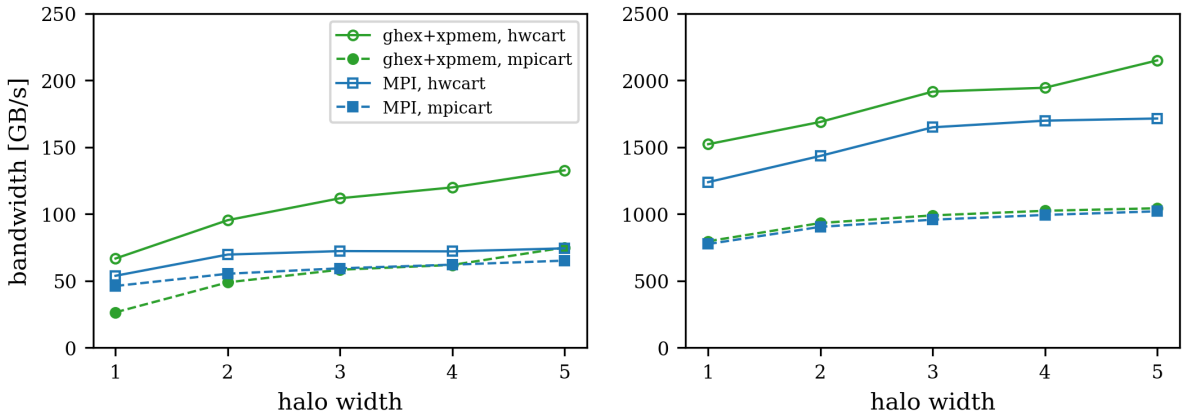

Figure 12. Halo exchange performance with GHEX and native MPI, using HWCART and `MPI_Cart`. Left: 1 compute node, 128 ranks. Right: 27 compute nodes, 3456 ranks.

The impact of HWCART on HE performance within a single compute node was up to 30%. It was much more pronounced for the multi-node case: on 27 compute nodes using HWCART instead of `MPI_Cart` in the pure MPI code improved the performance by 60%. Note also that HWCART was crucial in order to benefit from XPMEM and GHEX. There was essentially no improvement when using XPMEM with the standard `MPI_Cart`, but using both HWCART and XPMEM improved the performance of HE by factor 2 compared to a pure MPI implementation. This clearly shows the importance of HWCART and the hierarchical rank to memory domain mapping for both in-node, and off-node communication performance.

Figure 13 presents the weak scaling of BIFROST - a pure MPI, explicit Finite Difference, stellar atmosphere simulation code that solves the standard MHD partial differential equations on a staggered Cartesian grid [44]. The tests were run on LUMI-C on up to 1331 compute nodes, and on Betzy - a similar system based on AMD Epyc 2 and an InfiniBand interconnect. On both architectures each compute node ran 64 MPI ranks, i.e. only every second core was used. This strategy decreased the pressure on the memory bandwidth and resulted in a lower time to solution on both systems. The model domains were cubes, with the same number of ranks in each spatial dimension. The GHEX benchmarks used HWCART with the earlier described hierarchical rank to CPU mapping modified to reflect the fact that only half of the cores were used. Overall, GHEX with HWCART improved the iteration time (and consequently the total application execution time) significantly: around 13% on Betzy, and over 20% on LUMI-C. In addition, the scalability of the code was improved due to HWCART and a hardware-aware rank-to-node, and rank-to-core mapping.

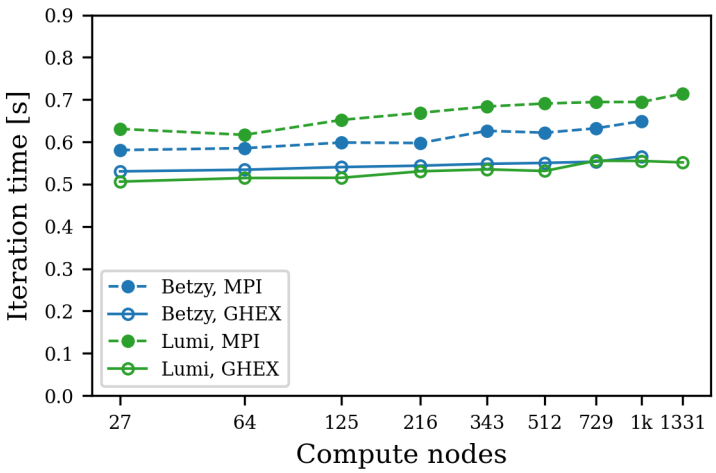

Figure 13. Weak scaling of Bifrost with GHEX and native MPI halo exchange on LUMI-C and Betzy (a similar system in Norway). Single-precision, 64^3 grid points per rank.

3.4. Case study: Porting Matrix-Matrix Multiplication from CPU to GPU, Step by Step

This is a step by step story on how to get starting with porting a CPU only to code to take advantage of GPU acceleration.

We have prepared a series of small didactic programs performing the multiplication of dense matrices. Our examples proceed applying optimizations and transformations, incrementally. The idea is to evolve from CPU to GPU code gradually. The only prerequisite is basic OpenMP knowledge. For a textbook introduction to cache-based microprocessor architectures, data access optimization and efficient OpenMP programming see [29]. We are interested in large matrices exceeding the size of the lowest level caches. We also limit ourselves to matrix sizes that are multiples of specific blocking sizes. For this reason we will show the kernels' loops only. The examples are independent of numerical type, hence it will appear as *n_t*. The literature on general matrix multiplication is vast and evolving. In contrast, this tutorial is only a limited introduction to a handful of the most common optimizations. Their effectiveness is to be measured with opportune tools (timing, code inspection, execution debug run, etc). Our starting point is the following kernel:

```
// Excerpt: MatMatMul_CPU___serial

for(int i=0;i&lt;N;++i)
    for(int k=0; k<N; ++k)
        for(int j=0; j<N; ++j)
            c[N * i + j] += alpha * a[N * i + k] * b[N * k + j];
```

In GEMM (General Matrix-Matrix multiplication) literature parlance, one refers to such as a "IKJ" loop. This kernel performs NxNxN updates of output matrix C, by contiguous write accesses; repeats N times the use of each element in A, performing NxN effective accesses A, by contiguous read accesses, and; repeats N times the entire read of matrix B (NxN accesses), for a total of NxNxN accesses. This kernel is apt to the simplest parallelisation by means of OpenMP, namely adding a work-partitioning directive:

```
// Excerpt: MatMatMul_CPU___openmp

#pragma omp parallel for
for(int i=0; i<N; ++i)
   for(int k=0;k<N;++k)
      for(int j=0;j<N;++j)
         c[N * i + j] += alpha * a[N * i + k] * b[N * k + j];
```

Here it is expected that each thread gets assigned a similar amount of contiguous indices. This leads each thread to contribute into a disjoint rows interval of c. This parallelisation scheme (called "by rows") is simple and requires no further synchronization mechanism. The OpenMP construct to request operation on a GPU device is the *target* directive. Since the GPU has a memory of its own, it is necessary to specify which arrays to transfer there. The simplest method here is that of using the *map* directive. Since it is the user's choice to determine which arrays to reuse after this device loop, one must also explicitly specify which arrays to copy back to the CPU memory. Once the data is mapped, one can distribute work on the GPU 'teams' with the *teams* directive.

```
// Excerpt: MatMatMul_GPU___openmp

#pragma omp target map(to:alpha,a[:N*N],b[:N*N]) map(from:c[:N*N])
#pragma omp teams default(shared)
#pragma omp distribute parallel for
for(int i=0; i<N; ++i)
    for(int k=0; k<N; ++k)
```

```
        for(int j=0; j<N; ++j)
            c[N * i + j] += alpha * a[N * i + k] * b[N * k + j];
```

Copying array data into and back from the device is expensive. This can be mitigated by separating the memory transfer directives from the computing sections. In a real application one may want to place the input, the computation, and the output steps in separate functions. The following excerpt features three sections which can reside in different functions. The first section is to copy the array data on the device; the second is to perform the computation; the third and last one to copy selected data back to the CPU memory.

```
// Excerpt: MatMatMul_GPU___data_0

#pragma omp target enter data map(to:a[:N*N],b[:N*N])
#pragma omp target enter data map(to:c[:N*N])
...
#pragma omp target teams distribute
for(int i=0; i<N; ++i)
    for(int k=0; k<N; ++k)
#pragma omp parallel for
        for(int j=0; j<N; ++j)
            c[N * i + j] += alpha * a[N * i + k] * b[N * k + j];
...
#pragma omp target exit data map(from:c[:N*N])
```

