A multithreaded CUDA and OpenMP based power-aware programming framework for multi-node GPU systems

Paweł Czarnul*

Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233 Gdansk, Poland

SUMMARY

In the paper, we have proposed a framework that allows programming a parallel application for a multi-node system, with one or more GPUs per node, using an OpenMP+extended CUDA API. OpenMP is used for launching threads responsible for management of particular GPUs and extended CUDA calls allow to manage CUDA objects, data and launch kernels. The framework hides inter-node MPI communication from the programmer who can benefit from the traditional OpenMP+CUDA API in a multi-node environment. For optimization, the implementation takes advantage of the `MPI THREAD MULTIPLE` mode allowing: multiple threads handling distinct GPUs as well as overlapping communication and computations transparently using multiple CUDA streams. The solution allows data parallelization across available GPUs in order to minimize execution time and supports a power-aware mode in which GPUs are automatically selected for computations using a greedy approach in order not to exceed an imposed power limit. We have implemented and benchmarked three parallel applications including: finding the largest divisors; verification of the Collatz conjecture; finding patterns in vectors. These were tested on three various systems: a GPU cluster with 16 nodes, each with NVIDIA GTX 1060 GPU; a powerful 2-node system – one node with 8x NVIDIA Quadro RTX 6000 GPUs, the second with 4x NVIDIA Quadro RTX 5000 GPUs; a heterogeneous environment with one node with 2x NVIDIA RTX 2080 and 2 nodes with NVIDIA GTX 1060 GPUs. We demonstrated effectiveness of the framework through execution times versus power caps within ranges of 100-1400W, 250-3000W and 125-600W for these systems respectively as well as gains from using two versus one CUDA streams per GPU. Finally, we have shown that for the testbed applications the solution allows to obtain high speed-ups between 89.3% to 97.4% of the theoretically assessed ideal ones, for 16 nodes and 2 CUDA streams, demonstrating very good parallel efficiency.

Received . . .

KEY WORDS: GPU cluster; multi-node GPU system; parallelization; multithreaded programming framework; performance and power optimization; power cap

*Correspondence to: Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233 Gdansk, Poland, Tel.: +48-58-3471288, Fax: +48-58-3486125

Prepared using cpeauth.cls *[Version: 2010/05/13 v3.00]*

1. INTRODUCTION

Graphical Processing Units (GPUs) are contemporary computing devices very well suited for massively parallel execution of selected classes of code, ideally without thread divergence and processing data with coalesced access to global memory and allowing further optimizations using GPU shared memory etc. Further increase in processing power for such workloads comes from packing multiple GPUs within a node and integrating several nodes into a GPU cluster/multi-node system. Efficient and scalable programming such GPU systems requires proper APIs. Specifically, within a single CPU+GPU node, general purpose APIs typically used for GPUs include CUDA, OpenCL, OpenACC [1] and OpenMP [2]. For a GPU cluster, the aforementioned APIs are combined with MPI e.g. CUDA+MPI [1] or OpenMP+CUDA+MPI [3]. However, this is a real challenge for many programmers as they often use just a single or only selected APIs [4]. Consequently, using a simplified programming model for a GPU cluster, allowing good scaling across GPUs in several nodes, is a very desired feature. In this context, the goal of this paper is to propose a multithreaded programming model based on OpenMP, traditionally used within a single node, that allows to manage several GPUs, not only within a single node but a whole GPU cluster. Communication with the other nodes, when transferring data or launching CUDA kernels on the remote nodes, is completely hidden from the programmer behind the extended CUDA API.

Additionally, contemporary high performance computing aims at consideration of energy and power, next to performance [5]. Specifically, of interest are power-aware schedulers for independent tasks in a heterogeneous cluster [6, 7] or selection of devices under power caps in a heterogeneous CPU+GPU environment in order to minimize application execution time [8]. Energy-aware optimization has been demonstrated for various types of workloads and HPC systems, specifically for data-parallel applications [9, 10]. We shall observe that applying power caps for running parallel applications may also lead to considerable energy savings, specifically larger energy savings than time loss percentage wise for both CPUs [11] and GPUs [12, 13]. In this context, the goal of the paper is to extend the proposed programming model with a possibility to impose a global power limit on the GPUs selected for computations and minimizing execution time under the power cap.

Consequently, in this paper we contribute by the following:

1. Proposal of a multithreaded power-aware OpenMP+CUDA based programming model for a multi-GPU cluster environment. The solution hides internode communication behind the multithreaded model and allows minimization of execution time with setting an upper bound on the total power of selected compute devices.
2. Implementation of the model using a combination of OpenMP, MPI and CUDA, incorporating features such as using several streams per device in CUDA, distinct MPI communicators for handling messages between the master process and other nodes as well as multithreading within MPI for optimized communication.
3. Demonstration of scalability and adaptability of the solution in both homogeneous and heterogeneous environments with various GPUs such as: NVIDIA Quadro RTX 6000, Quadro RTX 5000, RTX 2080 and GTX 1060 in various configurations with 1 to 8 GPUs per node.

Programmers can benefit from the model and implementation by the possibility of much easier programming and execution, compared to an alternative approach required to obtain such

DOI: 10.1002/cpe

functionality. The latter would require manual coupling of MPI, OpenMP and CUDA and manual querying GPUs using NVIDIA NVML as well as coordination using MPI.