Another relevant change is that we use a different workload distribution. Having the *omp parallel for* directive one level earlier (on *k*) would incur in a data race. For the same reason a solution like the following:

```
#pragma omp target nowait
#pragma omp teams distribute parallel for collapse(2)
for(int i=0; i<N; ++i)
    for(int k=0; k<N; ++k)
    ...
#pragma omp barrier
```

which matches closer the work-partitioning scheme in *MatMatMul_GPU___openmp* can not be used in this specific kernel. Kernels which follow, instead, could use a two-level combined work-partitioning, but we observed performance to be inferior than with this layout, especially on smaller matrices. On the CPU as well as on the GPU, the above loops lack any blocking optimization. That limits potential performance, as due to the way memory is being loaded (in bulk), a significant portion of the cached memory from operands a and b ends up being discarded after load. In the kernel listings that follow, we will omit showing the *enter data* and *exit data* clauses. In the following listing, we introduce an additional hierarchy of loops, so to have blocking optimization. Now there are three loops acting on blocks of indices, and three loops acting on a single block level.

```
// Excerpt: MatMatMul_GPU___data_1

#pragma omp target teams distribute
for(int bi=0; bi<N/ibs; ++bi)
#pragma omp parallel for
for(int bj=0;bj<N/jbs;++bj)
for(int bk=0;bk<N/kbs;++bk)
for(int ii=0;ii<ibs;++ii)
for(int jj=0;jj<jbs;++jj)
{
 const int i = bi * ibs + ii;
```

```
 const int j = bj * jbs + jj;
 n_t acc = 0;
 for(int kk=0; kk<kbs; ++kk)
 {
  const int k = bk * kbs + kk;
  acc += alpha * a[N * i + k] * b[N * k + j];
 }
 c[N * i + j] += acc;
}
```

In the above, the parallelisation scheme is the same as in the previous listing: by blocks of consecutive rows. Because of this, reordering of the inner loops is possible without any conflict. This listing also introduces the use of an accumulator variable for the innermost loop; its presence is a suggestion to the compiler to use a register instead of a memory location. So far our listings are compatible with CPU as well as GPU. The choice of blocking variables *ibs*, *jbs*, and *kbs* can strongly impact performance on different devices. An optimal choice is not trivial, and may require a mix of modeling and experiments. In the particular case above, another limitation of the current approach emerges. That is, the index space is more limited, because *N/ibs* is to be subdivided among OpenMP teams and threads alike. For this reason our next step introduces 2D parallelism, via the *collapse* clause. Having previously detected the number of teams, these are now mapped onto tj and ti. In addition to that, we introduce explicit cache blocking of the operands by means of small helper arrays. The um2v (*untransposed matrix to vector*) helper function copies a *ibs* x *kbs* submatrix from *a* onto the stack into *aa*; similarly for a *kbs* x *jbs* submatrix of *b*, into *bb*.

```
// Excerpt: MatMatMul_GPU___data_2

#pragma omp target teams distribute
for(int ti=0;ti<its;++ti)
#pragma omp parallel for
for(int tj=0;tj<jts;++tj)
for(int bi=ti*itb;bi<(ti+1)*itb;++bi)
for(int bk=0;bk<N/kbs;++bk)
{
 std::array<n_t,ibs*kbs> aa;
 um2v<n_t,ibs,kbs>(aa, a, N, bi*ibs, bk*kbs);

 for(int bj=tj*jtb;bj<(tj+1)*jtb;++bj)
 {
  std::array<n_t,kbs*jbs> bb;
  um2v<n_t,kbs,jbs>(bb, b, N, bk*kbs, bj*jbs);

  for(int ii=0;ii<ibs;++ii)
  for(int jj=0;jj<jbs;++jj)
  {
   const int i = bi * ibs + ii;
   const int j = bj * jbs + jj;
   n_t acc = 0;

   for(int kk=0;kk<kbs;++kk)
   {
    acc += alpha * aa[kbs * ii + kk] * bb[jbs * kk + jj];
   }
   c[N * i + j] += acc;
  }
 }
}
...
```

```
template <class N_T,int N_R,int N_C>
void um2v(std::array<N_T,N_R*N_C> &amp;amp; dm, const N_T *sm, const int lda,
   const int aro, const int aco)
{
 // untransposed rectangular contiguous submatrix to vector copy
 for(int ii=0;ii<N_R;++ii)
  for(int kk=0;kk<N_C;++kk)
   dm[N_C * ii + kk] = sm[lda * (aro + ii) + (aco + kk)];
}
```

Introduction of aa and bb as copies of the current blocks in a and b is a step further in increasing memory reuse. Unfortunately, that makes the code significantly more verbose. However, we continue with this technique. On the top of the arrays introduced so far (for cache blocking), one can introduce even smaller arrays, and use these in the innermost loops. If effective, this technique is called 'register blocking'. We also introduce a "cache block" for the output matrix. The two new extra helper functions tm2v and umr2v are defined just after the kernel listing.

```
// Excerpt: MatMatMul_GPU___data_3

#pragma omp target teams distribute
for(int ti=0;ti<its;++ti)
#pragma omp parallel for
for(int tj=0;tj<jts;++tj)
for(int bi=ti*itb;bi<(ti+1)*itb;++bi)
for(int bk=0;bk<N/kbs;++bk)
{
 std::array<n_t,ibs*kbs> aa;
 um2v<n_t,ibs,kbs>(aa, a, N, bi*ibs, bk*kbs);

 for(int tbj=tj*jtb;tbj<(tj+1)*jtb;++tbj)
 {
  const int bj = tbj;
  std::array<n_t,jbs*kbs> bbt;
  tm2v<n_t,jbs,kbs>(bbt, b, N, bk*kbs, bj*jbs);

  for(int ii=0;ii+iss-1<ibs;ii+=iss)
  for(int jj=0;jj+jss-1<jbs;jj+=jss)
  {
   std::array<n_t,iss*jss> cc;
   const int io = bi * ibs + ii;
   const int jo = bj * jbs + jj;

   for(int vi=0;vi<iss;++vi)
    for(int vj=0;vj<jss;++vj)
    {
     const int i = vi + io;
     const int j = vj + jo;

     cc[jss * vi + vj] = c[N * i + j];
    }

   for(int kk=0;kk+kss-1<kbs;kk+=kss)
   {
    std::array<n_t,kss> av[iss];
    umr2v<n_t,iss,ibs,kbs,kss>(av, aa, ii, kk);
    std::array<n_t,kss> bvt[jss];
    umr2v<n_t,jss,jbs,kbs,kss>(bvt, bbt, jj, kk);
```

```
    for(int vi=0;vi<iss;++vi)
     for(int vj=0;vj<jss;++vj)
      for(int vk=0;vk<kss;++vk)
       cc[jss * vi + vj] += alpha * av[vi][vk] * bvt[vj][vk];
   }

   for(int vi=0;vi<iss;++vi)
    for(int vj=0;vj<jss;++vj)
    {
     const int i = vi + io;
     const int j = vj + jo;

     c[N * i + j] = cc[jss * vi + vj];
    }
  }
 }
}
...
template <class N_T,int N_R,int N_C>
void tm2v(std::array<N_T,N_R*N_C> &amp;amp; dm, const N_T *sm, const int lda,
   const int aro, const int aco)
{
 // transposed rectangular contiguous submatrix to vector copy
 for(int ii=0;ii<N_C;++ii)
  for(int kk=0;kk<N_R;++kk)
   dm[N_C * kk + ii] = sm[lda * (aro + ii) + (aco + kk)];
}

template <class N_T,int D_R,int S_R,int S_C,int V_L>
inline
void umr2v(std::array<N_T,V_L> dv[D_R], const std::array<N_T,S_R*S_C> &amp;amp; sm,
   const int sro, const int sco)
{
 // untransposed matrix' rectangular submatrix to vector rows copy
 for(int ii=0;ii<D_R;++ii)
  for(int vj=0;vj<V_L;++vj)
   dv[ii][vj] = sm[S_C * ( sro + ii ) + (sco + vj)];
}
```

The following kernel introduces a lower level of blocking for the result operands, and rearranges the loops.

```
// Excerpt: MatMatMul_GPU___data_4

#pragma omp target teams distribute
for(int ti=0;ti<its;++ti)
#pragma omp parallel for
for(int tj=0;tj<jts;++tj)
for(int bi=ti*itb;bi<(ti+1)*itb;++bi)
for(int bj=tj*jtb;bj<(tj+1)*jtb;++bj)
{
 std::array<n_t,ibs*jbs> cc;
 std::array<n_t,jbs*kbs> bbt;
 std::array<n_t,kss> av[iss];
 std::array<n_t,kss> bvt[jss];
 std::array<n_t,ibs*kbs> aa;
```

```
 std::array<n_t,jss> cv[iss];

 for(int ii=0;ii<ibs;ii++)
 for(int jj=0;jj<jbs;jj++)
  cc[jbs * ii + jj] = 0;

 for(int sbk=0;sbk<N/kbs;++sbk)
 {
  const int bk = sbk;

  tm2v<n_t,jbs,kbs>(bbt, b, N, bk*kbs, bj*jbs);
  um2v<n_t,ibs,kbs>(aa, a, N, bi*ibs, bk*kbs);

  for(int ii=0;ii+iss-1<ibs;ii+=iss)
  for(int jj=0;jj+jss-1<jbs;jj+=jss)
  for(int kk=0;kk+kss-1<kbs;kk+=kss)
  {
   umr2v<n_t,iss,ibs,kbs,kss>(av, aa, ii, kk);
   umr2v<n_t,jss,jbs,kbs,kss>(bvt, bbt, jj, kk);

   for(int vi=0;vi<iss;++vi)
    for(int vj=0;vj<jss;++vj)
    {
     cv[vi][vj] = 0;
     for(int vk=0;vk<kss;++vk)
      cv[vi][vj] += av[vi][vk] * bvt[vj][vk];
    }
   for(int vi=0;vi<iss;++vi)
    for(int vj=0;vj<jss;++vj)
    {
     const int i = ii + vi;
     const int j = jj + vj;
     cc[jbs * i + j] += cv[vi][vj];
    }
  }
 }
 for(int ii=0;ii<ibs;ii++)
 for(int jj=0;jj<jbs;jj++)
 {
  const int i = bi * ibs + ii;
  const int j = bj * jbs + jj;

  c[N * i + j] += alpha * cc[jbs * ii + jj];
 }
}
```