The outline of the paper is as follows. Section 2 provides a summary of related state-of-the-art works, on the topics of: models and frameworks for CPU+GPU cluster hiding details of inter-node communication and supporting optimization of processing and scalability; tools allowing transparent access to GPU resources and GPU virtualization in a parallel environment and finally performance-energy and power-aware computing in such systems. In Section 3, the proposed solution for multithreaded programming in a multi-GPU cluster environment is presented, with: the programming model and application architecture combining OpenMP and CUDA, optimization criteria including execution time and execution time with a power cap as well as implementation of automatic optimization of application execution with a greedy algorithm for device selection. Section 4 contains description of the three testbed applications, three testbed environments and corresponding results as well as discussion, including: performance-power profiles showing execution times under specific power caps, execution times versus the number of nodes and CUDA streams, speed-ups versus the number of nodes and finally assessment of the overhead of the framework. Section 5 includes summary and planned future work. Additionally, Appendix A contains a comprehensive list of the framework API, Appendix B describes the syntax and details of kernel invocation, Appendix C outlines code structure and dependencies including source files and Appendix D includes profiles of the tested workloads with execution times of successive data packets, executed either locally or remotely. Appendix E presents a concept allowing extension of the proposed implementation to use available CPU cores for computations, in a way similar to how the GPUs are used.

2. RELATED WORK

Explicit multi-node GPU programming using CUDA and MPI can benefit from CUDA-aware MPI implementations allowing usage of GPU memory pointers for GPU-GPU communication on various nodes. CUDA Inter-process Communication (IPC) allows GPU-GPU communication that can cross the process boundary [14] with optimizations for intra-node MPI communication for multi-GPU nodes shown in paper [15].

On the other hand, multithreaded programming models hiding details of inter-node communication details exist, both in the context of CPU as well as GPU based systems. One of the examples could be given with reference to the Partitioned Global Address Space (PGAS) programming model in which a set of processors with local storage is distinguished. At least parts of those storages are shared with others. Several compliant languages and libraries have been discussed in [16] including: original PGAS languages – CAF, Titanium, UPC; HPCS PGAS languages – Chapel, X10, Fortress; Retrospective PGAS languages – HPF, ZPL and GA as well as XCalableMP (XMP) – PGAS extension for C and Fortran. Notable recent examples include using PCJ for HPC systems [17], big data processing [18], clouds [19] as well as Shoal for clusters of processors and FPGAs [20]. HPX [21] is a C++ library developed for concurrency and parallelism that supports parallel, concurrent and distributed functions for general purpose programming, in particular Active Global Address Space (AGAS) that allows moving objects between nodes without

DOI: 10.1002/cpe

changing addresses. In paper [22] a PGAS-like memory level called world memory was proposed to extend OpenMP to distributed memory systems. The concept allows sharing data between processes as well as synchronization and the model with extensions would allow mapping to either multi-core CPUs or a GPU cluster. An example for a distributed memory Gauss-Seidel method using the extension was shown.

In [8] users can use clusters via an engine implemented with Java EE and subsequent communication with cluster managers is realized via SOA. Communication with node managers is performed with TCP and OpenCL kernels are used for computations. This allows using a collection of clusters with heterogeneous compute devices such as CPUs and GPUs. The optimization problem of minimization of data-parallel application execution time under imposed power limit for compute devices is solved by: selection of devices is used with solving a 0-1 knapsack problem (to which a greedy approximation algorithm was applied) and data partitioning for load balancing. Selection was generalized to to the 0/1 knapsack problem. It was shown that for a heterogeneous system with 3 heterogeneous nodes and 6 different compute devices and power caps between 100W and 1500W execution times follow theoretical values computed based on raw compute devices' performance for a parallel MD5 password breaking application, with kernels implemented using OpenCL.

A similar flexible approach for heterogeneous clusters and data-parallel applications was presented in the KernelHive system [23] where the engines managing the whole distributed system of clusters and each cluster are implemented with Java while for each node there is a C++ implemented daemon managing execution of OpenCL kernels. The system allows to plug in various optimizers including those considering power consumption of compute devices. Scaling of computations has been shown for an environment of up to 18 nodes for MD5 workflows. Compared to MPI+OpenCL, KernelHive introduced the overhead of 11% for up to 40 cluster nodes and 320 cores. Within a single node, CUDA Unified Memory provides a unified view on memory shared between the host and the GPU sides allowing effective programming for multi-CPU + multi-GPU single node systems [24]. This concept has been extended as Software Unified Memory (SUM) for HyCOMP (Hybrid Cluster OpenMP) meant for hybrid CPU/GPU clusters [25]. SUM relies on pages for consistency and uses the eager-release update protocol for coherence. It has been shown that for matrix multiplication HyCOMP is approx. 7% slower than hybrid MPI+CUDA for large data sizes. In paper [26] authors extended XCalableMP – a PGAS solution for clusters for multi-node GPU clusters requiring only small code changes with additional GPU-related memory allocation and sync clauses demonstrating good scalability between from 1, through 2 to 4 nodes for N-body simulation using an AMD Opteron+InfiniBand environment.

On the other hand, there also exists a variety of solutions that provide transparent access to GPU resources in a parallel environment. As outlined in [27], GPU virtualization solutions can be divided into two groups: within VMs – typically using shared memory for data transfer from main memory in VM to the GPU memory; and in either the native or the VM domain – typically using the network layer for data transfer between the client and possibly a different server with a GPU. Examples within the former group include: vCUDA, GViM, gVirtuS, Shadowfax while the latter: rCUDA, GridCuda, DS-CUDA, Shadowfax II. Another approach extending a single node programming model to a cluster was proposed in [28] for OpenCL as an API for GPU clusters. When a request to a GPU is issued from a host thread to a GPU on a compute node, it is enqueued to a per-device command queue and then sent by a command scheduler on a host node via an

DOI: 10.1002/cpe

inter-node interconnect to a command scheduler on the compute node. Subsequently, the request goes through a per-GPU ready queue and device thread to a GPU. A result is passed through a completion queue and the command scheduler on the compute node, via a completion queue and the command scheduler on the host node. The authors used a host node with 2 Intel Xeon X5680 CPUs and 72GB RAM and 8 compute nodes with 2 Intel Xeon 5660 CPUs with 24GB RAM and 4 NVIDIA GTX 480 GPUs. They benchmarked the solution for several applications: Binomial option pricing (BO), Black-Scholes PDE (BS), Coulombic Potential (CP), Embarrassingly Parallel (EP), Matrix Multiplication (MM) and Nbody simulation (NB). The authors showed very good speed-ups over 1 GPU close to 28 for BO, exceeding 28 for EP and close to 30 for BS and CP using 32 GPUs. For the two applications requiring much more communication i.e. NB and MM the speed-up was approx. 24 for NB and severely hit barely exceeding 4 for 32 and approx. 6 for 16 GPUs. Another solution allowing to use CUDA for distributed heterogeneous CPU+GPU clusters was proposed in [29]. Specifically, the authors proposed BigGPU which is distributed system that can recompile and execute PTX codes on CPU/GPU devices within the cluster. The programmer provides just the standard CUDA code and the system can distribute and balance the load among computational devices. For experimental evaluation the authors used 4 workstations with Intel Xeon E5645, 24 GB RAM and NVIDIA GeForce GTX 550 Ti as well as 10-Gigabit Ethernet. Three applications were benchmarked: matrix multiplication (MM), motion estimation (ME) and N-body simulation (NB). Scaling between 1, 2 and 4 GPUs was shown with execution times dropping between 1 to 4 GPUs for large data sizes for the applications: from 110 to over 50s for MM from 17 to 8s for ME, from 1100 to 300s for NB. In a heterogeneous CPU+GPU environment with 4 nodes, benefits from adding CPUs were visible for NB and slightly for ME only. Main costs affecting speed-ups were the memory copy and data-consistency mechanisms. In paper [30] authors proposed SnuCL-D – a system for scalable execution of standard (i.e. single node) OpenCL codes in a distributed environment with replication of the OpenCL host program and data in each node. It has been compared to SnuCL with the centralized node approach using a microbenchmark that copies contents of buffers to other buffers showing significant speed-ups over SnuCL for more that 32 nodes, even 78 times faster using 512 nodes and 4096 cores (2 Intel Xeon x5570 per node). The authors also used a series of benchmarks in further tests such as blackscholes, BinomialOption, CP, N-body, matrix multiplication and NPB applications: EP, FT, CG, MG, SP and BT. For a GPU cluster with 36 nodes, each with 2 Intel Xeon E5-2650 CPUs, 128GB RAM and 4 AMD Radeon HD7970 GPUs SnuCL-D visibly outperforms SnuCL for 32 nodes (128 GPUs) for N-body, matrix multiplication, FT and SP, BT, CG and MG but did not scale well itself due to small input data size for CG, MG, SP and FT. In paper [31] authors proposed a solution for offloading computations from an OpenMP program to a cluster appearing as an accelerator device, transforming OpenMP directives to Spark calls. For applications such as 2MM and 1GB sparse matrices, the authors achieved speed-ups of 115 using 240 cores as well as for an application solving the Collision Cross-Section problem the speed-up of 80 using 320 cores.

In [27], the authors studied benefits of using remote GPU virtualization, specifically: allowing a larger number of GPUs to be available to an application; allowing access to remote GPUs even if most or all CPU resources on that node are allocated to a non-GPU application (however, as per studies in [32, 33] still a CPU core for accessing a GPU such as for the rCUDA daemon would be preferable for performance reasons); increasing GPU utilization, minimization of workload

DOI: 10.1002/cpe

execution time as well as energy consumption; the possibility of using one GPU from several VMs. Specifically, the authors studied use cases with 3 workloads: one composed of 400 instances using GPU-Blast, LAMMPS, mCUDA-MEME, GROMACS; second one composed of 400 instances using BarraCUDA, MUMmerGPU, GPU-LIBSVM, NAMD and the third combined one. Using a 16 node cluster (and one access node) with nodes featuring 2 Xeon E5-2620 v2 CPUs, 1 NVIDIA K20 GPU with InfiniBand (FDR) interconnect they demonstrated that using rCUDA versus the traditional CUDA API it was possible to obtain for the three workloads: 48%, 37% and 27% reduction in execution times respectively, approximately doubling GPU utilization and consequently reduction of energy consumption by 40%, 25% and 15% respectively. The feature of decoupling CPU and GPU computational parts using rCUDA is also studied in [34] where sharing of physical GPUs can result in an increase of system throughput up to two times and in reduction of energy consumption by approx. 15%. Features such as creating virtual instances out of the physical GPU can be exploited and communication and computations can be overlapped for reduction of time and energy. In [35] benefits of using rCUDA with a modified Slurm scheduler capable of assigning remote GPUs to jobs are shown, resulting in increasing the cluster throughput two times and in energy reduction up to 40%.