Our parallelisation scheme so far foresees two memory areas on the GPU: the device memory, and the per-threads memory. That is, we are not exploiting yet the memory that each team of threads can share. For that, let us see how kernels written in HIP (or CUDA) look like. Consider the following HIP kernel from AMD [74] available as of 2022.12.01 on which we paste in its original form here:

```
/// \brief Multiplies matrices \p A and \p B and stores the result to \p C.
/// - The number of rows of the result matrix is equal to the number of
///   rows of matrix A which is \p blockDim.y*gridDim.y.
/// - The number of columns of the result matrix is equal to the number
///   of columns of matrix B which is \p blockDim.x*gridDim.x.
```

```
/// - The number of columns of matrix \p A is passed as argument.
/// - The matrix elements are stored in a row-major order.
///
/// - Each thread in the grid is responsible for one element of the
///   result matrix.
/// - Each element is calculated cooperatively in a tiled manner.
///   In each step, a BlockSize*BlockSize tile is loaded to the shared
///   memory so individual threads can address this shared cache instead
///   of loading the same values from the global device memory
///   individually.
///   The end result is accumulated through each step on a per-thread
///   basis.
/// - The matrix dimensions are assumed to be multiples of the block
///   size for simplicity.
template<unsigned int BlockSize>
__global__ void matrix_multiplication_kernel(const float*       A,
                                             const float*       B,
                                             float*             C,
                                             const unsigned int a_cols)
{
    const unsigned int tx = threadIdx.x;
    const unsigned int ty = threadIdx.y;
    const unsigned int bx = blockIdx.x;
    const unsigned int by = blockIdx.y;

// b_cols must match the number of output matrix columns.
    const unsigned int b_cols = blockDim.x * gridDim.x;

// The number of tiles is determined by A's columns (which is
// equal to B's rows).
    const unsigned int steps = a_cols / BlockSize;

    // thread_result is the accumulation variable.
    float thread_result = 0.0F;
    for(unsigned int step = 0; step < steps; step++)
    {
// Shared memory is used to cache the tile from both input matrices.
// The tile is a square of BlockSize*BlockSize.
 __shared__ float a_values[BlockSize][BlockSize];
 __shared__ float b_values[BlockSize][BlockSize];

 // Index of the top-left element of the tile in A.
 // "BlockSize * a_cols * by" is the number of elements to move "down".
 // "BlockSize * step" is the number of elements to move "right".
 const unsigned int a_idx = BlockSize * (a_cols * by + step);

 // Index of the top-left element of the tile in B.
 // "BlockSize * b_cols * step" is the number of elements to move "down".
 // "BlockSize * bx" is the number of elements to move "right".
 const unsigned int b_idx = BlockSize * (b_cols * step + bx);

 // Load each element in the tile to shared memory.
 a_values[ty][tx] = A[a_idx + a_cols * ty + tx];
 b_values[ty][tx] = B[b_idx + b_cols * ty + tx];

 // Synchronization is needed to make sure that all elements are
 // loaded before starting the calculation.
 __syncthreads();
```

```
 // Each thread calculates the scalar product of the tile and
 // increments the thread-individual thread_result.
 for(unsigned int i = 0; i < BlockSize; i++)
    {
       thread_result += a_values[ty][i] * b_values[i][tx];
    }

 // Synchronize to ensure that the calculation is finished before the
 // next tile's elements start to load.
        __syncthreads();
    }
```

This example illustrates how the *BlockSize* x *BlockSize* threads in a "block" coordinate to access adjacent memory, in a fashion called "coalesced access". Notice how the *BlockSize* parameter is compile-time determined, in that it is specified as a template parameter (check out the HIP chapter in this document). We also observe the context of this kernel seems to differ much from our OpenMP loops seen earlier. The following listing builds a bridge instead, in that it does minimal adaptation of the same exact kernel (without the comments) and puts it in a mock-up setting (notice the stubs for specific HIP keywords) where it can be used on a CPU, running with similar outcomes:

```
// Excerpt: kernel_2d_hip.cpp

// adaptation from
// https://github.com/amd/rocm-examples/tree/develop/HIP-Basic/\
// matrix_multiplication
#include <iostream>
#include <vector>
#include <omp.h>

using n_t = float;

const int BlockSize = 16;
struct dim3 { int x{}, y{}; };

# define __shared__ static // stub
# define __global__ // stub
void barrier() {
 #pragma omp barrier
}
# define __syncthreads() barrier()

template<unsigned int BlockSize>
__global__ void matrix_multiplication_kernel(
          const dim3 & gridDim, const dim3 & blockDim,
          const dim3 & blockIdx,
          const dim3 & threadIdx, const n_t*A, const n_t*B, n_t*C,
          const unsigned int a_cols)
{
    const unsigned int tx = threadIdx.x;
    const unsigned int ty = threadIdx.y;
    const unsigned int bx = blockIdx.x;
    const unsigned int by = blockIdx.y;

    // b_cols must match the number of output matrix columns.
    const unsigned int b_cols = blockDim.x * gridDim.x;
```

```
    // The number of tiles is determined by A's columns
    // (which is equal to B's rows).
    const unsigned int steps = a_cols / BlockSize;

    // thread_result is the accumulation variable.
    float thread_result = 0.0F;
    for(unsigned int step = 0; step < steps; step++)
    {
    // Shared memory is used to cache the tile from both input matrices.
    // The tile is a square of BlockSize*BlockSize.
        __shared__ float a_values[BlockSize][BlockSize];
        __shared__ float b_values[BlockSize][BlockSize];

    // Index of the top-left element of the tile in A.
    // "BlockSize * a_cols * by" is the number of elements to move "down".
    // "BlockSize * step" is the number of elements to move "right".
        const unsigned int a_idx = BlockSize * (a_cols * by + step);

    // Index of the top-left element of the tile in B.
    // "BlockSize * b_cols * step" is the number of elements to move "down".
    // "BlockSize * bx" is the number of elements to move "right".
        const unsigned int b_idx = BlockSize * (b_cols * step + bx);

    // Load each element in the tile to shared memory.
        a_values[ty][tx] = A[a_idx + a_cols * ty + tx];
        b_values[ty][tx] = B[b_idx + b_cols * ty + tx];

    // Synchronization is needed to make sure that all elements are
    // loaded before starting the calculation.
        __syncthreads();

    // Each thread calculates the scalar product of the tile and
    // increments the thread-individual thread_result.
        for(unsigned int i = 0; i < BlockSize; i++)
        {
            thread_result += a_values[ty][i] * b_values[i][tx];
        }

    // Synchronize to ensure that the calculation is finished before
    // the next tile's elements start to load.
        __syncthreads();
    }

    // Calculate the index of the top-left element of the output block.
    const unsigned block_offset = b_cols * BlockSize * by + BlockSize * bx;

    // Every thread stores the final result to global memory.
    C[block_offset + b_cols * ty + tx] = thread_result;
}

template<typename F, typename... Ts>
void launch2D(const dim3 & numBlocks, const dim3 & blockDim, F & f,
              Ts&&... ts)
{
 for (int bx=0;bx<numBlocks.x;++bx)
 for (int by=0;by<numBlocks.y;++by)
 {
  #pragma omp parallel num_threads(blockDim.x*blockDim.y)
```

```
  {
   const int tn = omp_get_thread_num();
   const int tx = tn % blockDim.y;
   const int ty = tn / blockDim.y;
   f(numBlocks, blockDim, {bx,by}, {tx,ty}, ts...);
  }
 }
}

int main()
{
  const int m{7}; // arbitrary
  const int N{BlockSize*m};
  const std::vector<n_t> a(N*N,2.0);
  const std::vector<n_t> b(N*N,2.0);
  std::vector<n_t> c(N*N,0);
  const dim3 threadsperBlock {BlockSize,BlockSize};
  const dim3 numBlocks{N/threadsperBlock.x,N/threadsperBlock.y};
  launch2D(numBlocks, threadsperBlock,
           matrix_multiplication_kernel<BlockSize>,
    a.data(), b.data(), c.data(), N);
 ...
}
```