Many works present tools that assist in finding better than default performance-energy trade-offs, typically assuming or well suited to certain processing models. In such cases, these are not used at the program/API level but rather as external tools. Examples include browsing the available space of power caps for NVIDIA GPUs optimizing energy consumption, Energy Delay Product (EDP) and Energy Delay Sum (EDS) for training models such as Alexnet, VGG-19, Inception V3, Inception V4, Resnet50 and Resnet152 using Nvidia Quadro RTX 6000 and Nvidia V100 GPUs [12]. As an example, for V100 average energy savings of 24%–33%, for EDP 23%–27% with performance loss of 13%–21% and for EDS (k=2) 23.5%–27.3% with performance loss of 4.5%–13.8% were observed. In [36] a GPOEO solution was proposed, developed specifically for iterative machine learning applications. The tool measures performance counter as well as energy online and time and energy models are used to find best predicted configuration that optimizes a function of time and energy. The authors demonstrated mean 16.2% energy savings at the 5.1% performance loss for an environment with an AMD 5950X CPU + NVIDIA RTX3080Ti GPU run for 71 ML applications. At a higher level, in paper [37] authors proposed an energy consumption cost efficient deep learning job allocation (CE-DLA) solution for training and inference jobs assuming the dynamic electricity price. They designed a mixed integer nonlinear programming formulation and demonstrated 43% better energy consumption cost than PA-MBT and 15% than EPRONS competitors assuring acceptable performance.

Power capping is considered in the context of an incoming stream of jobs scheduled on HPC systems for e.g. maximizing throughput under an imposed power cap. For instance, Slurm allows to set power caps using DVFS as well as shut down and start nodes [38]. In paper [39], PowerCoord was proposed for systems with multiple CPU and GPU sockets and is able to coordinate between the CPU and GPU domains in order to maximize throughput and maintain the power under the imposed cap. The approach uses reinforcement learning in order to select one of several heuristic allocation policies. Throughput improvement of 18% compared to a case without coordination between domains.

DOI: 10.1002/cpe

3. PROPOSED SOLUTION FOR MULTITHREADED PROGRAMMING IN A MULTI-GPU CLUSTER ENVIRONMENT

In the context of the aforementioned existing works we contribute by providing an OpenMP+CUDA based shared memory multi-threaded programming model and implementation [†] for a GPU cluster, i.e. a cluster that consists of several nodes with 1+GPU(s) per node, with power-aware processing, applicable at runtime.

3.1. Programming model and application architecture

Using a single node with potentially several GPUs typically involves coupling APIs [33] such as:

1. CUDA – for management of each GPU, including memory management, data copying, kernel invocation (possibly using several CUDA streams) etc.
2. OpenMP (or other multithreaded API) – for management several GPUs i.e. a dedicated thread manages its own GPU, synchronizing for input data and writing down results on the host side with other threads.

The proposed solution assumes the architecture of a parallel application implemented with OpenMP for launching multiple threads, each of which manages its own GPU. In this paper we propose to maintain this application architecture i.e. the view of one process and multiple threads but handling not only local GPUs but also remote GPUs in a cluster in a transparent way. The OpenMP part is maintained without changes and CUDA functions have been overridden with their analogous counterparts starting with `__cudampi__`, followed by the traditional CUDA call e.g. `__cudampi__cudaSetDevice()` instead of `cudaSetDevice()` with same arguments.

The parallel application consists of several processes, each of which runs on a different node with a GPU or GPUs. Implementation wise, the inter-node communication is realized with MPI that can, depending on the implementation, provide low latency and high throughput communication between nodes such as using Infiniband. Furthermore, we require the implementation to support the `MPI_THREAD_MULTIPLE` mode for threads to communicate with other processes independently.

As shown in Figure 1, within process with MPI rank 0, several threads are created. Each of these manages a different GPU, which is either local for process 0 or remote on a different node. In the former case, commands are passed directly to a GPU using CUDA. In the case of a remote GPU, commands are passed using MPI to an appropriate process on a remote node on which the given GPU is located. Furthermore, within that process there is a thread dedicated to that particular GPU that manages it locally via CUDA.

Consequently, within each remote (slave) MPI process, there are as many threads as the number of its local GPUs. Each such thread sets its current GPU with `cudaSetDevice()` and proceeds with responding to requests from process 0's thread. Since there are communication pairs between process 0 and a given process referring to different GPUs, this will allow independent and potentially parallel management of GPUs assuming multi-core CPUs at both ends. For such communication to take place, we have created independent MPI communicators each of which is dedicated to a

[†] available at https://kask.eti.pg.gda.pl/gitlab/pczarnul/cudampilib

DOI: 10.1002/cpe

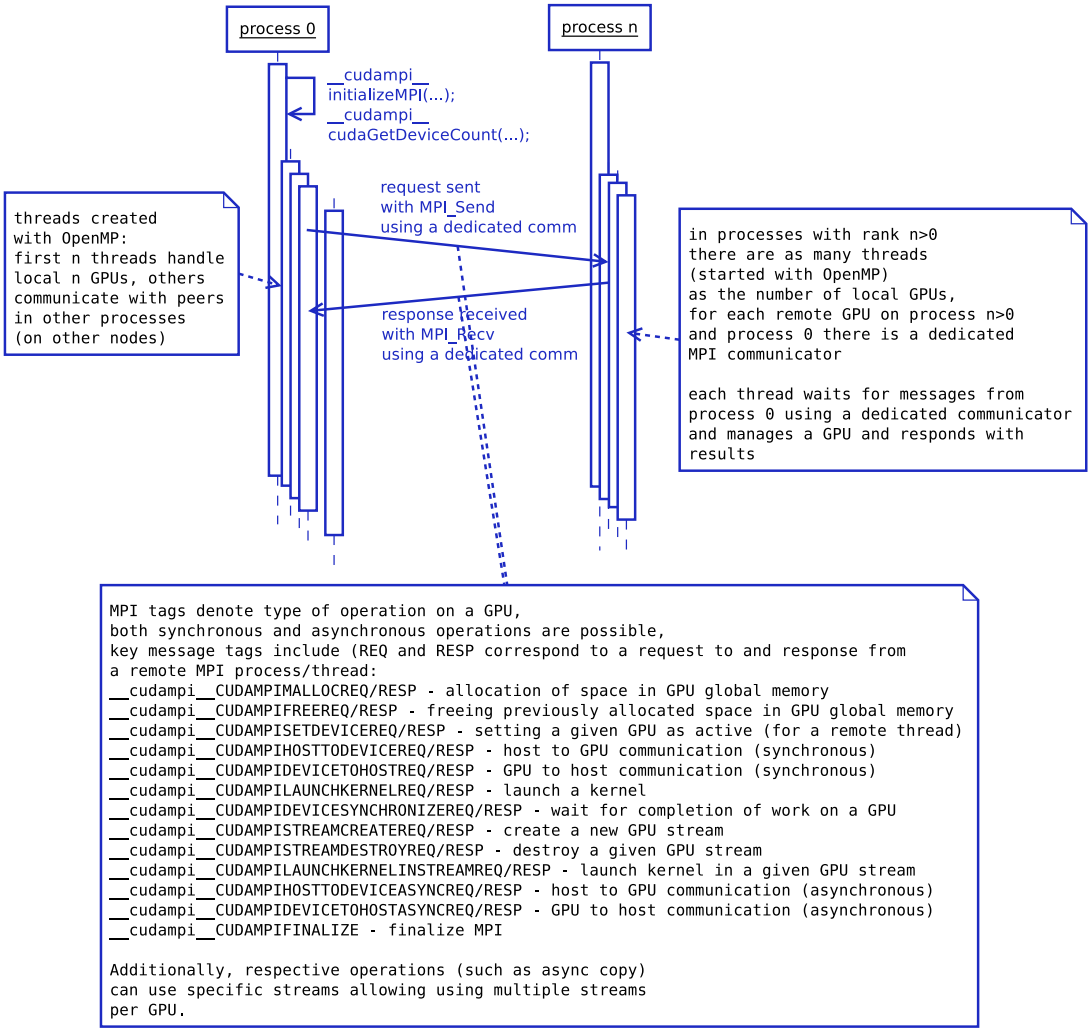

Figure 1. Interaction diagram outlining processes, threads, MPI (communicators) and message tags

different target GPU – effectively used by one pair of threads in process 0 and the process which resides on a node of a given GPU. Then MPI tags refer to various types of messages and operations supported by the framework for remote GPUs, as outlined in Figure 1.

Because of the specific CUDA kernel invocation syntax, the proposed solution involves an assumption to pass to a kernel a pointer to a previously allocated space in GPU's global memory, which is a common practice. Prior to invocation of a kernel, input parameters are copied to that global memory in a GPU rather than passed by values in the kernel invocation. We assume that each grid's thread would then fetch input data based on its global id. This approach allows to hide the proposed implementation practically completely. We shall note that OpenCL's syntax would allow to override setting kernel parameters in an easier way than for CUDA but, in fact, this is just a technical detail.

3.2. Additional API

Additional, compared to CUDA, helper calls are supported, including:

DOI: 10.1002/cpe

1. `void __cudampi__kernelinstream(void *devPtr,cudaStream_t stream)` for launching a kernel in a given `stream` with arguments `devPtr`. The kernel itself is launched in function `launchkernelinstream(void *devPtr,cudaStream_t stream)` which invokes the kernel code provided by the programmer in function `appkernel(void *devPtr)`. Details on how the CUDA grid can be configured for this invocation is shown in Appendix B.
2. `int __cudampi__getnextchunkindex(long long *globalcounter,int batchsize,long long max)` for a given thread in process 0 a new data packet index is returned (using an OpenMP critical section) for processing on the GPU. Returns the next available data chunk index, skipping by `batchsize`. `max` is returned when the given GPU is disabled from further computations. Further details on how this function is used to disable a particular GPU from further usage are described in Section 3.5. `globalcounter` points at the global counter (guarded by a critical section).
3. `void __cudampi__initializeMPI(int argc,char **argv)` – initializes the MPI environment with multithreading support passing the arguments of the `main()` function.
4. `void __cudampi__terminateMPI()` – terminates the multi-node MPI environment.
5. `void __cudampi__setglobalpowerlimit(float powerlimit)` – sets a global, total power limit ([W]) for the GPUs selected for application execution.
6. `__cudampi__cudaSetDevice()` – sets an active GPU – invoked by each thread of process 0, an actual active GPU is set using an equivalent CUDA call either locally or remotely, depending on the GPU.