It is important to mention that on the CPU, inner loop synchronization is inherently to be avoided, for performance reasons. In GPU programming instead, synchronization at the most nested level is fine -- the threads are executed by very close units which are being scheduled in block. So contrast this with *MatMatMul_GPU___data_3* and *MatMatMul_GPU___data_4*, where there is no inner loop synchronization. This example therefore departs significantly from the previous ones, but is still useful on the CPU to prototype and verify GPU-oriented kernels with coalescing. Adapting this kernel to run in the same fashion as the other kernels, one gets:

```
// Excerpt: MatMatMul_GPU___data_5

#pragma omp target enter data map(to:a[:N*N],b[:N*N])
#pragma omp target enter data map(to:c[:N*N])
...
#pragma omp target teams distribute collapse(2)
for (int blockIdx_x =0; blockIdx_x < gridDim_x; ++blockIdx_x )
for (int blockIdx_y =0; blockIdx_y < gridDim_y; ++blockIdx_y )
{
 const int bx = blockIdx_x;
 const int by = blockIdx_y;
 const int te = omp_get_team_num();

 n_t a_values[BlockSize][BlockSize];
 n_t b_values[BlockSize][BlockSize];
 #pragma omp parallel
 {
  const int a_cols = N;
  const int tn = omp_get_thread_num();
  const int tx = tn % blockDim_y;
  const int ty = tn / blockDim_y;

  const int b_cols = blockDim_x * gridDim_x;
  const int steps = a_cols / BlockSize;
  n_t thread_result = 0.0;
```

```
  for(int step = 0; step < steps; step++)
  {
   const int a_idx = BlockSize * (a_cols * by + step);
   const int b_idx = BlockSize * (b_cols * step + bx);

   a_values[ty][tx] = a[a_idx + a_cols * ty + tx];
   b_values[ty][tx] = b[b_idx + b_cols * ty + tx];
   #pragma omp barrier
   for(int i = 0; i < BlockSize; i++)
   {
    thread_result += a_values[ty][i] * b_values[i][tx];
   }
   #pragma omp barrier
  }
  const unsigned block_offset = b_cols * BlockSize * by + BlockSize * bx;
  c[block_offset + b_cols * ty + tx] += alpha * thread_result;
 }
}
...
#pragma omp target exit data map(from:c[:N*N])
```

This last kernel, if executed on the GPU, is the best of the series so far. It is the last kernel of this tutorial. Further optimization rounds would be to introduce register blocking at the deepest nesting level. That is, the *a_values* and *b_values* arrays would be expanded by a certain factor, and the loops afterwards would act on small contiguous elements at a time. We have seen this in the examples *MatMatMul_GPU___data_3* and *MatMatMul_GPU___data_4*. Kernels with all these optimizations occur in the "MAGMA BLAS" kernel and "rocBLAS"; see respectively: *gemm_template_device_nn* in [75] and *gemm_batched_general_kernel* in [76].

In those state-of-the-art GPU kernels, "optimal" register blocking can involve odd dimensions, and of course, because of their generality, remainder loops are present to handle the indices which fall off-modulo. For a state-of-the-art reference on CPUs, please see Prof. Robert van de Geijn's [77].

Full sources for this tutorial are available in [78].

In developing these examples with the CC (Cray flavour of the Clang compiler), it has been useful to use the CRAY_ACC_DEBUG variable. Another important help has been scrupulous testing by means of using results check on each kernel version.

4. Performance Analysis

4.1. NAMD

4.1.1. AMD CPU support

The CPU version of NAMD was built on LUMI machine. Here, EasyBuild was used for installing software in the local user space. For building NAMD v2.14, the following sequence of commands was employed:

```
ml LUMI/21.08  EasyBuild-user partition/C
eb -r NAMD-2.14-cpeGNU-21.08-MPI.eb
```

The performance of NAMD was studied by using the satellite tobacco mosaic virus (STMV) benchmark for 1,066,628 (1 M) and 223,991,880 (224 M) atoms which was obtained from the NAMD website [52]. The performance was measured by the number of nanoseconds per day (ns/day) with reference to the number of nodes used, results can be seen in Figure 14. In the case of the 224 M system, the memory-optimized version of NAMD was used that is more efficient for large models. It can be obtained by adding the option *--with-memopt* to the line starting with *./config* in the EasyBuild file for NAMD mentioned above. The compressed model can be used by adding the following lines to the NAMD input script:

```
useCompressedPsf on
structure          210stmv.js.inter
bincoordinates     210stmv.coor
```

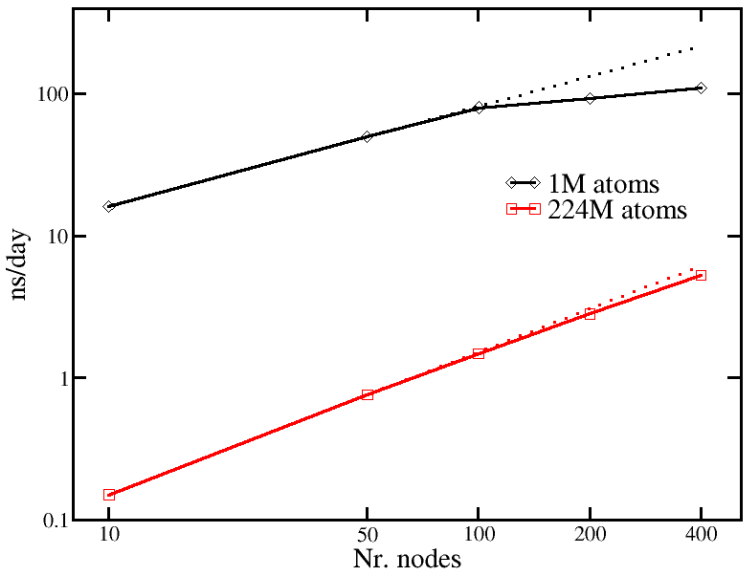

Figure 14. Performance of NAMD using the STMV benchmark case with reference to number of nodes for 1 M and 224 M atoms systems. Solid lines indicate the timing of the simulations while dashed lines the upper-bound power-law behavior.

A typical script for running the MPI version of NAMD on 400 nodes, built in the present study, looks as follows:

```
#!/bin/bash -l
#SBATCH --job-name=test-job          #job name
```

```
#SBATCH --account=project_X          #your project
#SBATCH --time=01:00:00              #wall-time
#SBATCH --nodes=400                  #nr. nodes
#SBATCH --ntasks=51200               #nr. MPI tasks
#SBATCH --ntasks-per-node=128        #nr. MPI tasks/node
#SBATCH --cpus-per-task=1            #nr. CPU/task
#SBATCH --partition=standard         #partition for < 500 nodes
##SBATCH --partition=bench           #partition for > 500 nodes

ml LUMI/21.08                        #module for LUMI environment
ml EasyBuild-user                    #module for EasyBuild
ml partition/C                       #module for partition
ml NAMD-MEM/2.14-cpeGNU-21.08-MPI    #modified NAMD memory-optimized version

srun namd2 210stmv2fs.namd           #running the case with NAMD MPI
```

4.1.2. AMD GPU support

NAMD Scalable Molecular Dynamics software is a highly scalable code [49] that supports different levels of parallelism including shared memory (multicore), MPI, and hybrids of them. NAMD also supports GPU acceleration with CUDA and recently a HIP implementation for AMD ROCm architectures was also added.

Although the NAMD source code for the HIP implementation can be obtained from the NAMD site [50], we recommend to obtain it from the GitLab repository [51] which is more frequently updated.

NAMD was compiled with support for AMD Instinct MI100 GPU by using the multicore implementation. The computing power was obtained from the Paralleldatorcentrum (PDC) at the Kungliga Tekniska Högskolan (KTH), Sweden. The building process of NAMD's dependencies (charmm++, tcl, and fftw) is described in the *notes.txt* file in the root directory of the repository. The rocm/4.5.0 module was used for building NAMD. The configuration of NAMD was set with:

```
./config Linux-x86_64-g++.hip --with-hip --charm-arch multicore-linux-x86_64
cd Linux-x86_64-g++.hip/
make
```

A test case of a protein in water (158,944 atoms) was used to monitor the performance of NAMD. This test case can be obtained from the HPC2N GitHub account [53]. In Table 13 we compare the performance of NAMD, measured in nanoseconds per day (ns/day), on different types of GPU cards on a single node: K80, V100, and MI100. A node with K80 or V100 cards contains 28 CPU cores only while the one with MI100 card contains up to 128 CPU cores.

Table 13. NAMD performance

Nr. cores	1xMI100 (ns/day)	2xV100 (ns/day)	2xK80 (ns/day)
28	12.7	18.5	7.8
56	13.9	-	-
128	14.4	-	-

The multiple time step algorithm (MTS) implemented in NAMD is one possibility to increase the performance of the simulation. If the non-bonded interactions (short and long range) are computed every 2 steps instead of every step as in the test case reported in the table, one achieves 17.2 ns/day by using 28 processes. Energy is well-conserved by using the MTS algorithm as it can be seen in Figure 15.

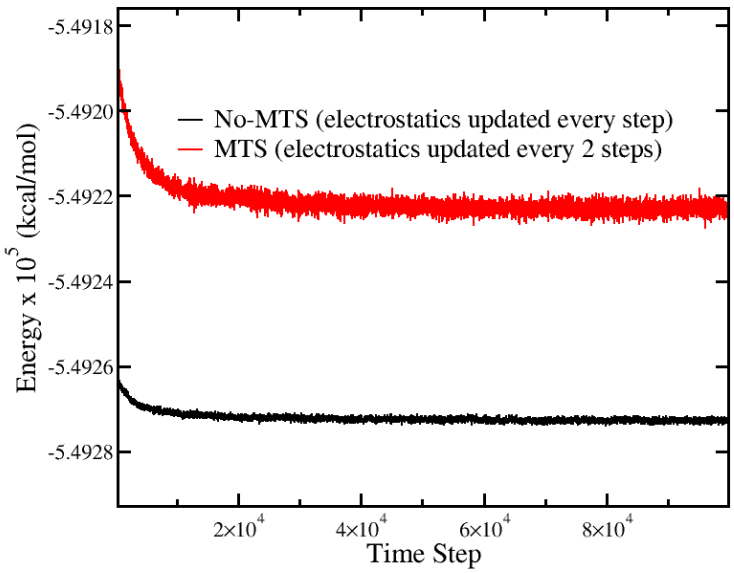

Figure 15. Energy conservation plot for the standard electrostatics computation (No-MTS) where these interactions are updated every time step and for the MTS algorithm where the interactions are updated every 2 time steps .

GPU usage during the simulation can be monitored with the rocm-smi tool:

```
$rocm-smi
======================= ROCm System Management Interface ================
================================== Concise Info ===========================
GPU  Temp   AvgPwr  SCLK     MCLK     Fan  Perf  PwrCap  VRAM%  GPU%
0    63.0c  79.0W   1502Mhz  1200Mhz  0%   auto  290.0W    1%   94%
============================================================================
============================== End of ROCm SMI Log =======================
```

NAMD can be profiled with the rocprof tool by running:

```
rocprof --hip-trace --roctx-trace --stats ./namd2 +p28\
step4_equilibration.inp
```

The output of the profiling analysis will be in the form of *.csv* and *.json* files. These are text files that are not easy to read with the naked eye. Here, we used the tracing tool available in the Chrome browser (*chrome://tracing/*) to visualize the resulting *.json* file. A typical profile looks like the one in Figure 16.

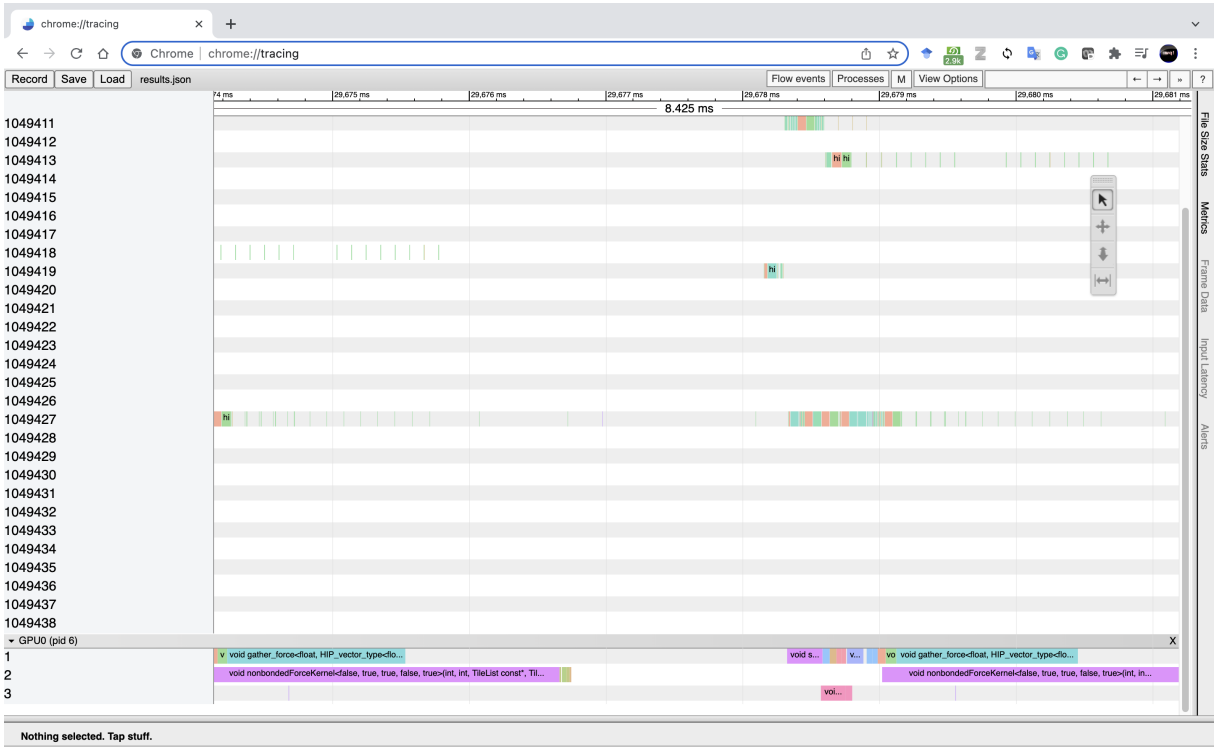

Figure 16. Typical profiling analysis obtained with rocprof tool and visualised with Chrome tracing feature.

4.2. Profiling with rocprof

4.2.1. Statistics

We assume that you have a HIP binary called saxpy that prints the max error. Extract statistics for the kernels and find out the kernel names if not sure.

```
rocprof --stats ./saxpy

RPL: on '210406_143600' from '/opt/rocm-4.1.0/rocprofiler' in
   '/.../hip/porting/codes/saxpy/hip_solution'
RPL: profiling '"./saxpy"'
RPL: input file ''
RPL: output dir '/tmp/rpl_data_210406_143600_2926946'
RPL: result dir '/tmp/rpl_data_210406_143600_2926946/\
input_results_210406_143600'
ROCProfiler: input from "/tmp/rpl_data_210406_143600_2926946/\
input.xml"
  0 metrics
  0 traces
Max error: 0.000000
ROCPRofiler: 1 contexts collected, output directory
/tmp/rpl_data_210406_143600_2926946/input_results_210406_143600
File '/path/hip/porting/codes/saxpy/hip_solution/results.csv' \
is generating
File '/path/hip/porting/codes/saxpy/hip_solution/results.stats.csv' \
is generating
```

```
cat results.stats.csv

"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"saxpy(int, float, float*, float*) [clone .kd]",1,14651626,14651626,100.0
```

We can see the name of the kernel "saxpy(int, float, float*, float*) [clone .kd]" the number if the times that the kernel was called, the total duration time in nanoseconds, the average duration in nanoseconds, and the percentage of the measured time.

4.2.1.1. Metrics

1) Create a file for example metrics.txt with the following content (you can declare whatever metrics you want):

```
pmc: GPUBusy Wavefronts VALUInsts SALUInsts SFetchInsts MemUnitStalled
VALUUtilization VALUBusy SALUBusy L2CacheHit WriteUnitStalled
LDSBankConflict
range: 0:1
gpu: 0
kernel: saxpy
```

The pmc line is the performance counters and you can decide which ones you want, as also read their meaning from XML files in «/opt/rocm/rocprofiler/lib/»[6]. The line range is about the number of the kernels, here we have just one. The GPU declares which GPU is used. The kernel tp be used is the names kernel, to write more than one leave a space between them. Seems that it works if you write the name of the kernel without all the extra options.

2) Execute:

```
rocprof -i metrics.txt ./saxpy

RPL: on '210406_145228' from '/opt/rocm-4.1.0/rocprofiler' in
  '/path/hip/porting/codes/saxpy/hip_solution'
RPL: profiling '"./sacpy"'
RPL: input file 'metrics.txt'
RPL: output dir '/tmp/rpl_data_210406_145228_2929575'
RPL: result dir '/tmp/rpl_data_210406_145228_2929575/\
input0_results_210406_145228'
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.csv'\
 is generating
[markoman@o186i142 hip_solution]$ rocprof -i metrics.txt ./saxpy
RPL: on '210406_145230' from '/opt/rocm-4.1.0/rocprofiler'
   in '/path/hip/porting/codes/saxpy/hip_solution'
RPL: profiling '"./saxpy"'
RPL: input file 'metrics.txt'
RPL: output dir '/tmp/rpl_data_210406_145230_2929751'
RPL: result dir '/tmp/rpl_data_210406_145230_2929751/\
input0_results_210406_145230'
ROCProfiler: input from "/tmp/rpl_data_210406_145230_2929751/input0.xml"
  gpu_index = 0
  kernel = saxpy
  range = 0:1
  12 metrics
    GPUBusy, Wavefronts, VALUInsts, SALUInsts, SFetchInsts,
MemUnitStalled, VALUUtilization, VALUBusy, SALUBusy, L2CacheHit,
```

[6]This directory could be different when modules related to the rocprofiler are loaded.

```
WriteUnitStalled, LDSBankConflict
  0 traces
Max error: 0.000000
ROCPRofiler: 1 contexts collected, output directory
/tmp/rpl_data_210406_145230_2929751/input0_results_210406_145230
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.csv'\
 is generating
```

3) Read the metrics.csv

```
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,
vgpr,sgpr,fbar,sig,obj,GPUBusy,Wavefronts,VALUInsts,SALUInsts,SFetchInsts,
MemUnitStalled,VALUUtilization,VALUBusy,SALUBusy,L2CacheHit,
WriteUnitStalled,LDSBankConflict
0,"saxpy(int, float, float*, float*) [clone .kd]",0,0,0,2929907,2929934,
1073741824,256,0,0,4,24,0,0x0,0x7ff9e74097c0,100,16777216,11,3,3,14,
100,7,1,33,0,0
```

The *GPUBusy* shows percentage of the utilisation of the GPU from this kernel, the *Wavefronts* is the number of the wavefronts, the *MemUnitStalled* the percentage GPU time that memory unit is stalled etc. The explanations are in the above mentioned XML files.