3.3. Application template

The application structure, implemented by the programmer, considering the proposed model and processing of batchsize sized vectors, using either 1 or 2 streams is shown in Listing 1.

Furthermore, the design of the solution allows for usage of many CUDA streams per GPU, which has been implemented in the tested software. By default, functions such as data copy and kernel invocation without explicit stream passing can be used. Additionally, standard CUDA stream creation, management and usage can be used, which allows for e.g. using 2+ streams per GPU from one CPU thread managing a GPU for hiding CPU-GPU communication and kernel runs in various streams. In this case, stream creation and management is passed to the target GPU thread, if run remotely.

3.4. Optimization criteria

Within the framework, two alternative optimization criteria are considered and realized:

minimization of execution time – as in a dynamic master-slave approach where threads managing GPUs request successive data chunks for processing. Effective load balancing is possible if the number of data chunks is considerably larger (e.g. a few times larger more) than the number of GPUs and thanks to the dynamic master-slave technique.

minimization of execution time with a limit on the total power consumption of used GPUs – in this case a greedy algorithm selects such GPUs so that their total power consumption does

DOI: 10.1002/cpe

```
__cudampi__initializeMPI(argc,argv);
__cudampi__setglobalpowerlimit(powerlimit);
__cudampi__cudaGetDeviceCount(&cudadevicescount);
cudaHostAlloc((void**)&vectora,sizeof(char)*VECTORSIZE,cudaHostAllocDefault);
cudaHostAlloc((void**)&vectorc,sizeof(char)*VECTORSIZE,cudaHostAllocDefault);
initialize_vectors(...);
#pragma omp parallel num_threads(cudadevicescount)
{ long long mycounter; int finish=0;void *devPtra,*devPtrc,*devPtra2,*devPtrc2;
     int i; cudaStream_t stream1,stream2; int mythreadid=omp_get_thread_num();
    void *devPtr,*devPtr2; long long privatecounter=0;
  __cudampi__cudaSetDevice(mythreadid);
  #pragma omp barrier
  __cudampi__cudaMalloc(&devPtra,batchsize*sizeof(char));
  __cudampi__cudaMalloc(&devPtrc,batchsize*sizeof(char));
  __cudampi__cudaMalloc(&devPtr,2*sizeof(void *));
  __cudampi__cudaMalloc(&devPtra2,batchsize*sizeof(char));
  __cudampi__cudaMalloc(&devPtrc2,batchsize*sizeof(char));
  __cudampi__cudaMalloc(&devPtr2,2*sizeof(void *));
  __cudampi__cudaStreamCreate(&stream1);
  __cudampi__cudaStreamCreate(&stream2);
  __cudampi__cudaMemcpyAsync(devPtr,&devPtra,sizeof(void *),
  cudaMemcpyHostToDevice,stream1);
  __cudampi__cudaMemcpyAsync(devPtr+sizeof(void *),&devPtrc,
  sizeof(void *),cudaMemcpyHostToDevice,stream1);
  __cudampi__cudaMemcpyAsync(devPtr2,&devPtra2,sizeof(void *),
  cudaMemcpyHostToDevice,stream2);
  __cudampi__cudaMemcpyAsync(devPtr2+sizeof(void *),&devPtrc2,
  sizeof(void *),cudaMemcpyHostToDevice,stream2);
  do {
    mycounter=__cudampi__getnextchunkindex(&globalcounter,batchsize,VECTORSIZE)
        ;
    if (mycounter>=VECTORSIZE) finish=1; else {
      __cudampi__cudaMemcpyAsync(devPtra,vectora+mycounter,batchsize*sizeof(
          char),cudaMemcpyHostToDevice,stream1);
      __cudampi__kernelinstream(devPtr,stream1);
      __cudampi__cudaMemcpyAsync(vectorc+mycounter,devPtrc,batchsize*sizeof(
          char),cudaMemcpyDeviceToHost,stream1);
    }
    if (streamcount==2) // do it again in the second stream
      if (!finish) {
        mycounter=__cudampi__getnextchunkindex(&globalcounter,batchsize,
            VECTORSIZE);
        if (mycounter>=VECTORSIZE) finish=1; else {
          __cudampi__cudaMemcpyAsync(devPtra2,vectora+mycounter,batchsize*
              sizeof(char),cudaMemcpyHostToDevice,stream2);
          __cudampi__kernelinstream(devPtr2,stream2);
          __cudampi__cudaMemcpyAsync(vectorc+mycounter,devPtrc2,batchsize*
              sizeof(char),cudaMemcpyDeviceToHost,stream2);
        }
      }
    privatecounter++;if (privatecounter%2) __cudampi__cudaDeviceSynchronize();
  } while (!finish);
  __cudampi__cudaDeviceSynchronize(); __cudampi__cudaStreamDestroy(stream1);
}
__cudampi__terminateMPI();
```

Listing 1: Template for multi-node multi-GPU programming with the proposed framework

not exceed a given limit and selects available devices with best ratio of performance/power consumption.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

```
input: set D_in of available devices d_i, power limit P
output: set D_out of selected devices
avP = P;
do {
  D = D_in;
  find such d_i in D:
  1. power_{d_i} ≤ avP; and
  2. performance_{d_i}/power_{d_i} is largest across all i in D;

  remove d_i from D;
  add d_i to D_out;
} while (d_i has been found);
```