4.2.2. Traces

```
rocprof --sys-trace ./saxpy

RPL: on '210406_150505' from '/opt/rocm-4.1.0/rocprofiler'
  in '/path/hip/porting/codes/saxpy/hip_solution'
RPL: profiling '"./saxpy"'
RPL: output dir '/tmp/rpl_data_210406_150505_2932085'
RPL: result dir '/tmp/rpl_data_210406_150505_2932085/\
input0_results_210406_150505'
ROCTracer (pid=2932241):
    HSA-trace()
    HIP-trace()
Max error: 0.000000
START timestamp found (487661870180338ns)
scan ops data 3:4
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.hsa_stats.csv'\
 is generating
dump json 3672:3673
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.json'\
 is generating
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.copy_stats.csv'\
 is generating
dump json 2:3
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.json'\
 is generating
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.hip_stats.csv'\
 is generating
dump json 9:10
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.json'\
 is generating
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.stats.csv'\
 is generating
```

```
dump json 0:1
File '/path/hip/porting/codes/saxpy/hip_solution/metrics.json'\
 is generating
```

There are a few files called `results*` created for copy data, HIP API, HSA etc, you can open and see the duration of many events.

```
results.copy_stats.csv
results.db
results.hip_stats.csv
results.hsa_stats.csv
results.json
results.stats.csv
results.sysinfo.txt
```

You can visualize the json file with Chrome browser:

- Download the json file to your laptop
- Open Chrome browser and go to the web page: chrome://tracing/
- Click the load button and select the result.json file

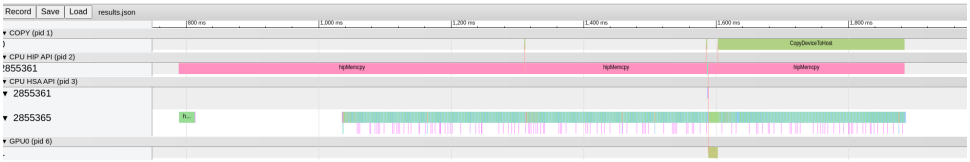

Figure 17. Load rocprof trace with Chrome

5. Running applications in Containers

This section discusses running of applications with focus on containers for scientific applications.

5.1. Introduction

LUMI supports Singularity/Apptainer [79] containers as an alternative way to bring your scientific application to LUMI instead of installing it using EasyBuild [80] or https://docs.lumi-supercomputer.eu/software/installing/spack Spack . If you are familiar with Docker [81] containers, Singularity/Apptainer containers are essentially the same thing, but are better suited for multi-user HPC systems such as LUMI. The main benefit of using a container is that it provides an isolated software environment for each application, which makes it easier to install complex applications.

Building containers on LUMI is, however, not possible. The singularity build command requires some level of root privileges, e.g. sudo or fakeroot, which are disabled on LUMI for security reasons. Thus, in order to prepare a Singularity/Apptainer container for LUMI, you have two options: Pull an existing container image (Singularity or Docker) from a registry or build your own container on your local hardware, e.g. your laptop.

Singularity allows pulling images (Singularity or Docker) from container registries such as DockerHub [82] or AMD InfinityHub [83]. Pulling container images from registries can be done on LUMI. For instance, the Ubuntu image ubuntu:22.04 can be pulled from DockerHub with the following command:

```
$ singularity pull docker://ubuntu:22.04
```

This will create the Singularity image file ubuntu_22.04.sif in the directory where the command was run. Once the image has been pulled, the container can be run. Instructions for running the container may be found at the container jobs page [84].

5.2. Building LUMI MPI compatible containers

Here we provide an example of building a container that is compatible with the MPI stack on LUMI. For MPI-enabled containers, the application inside the container must be dynamically linked to an MPI version that is ABI-compatible [85] with the host MPI. The following Singularity definition file mpi_osu.def, installs MPICH-3.1.4, which is ABI-compatible with the Cray-MPICH found on LUMI. That MPICH will be used to compile the microbenchmarks[86]. Finally, the OSU point to point bandwidth test is set as the runscript of the image.

```
from: ubuntu:21.04
%post
   # Install software
   apt-get update
   apt-get install -y file g++ gcc gfortran make gdb strace \
wget ca-certificates --no-install-recommends

   # Install mpich
   wget -q http://www.mpich.org/static/downloads/3.1.4/\
mpich-3.1.4.tar.gz
   tar xf mpich-3.1.4.tar.gz
   cd mpich-3.1.4
   ./configure --disable-fortran --enable-fast=all,O3 --prefix=/usr
   make -j$(nproc)
   make install
   ldconfig
```

```
   # Build osu benchmarks
   wget -q http://mvapich.cse.ohio-state.edu/download/\
mvapich/osu-micro-benchmarks-5.3.2.tar.gz
   tar xf osu-micro-benchmarks-5.3.2.tar.gz
   cd osu-micro-benchmarks-5.3.2
   ./configure --prefix=/usr/local CC=$(which mpicc) CFLAGS=-O3
   make
   make install
   cd ..
   rm -rf osu-micro-benchmarks-5.3.2
   rm osu-micro-benchmarks-5.3.2.tar.gz

%runscript
   /usr/local/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_bw
```

The image can be built on your local hardware (not LUMI) with

```
$ sudo singularity build mpi_osu.sif mpi_osu.def
```

The mpi_osu.sif file must then be transferred to LUMI [https://docs.lumi-supercomputer.eu/firststeps/movingdata/]. See the container jobs MPI documentation page for instructions on running this MPI container on LUMI.

5.3. Container jobs

LUMI provides access to a singularity runtime for running applications in software containers. Currently, there are two major providers of the singularity runtime, namely Singularity CE [https://docs.sylabs.io/guides/latest/user-guide/] and Apptainer [http://apptainer.org/docs/user/main/index.html], with the latter being a fork of the former. For most cases these should be fully compatible. LUMI provides the singularity runtime included in the HPE Cray Operating System (a SUSE Linux based OS) running on LUMI - no modules need to be loaded to use singularity on LUMI. You can always check the version of singularity using the command

```
singularity --version.
```

See the Apptainer/Singularity containers install page [https://docs.lumi-supercomputer.eu/software/containers/singularity/] for details about creating LUMI compatible software containers.

5.3.1. The basics of running a container on LUMI

Applications in a container may be run by combining Slurm commands with Singularity commands, e.g. to get the version of Ubuntu running in a container stored as "ubuntu_22.04.sif", we may use srun to execute the singularity container

```
$ srun --partition=<partition> --account=<account> \
singularity exec ubuntu_21.04.sif cat /etc/os-release
```

which prints something along the lines of:

```
PRETTY_NAME="Ubuntu 22.04.1 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.1 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
```

```
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/\
privacy-policy"
UBUNTU_CODENAME=jammy
```

Binding network file systems in the container can be done as follows. By default, the network file system partitions, such as /scratch or /project are not accessible from the within the container. To make them available, they need to be explicitly bound by passing the -B/--bind command line option to singularity exec/run. For instance

```
$ srun --partition=<partition> --account=<project_name>\
 singularity exec -B /scratch/>project_name< ubuntu_21.04.sif \
ls /scratch/<account>
```

Warning: Since project folder paths like /scratch/<project_name> and /project/<project_name> are symlinks on LUMI, you must bind these full paths to make them available in the container. Simply binding /scratch or /project will not work.

5.3.2. Running containerized MPI applications

Running MPI applications in a container requires that you either bind the host MPI (the MPI stack provided as part of the software stack available on the compute node) or install a LUMI compatible MPI stack in the container.

Warning: For MPI-enabled containers, the application inside the container must be dynamically linked to an MPI version that is ABI-compatible with the host MPI.

The two approaches for MPI Containers in short (details follow later):

5.3.2.1. Fully containerised, Optimised MPI installation

- Replicating the network stack:
  - Check what kind of operating system the target is running
  - Check what kind of network stack the target system has
  - Type of interconnect, software distribution type and version
  - Check for any additional kernel modules enabled for shared memory transport (e.g CMA,knem,xpmem)
- During container construction:
  - Install all the network stack components
  - Install any other required libraries
  - Build the MPI library against all the installed components
  - It might be required to first build some lower level communication libraries, or there might be a pre-built package available from the same source as the network components.
  - Compile program with installed MPI wrappers

5.3.2.2. Hybrid MPI installation

- Install MPI inside the container

- Bind mount the host MPI (and all needed dynamic dependencies)
- Start the program
- using either srun outside the container
- Using this approach we will also be compatible with the startup environment (assuming that we mounted any sockets if needed.)

5.3.2.3. Using the host MPI

In order to properly make use of LUMI's high-speed network, it is necessary to mount a few host system directories inside the container and set LD_LIBRARY_PATH so that the necessary dynamic libraries are available at run time. This way, the MPI installed in the container image is replaced by the host's MPI stack. All the necessary components are available in a module that can be installed by the user via EasyBuild $ module load LUMI partition/<lumi-partition> EasyBuild-user

```
$ srun --partition=<partition>; --account=eb\
 singularity-bindings-system-cpeGNU-<toolchain-version>.eb -r
```

Running e.g. the OSU point-to-point bandwidth test container [87] can then be done using

```
$ module load singularity-bindings
$ srun --partition=<partition> --account=<account> --nodes=2\
 singularity run mpi_osu.sif
```

which gives the bandwidth measured for different message sizes, i.e. something along the lines of

```
# OSU MPI Bandwidth Test v5.3.2
# Size      Bandwidth (MB/s)
1                       3.00
2                       6.01
4                      12.26
8                      24.53
16                     49.83
32                     97.97
64                    192.37
128                   379.80
256                   716.64
512                  1386.52
1024                 2615.18
2048                 4605.69
4096                 6897.21
8192                 9447.54
16384               10694.19
32768               11419.39
65536               11802.31
131072              11997.96
262144              12100.20
524288              12162.28
1048576             12207.27
2097152             12230.66
4194304             12242.46
```

5.3.2.4. Using the container MPI

MPI applications can also be run using an MPI stack installed in the container. To do so, Slurm needs to be instructed to use the PMI-2 process management interface by passing --mpi=pmi2 to srun, e.g.

```
$ srun --partition=<partition> --account=<account> --mpi=pmi2 --nodes=2\
 singularity run mpi_osu.sif
```

which produces an output along the lines of

```
# OSU MPI Bandwidth Test v5.3.2
# Size      Bandwidth (MB/s)
1                       0.50
2                       1.61
4                       3.57
8                       6.54
16                      9.65
32                     18.04
64                     35.27
128                    67.76
256                    91.12
512                   221.09
1024                  278.88
2048                  471.54
4096                  917.02
8192                 1160.74
16384                1223.41
32768                1397.97
65536                1452.23
131072               2373.07
262144               2104.56
524288               2316.71
1048576              2478.30
2097152              2481.68
4194304              2380.51
```

Note that this approach gives lower bandwidths, especially for the larger message sizes, than is the case when using the host MPI. In general, the performance obtained from using the container MPI might be quite low compared to the results obtained when using the host's MPI. For a more in-depth discussion of the topic of MPI in containers, we suggest that introduction to MPI in containers see [88].

Figure 18 below illustrates the difference in bandwidth, which is significant: almost a factor of ten (the scale is logarithmic which hides the large difference somewhat). From this data the natural conclusion is that the best practice is to use the host MPI.

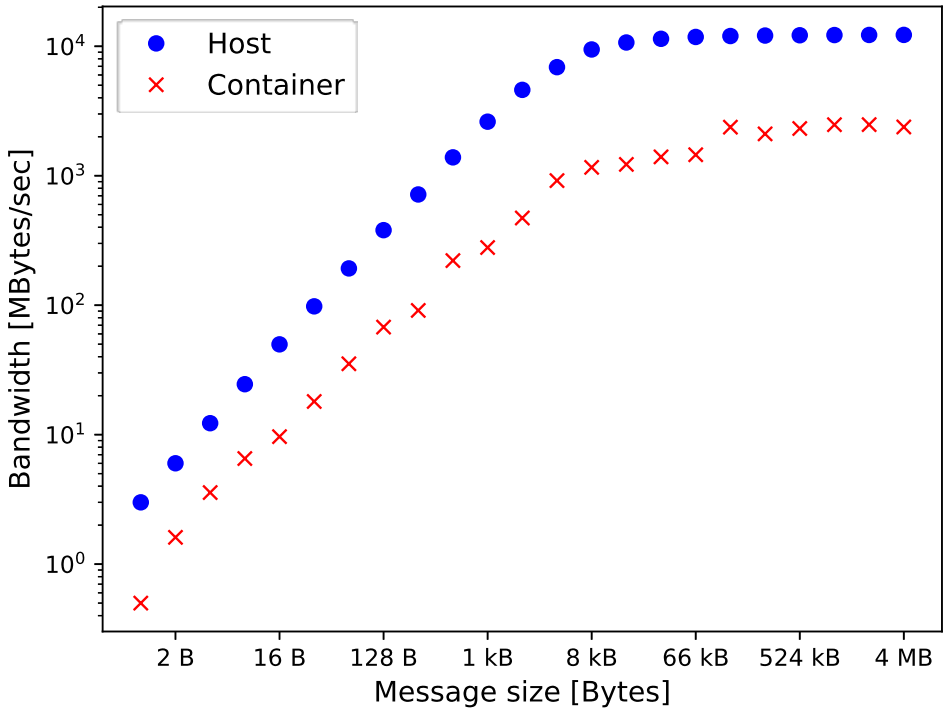

Figure 18. Container MPI bandwidth, Host based MPI versus Container based MPI (note logarithmic Bandwidth scale)

A. Acronyms and Abbreviations

1. Units

n	S.I. Unit nano = 10^{-9}
µ	S.I. Unit micro = 10^{-6}
k	S.I. Unit kilo = 10^{3}
M	S.I. Unit Mega = 10^{6}
G	S.I. Unit Giga = 10^{9}
T	S.I. Unit Tera = 10^{12}
P	S.I. Unit Peta = 10^{15}
E	S.I. Unit Exa = 10^{18}
ki	ISO/IEC 8000 Unit kibi = 2^{10}
Mi	ISO/IEC 8000 Unit Mebi = 2^{20}
Gi	ISO/IEC 8000 Unit Gibi = 2^{30}
Ti	ISO/IEC 8000 Unit Tibi = 2^{40}
Byte, B	8 bits, (an octet)
kiB	2^{10} Bytes
MiB	2^{20} Bytes
GiB	2^{30} Bytes
TiB	2^{40} Bytes
GHz	Giga Hertz, Hertz is frequency, events per second.
MHz	Mega Hertz, Hertz is frequency, events per second.
nm	Nano meter, 10^{-9} meter.
MT	Mega Transactions, millions of transactions per second, often used for memory transactions and network (InfiniBand) transactions
W	Watt, units of energy per second, Joules/second

A note on usage of units involving Bytes,

- Addressable memory is measured in units of power of two, e.g. kiB.
- Storage is addressed in units of power of ten, kB (storage industry use this unit).
- Bandwidth is measured in units power of ten, kB/second (network and telecom industry use this unit).

2. Terms

double precision	Double precision is used as a term for 64 bit floating-point data type, approximately 16 decimal precision.

single precision	Single precision is used as a term for 32 bit floating-point data type, approximately 8 decimal precision.
BLAS prefix	The standard netlib BLAS library [89] uses a single letter prefix to specify numerical precision : • s : Single precision • d : Double precision • c : Single precision complex • z : Double precision complex

3. Acronyms

ALU	Arithmetic Logical Units, one of the executional units within the processor (CPU)
AVX	Advanced Vector Instructions, a set of extra vector (SIMD) instructions added to the SSE vector instruction set. There is also a an extra addition, AVX2
CPU	Central Processing Unit, refer to a single core or the complete package to fit in a socket, ambiguous
GPU	Graphical Processing Unit, refer to an accelerator used to speed up certain operations.
DIMM	Dual In line Memory Module, variants RDIMM (registered ECC), LRDIMM (Load Reduced DIMM)
HBM2	High Bandwidth Memory, second generation.
NVMe	Non-Volatile Memory, a type of solid state flash memory.
BPG	Best Practice Guide

Many other acronyms occur with references in the bibliography.

B. Acknowledgments

- This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767
- This research was partially conducted using the resources of High-Performance Computing Center North (HPC2N) and the Paralleldatorcentrum (PDC) at the Swedish National Infrastructure for Computing (SNIC).
- This work have been supported by Sigma2, access to resources in Norway and as LUMI partner also provided LUMI access.

Further documentation

[1] IT Center for Science https://www.csc.fi/ .

[2] LUMI supercomputer https://www.lumi-supercomputer.eu/ .

[3] The top500 list https://top500.org/ .

[4] The LUMI consortium [https://lumi-supercomputer.eu/lumi-consortium/] .