Listing 2: Greedy algorithm for device selection

3.5. Implementation of automatic optimization of application execution

For the purpose of automated optimization of application execution, both execution times of processing particular chunks of data on a given GPU including communication (i.e. turnaround times for data packets for particular devices) as well as power consumption of a given GPU are monitored. The latter is performed by calling `nvidia-smi`. Specifically, in the framework, in the threads of MPI's process 0, for each GPU, a typical sequence of asynchronous calls is inserted into each stream i.e.: asynchronous host to device communication, kernel launch and asynchronous device to host communication. Since these calls are asynchronous from the point of view of the host, after a number (which can be defined) of such sequences has been started, device synchronization is invoked. This is needed since it then allows load balancing of data packet distribution across GPU devices (otherwise all data packets would have been spread across devices immediately without proper dynamic load balancing). This also means that after launching the calls, GPU is busy and at this point, before blocking `cudaDeviceSynchronize()`, a power readout is performed.

At the same time, round trip times for data chunks for all GPUs (via the local bus for local GPUs and via MPI using Ethernet or Infiniband for remote GPUs) are measured periodically, between synchronization points.

In the power-aware version, which is activated by calling a power limit by function `__cudampi__setglobalpowerlimit(powerlimit)`, after round trip times and power readouts have been gathered, thread 0 in process 0 performs device enabling (OpenMP locks are used for particular threads managing GPUs) based on their performance and power consumption, according to the greedy algorithm shown in Listing 2.

Performance of a given device can be either assigned a priori, for instance measured using a benchmark corresponding to the workload being optimized, or measured dynamically as an inverse of data chunk processing time. In the latter case, processing packets assigned to devices should be of similar size computationally wise before the next device selection using such data takes place. This issue is solved by performing this measurement in the overloaded implementation of function `cudaDeviceSynchronize()` i.e. `__cudampi__cudaDeviceSynchronize()` which averages execution of several data chunks per GPU. From this point on, function `__cudampi__getnextchunkindex(long long *globalcounter,int batchsize,long long max)`, depending on the GPU, either returns next data chunk index to be processed (batches of `batchsize` per GPU are considered) or

Prepared using cpeauth.cls
DOI: 10.1002/cpe

skips to `max` which effectively informs the given thread that there are no more data packets available to a given thread, in turn eliminating the given GPU from subsequent computations.

Additionally, switching between threads managing selection of devices has been implemented to handle a situation in which some threads have become inactive and thus cannot continue to handle this function. One of the threads handling an enabled/active device is selected to perform this task.

4. EXPERIMENTS

4.1. Testbed applications

The proposed framework allows for easy implementation of various applications that fall within the adopted model suitable for GPU(s) i.e. the one in which input data can be partitioned into data chunks that can be processed in parallel and which output can be integrated afterwards. For the sake of tests, the following applications have been implemented and subsequently tested with 1 and 2 streams per GPU:

1. finding a predefined number of pattern(s) PATTERNCOUNT of length PATTERNLENGTH (400 and 400 benchmarked) in a given input char vector (length=4e8, batchsize=5e4 for testbeds 1 and 3; length=2e9, batchsize=1e6 for testbed 2) and returning a vector stating occurrence or lack of thereof at a given location of the vector (index),
2. checking the Collatz conjecture for numbers passed as input in vector v_a and returning vector v_c that in $v_c[i]$ stores the number of iterations to reach number 1 according to the Collatz procedure for input number in $v_a[i]$ (2e8 numbers, batchsize=5e4 for testbeds 1 and 3; 1e9 numbers, batchsize=1e6 for testbed 2),
3. finding the largest divisor out of elements $v_a[i]$ and $v_b[i]$ (vector size=2e8, batchsize=5e4 for testbeds 1 and 3; vector size=1e9, batchsize=1e6 for testbed 2).

The rationale of testing various applications was to assess performance of the solution for potentially various ratios of compute to communication (host-to (via local or network interconnect) -device). Additionally, the applications differ in compute times across particular data chunks for a given problem. For the pattern search application execution times of data packets of a given size (numbers of elements in a data chunk) will also depend on values of particular elements. The proposed scheme will still balance load, provided the number of data chunks is considerably larger than the number of compute devices and relative ratios of execution times across data chunks are limited.

4.2. Testbed environments

The proposed implementation incorporates support for heterogeneous environments in which various nodes can feature various numbers of GPUs. This is allowed by specification of a configuration file that includes lines with the $< nodeid/MPIrank >:< GPUcount >$ syntax. For the purpose of tests, several environments have been used, that differ in: the number of GPUs and nodes as well as relative performance of GPUs in the environment, as follows:

DOI: 10.1002/cpe

Testbed 1: a cluster of 16 nodes, each with an NVIDIA GTX 1060 GPU and Gb Ethernet, Intel(R) Core(TM) i7-7700 CPU at 3.60GHz, 16 GB RAM;

Testbed 2: 2 very powerful nodes, one with: 8x NVIDIA Quadro RTX 6000 GPUs, 2 Intel(R) Xeon(R) Silver 4210 CPUs at 2.20GHz, 384 GB RAM, second with 4x NVIDIA Quadro RTX 5000 GPUs, 2 Intel(R) Xeon(R) Silver 4210 CPUs at 2.20GHz, 384 GB RAM;

Testbed 3: a heterogeneous environment with the following nodes: one with 2x NVIDIA RTX 2080, 2 Intel(R) Xeon(R) Gold 6130 CPUs at 2.10GHz, 256 GB RAM; 2 nodes, each with an NVIDIA GTX 1060 GPU, Intel(R) Xeon(R) CPU W3540 at 2.93GHz, 24 GB RAM.