[5] The LUMI documentation [https://docs.lumi-supercomputer.eu] .

[6] The LUMI introduction [https://www.lumi-supercomputer.eu/may-we-introduce-lumi] .

[7] Get started with LUMI [https://www.lumi-supercomputer.eu/get-started] .

[8] AMD EPYC 7763. [https://www.amd.com/en/products/cpu/amd-epyc-7763].

[9] AMD CDNA 2 architecture [https://www.amd.com/en/technologies/cdna2].

[10] Specification of AMD MI250X GPUs [https://www.amd.com/en/products/server-accelerators/instinct-mi250].

[11] AMD CDNA 2 architecture white paper [https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf].

[12] AMD MI250X bandwidth measurements [https://x-dev.pages.jsc.fz-juelich.de/2022/08/01/mi250-first-performances.html].

[13] IOZone storage benchmark [https://www.iozone.org/].

[14] Streaming processing [https://en.wikipedia.org/wiki/Stream_processing].

[15] Terminology from Weaving [https://www.anandtech.com/show/2549/4].

[16] Selecting-applications-suitable-for-porting-to-the-gpu [https://bssw.io/blog_posts/porting-codes-to-new-architecture].

[17] Better Scientific Software [https://bssw.io/blog_posts/porting-codes-to-new-architectures].

[18] Infinity fabric [https://en.wikichip.org/wiki/amd/infinity_fabric].

[19] Infinity fabric, LUMI documentation [https://docs.lumi-supercomputer.eu/hardware/compute/lumig/].

[20] Fastest Fourier Transfer in the West A popular open source implementation of Fourier Transforms [http://www.fftw.org/].

[21] Best Practice Guide : Modern-Processors-Accelerators Best Practice Guide : Modern-Processors-Accelerators [https://prace-ri.eu/training-support/best-practice-guides/modern-processors/].

[22] Best-Practice-Guide-Modern-Accelerators [https://prace-ri.eu/training-support/best-practice-guides/modern-processors/].

[23] Best Practice Guide : Modern-Processors-Accelerators Best Practice Guide : Modern-Processors-Accelerators [https://prace-ri.eu/training-support/best-practice-guides/modern-processors/].

[24] Fugaku supercomputer [https://en.wikipedia.org/wiki/Fugaku_(supercomputer)].

[25] Unified memory [https://www.anandtech.com/show/15593/amd-unveils-cdna-gpu-architecture-a-dedicated-gpu-architecture-for-data-centers].

[26] MI250X hardware [https://docs.amd.com/en-US/bundle/AMD-Instinct-MI250-High-Performance-Computing-and-Tuning-Guide-v5.3/page/AMD_Instinct_Hardware.html].

[27] GPU SIMD, blog article [https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world/].

[28] ROCm documentation [https://rocmdocs.amd.com/en/latest/ROCm_Libraries/ROCm_Libraries.html].

[29] Introduction to High Performance Computing for Scientists and Engineers, Georg Hager and Gerhard Wellein CRC Press 2011, with ISBN 978-1-4398-1192-4 .

[30] Julia language [https://julialang.org/].

[31] Julia GPU [https://juliagpu.org].

[32] Julia downloads [https://julialang.org/downloads/].

[33] AMDGPU API reference [https://amdgpu.juliagpu.org/stable/api/].

[34] OpenCL [https://www.khronos.org/opencl/].

[35] SYCL [https://www.khronos.org/sycl/].

[36] LLVM OpenACC development [https://openmp.llvm.org/openacc/Overview.html].

[37] OpenCL specification [https://www.khronos.org/blog/opencl-3.0-specification-finalized-and-initial-khronos-open-source-opencl-sdk-released].

[38] OpenCL documentation [https://www.khronos.org/opencl/assets/CXX_for_OpenCL.html#opencl-spec].

[39] Focal GitHub [https://github.com/LKedward/focal].

[40] CLFortran GitHub [https://github.com/cass-support/clfortran].

[41] Focal documentation [https://lkedward.github.io/focal-docs/quickstart/].

[42] Wikipedia SMT, https://en.wikipedia.org/wiki/Simultaneous_multithreading [https://en.wikipedia.org/wiki/Simultaneous_multithreading].

[43] Wikipedia SMT, https://en.wikipedia.org/wiki/Hyper-threading [https://en.wikipedia.org/wiki/Hyper-threading].

[44] Bifrost Gudiksen, B. V., Carlsson, M., Hansteen, V. H., Hayek, W., Leenaarts, J., & Martinez-Sykora, J.: 2011, Astronomy and Astrophysics 531, A154, The stellar atmosphere simulation code Bifrost. Code description and validation [https://www.mn.uio.no/rocs/english/projects/solar-atmospheric-mod elling/].

[45] HWCART repository, 2021 [https://github.com/ghex-org/hwcart].

[46] GHEX repository, 2021 [https://github.com/ghex-org/].

[47] GHEXBENCH repository, 2021 [https://github.com/ghex-org/ghexbench].

[48] Broquedis, Francois, et.al, hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications (2010).

[49] J. C. Phillips, D. J. Hardy, J. D. C. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Henin, W. Jiang, R. McGreevy, M. C. R. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. Kale, K. Schulten, C. Chipot, and E. Tajkhorshid. Scalable molecular dynamics on CPU and GPU architectures with NAMD, Journal of Chemical Physics, 153, 044130, (2020).

[50] NAMD website [https://www.ks.uiuc.edu/Research/namd/].

[51] NAMD GitLab repository [https://gitlab.com/tcbgUIUC/namd/].

[52] NAMD benchmarks [http://www.ks.uiuc.edu/Research/namd/benchmarks/].

[53] HPC2N GitLab repository [https://github.com/hpc2n/CourseEfficientMD/tree/main/benchmark/NAMD/GPU].

[54] McCalpin J. STREAM benchmark https://www. cs.virginia.edu/stream/ref.html#what [https://www.cs.virginia.edu/stream/ref.html#what].

[55] Memory [https://www.akkadia.org/drepper/cpumemory.pdf].

[56] MPI documentation [https://www.mpi-forum.org/docs/m pi-4.0/mpi40-report.pdf].

[57] OpenACC [https://www.openacc.org].

[58] OpenACC API [https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf].

[59] OpenMP [https://www.openmp.org].

[60] OpenMP API [https://www.openmp.org/wp -content/uploads/OpenMP-API-Specification-5-2.pdf].

[61] OpenMPI library [https://www.open-mpi.org/].

[62] MPICH library [https://www.mpich.org/].

[63] CUDA framework [https://docs.nvidia.com/cuda/].

[64] Running GPU-aware Open MPI [https://www.open-mpi.org/faq/?category=runcuda].

[65] GPU-aware MPI_Allreduce [https://dl.acm.org/doi/10.1145/2642769.2642773].

[66] Github repository [https://github.com/HichamAgueny/MPI_ACC_OMP].

[67] GPU topology [https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html].

[68] LUMI documentation [https://docs.lumi-supercomputer.eu/development/compiling/cce/].

[69] NVIDIA SDK Using OpenMP [https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#openmp-use].

[70] OpenMP historic development [https://www.osti.gov/pages/servlets/purl/1465188].

[71] Diaz, Jose Monsalve, et al. "Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead." Parallel Computing 89 (2019):102546.

[72] OMP-Offloading [https://github.com/pc2/OMP-Offloading].

[73] N-Body simulation [https://github.com/themathgeek13/N-Body-Simulations-CUDA].

[74] HIP MxM kernel from AMD [https://github.com/amd/rocm-examples/blob/develop/HIP-Basic/matrix_multiplication/main.hip].

[75] MAGMA BLAS kernel [https://bitbucket.org/icl/magma/src/master/magmablas/gemm_template_device.cuh].

[76] rocBLAS kernel [https://github.com/ROCmSoftwarePlatform/rocBLAS/blob/develop/library/src/blas3/Tensile/gemm_source.hpp].

[77] How to optimise GEMM [https://github.com/flame/how-to-optimize-gemm].

[78] GEMM tutorial at LRZ [https://github.com/LRZ-BADW/OMMOP].

[79] Singularity/Apptainer documentation [https://docs.sylabs.io/guides/latest/user-guide/].

[80] EasyBuild documentation [https://docs.lumi-supercomputer.eu/software/installing/easybuild].

[81] Docker Wikipedia [https://en.wikipedia.org/wiki/Docker_(software)].

[82] DockerHub [https://hub.docker.com].

[83] Infinity Hub [https://www.amd.com/en/technologies/infinity-hub].

[84] LUMI container docker documentation [https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/container-jobs].

[85] MPICH ABI [https://www.mpich.org/abi/].

[86] Ohio State University MPI microbenchmarks [https://mvapich.cse.ohio-state.edu/benchmarks/OSU].

[87] Building LUMI MPI compatible container documentation [https://docs.lumi-supercomputer.eu/software/containers/singularity/#building-lumi-mpi-compatible-containers].

[88] MPI in container [https://permedcoe.github.io/mpi-in-container].

[89] BLAS (Basic Linear Algebra Subprograms) [https://netlib.org/blas/].