4.3. Results

Figures 2, 3 and 4 present execution times versus imposed power limits, for the tested pattern search application, for all testbeds 1-3, respectively. Analogous charts for the collatz application are shown in Figures 5, 6 and 7 while for finding largest divisors (vecmaxdiv) in Figures 8, 9 and 10.

Figures 11, 12 and 13 depict execution times for the testbed applications: pattern search, collatz and vecmaxdiv respectively – using testbed 1, for various numbers of nodes (1-16) as well as either 1 or 2 streams per GPU. Figures 14, 15 and 16 present speed-ups for pattern search, collatz and vecmaxdiv respectively.

Finally, we can compare speed-ups of the proposed framework to its time on 1 node versus the time on 1 node obtained by the OpenMP+CUDA implementation, as a performance optimized baseline. Results are shown in Figures 17, 18 and 19 for the three applications respectively.

4.4. Discussion

4.4.1. Performance-power profiles From Figures 2 through 10 we can see that for all configurations the framework adjusts the assignment of data packets to the nodes which results in increasing execution times, for smaller power limits, similarly to the approach shown in paper [8]. Interestingly, this is visible either for both a homogeneous environment with 16 nodes (testbed 1) as well as even a small heterogeneous environment such as testbed 3, albeit with steps visible in the chart due to the small number of compute devices.

4.4.2. Execution time vs number of nodes and CUDA streams As noted in paper [33], 2 streams per GPU shall bring visible benefits for GPU applications processing independent data packets, as is the case for the applications considered in this paper (in that model 2 data packets resulted in one output packet corresponding to e.g. multiplication or addition of matrices). In Figures 11 through 13 we can see that the proposed implementation clearly results in improvement of execution times from using 2 versus 1 stream per GPU, for all applications and all configurations, e.g.: for pattern search 7.5% for 1 node, 6.4% for 8 and 5.4% for 16 nodes; for collatz 5.1% for 1 node, 5.4% for 8 and 4% for 16 nodes; for vecmaxdiv 0.5% for 1 node, 2.2% for 8 and 1.4% for 16 nodes. This means that the proposed solution can benefit from multiple CUDA streams in a GPU cluster.

4.4.3. Speed-ups vs number of nodes Apart from actually measured speed-ups, those can be compared to theoretical values assuming ideal load balancing across the compute devices.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

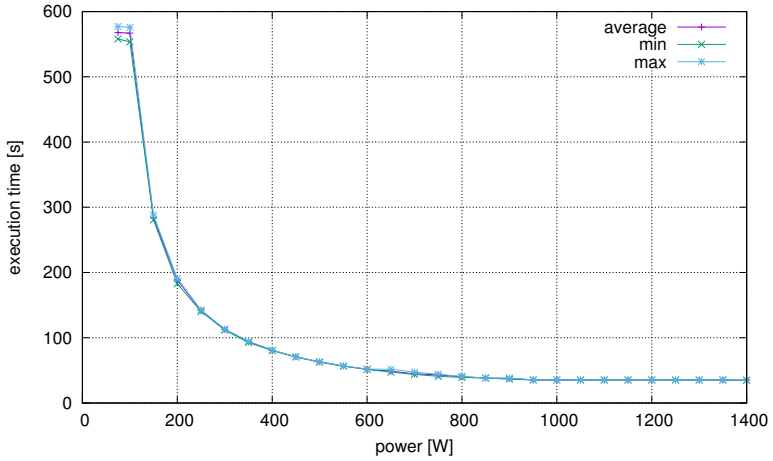

Figure 2. Pattern search application – execution time vs power limit, testbed 1

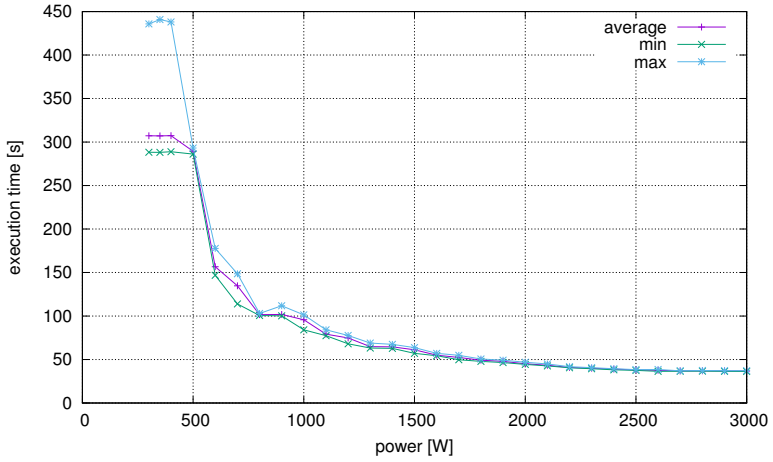

Figure 3. Pattern search application – execution time vs power limit, testbed 2

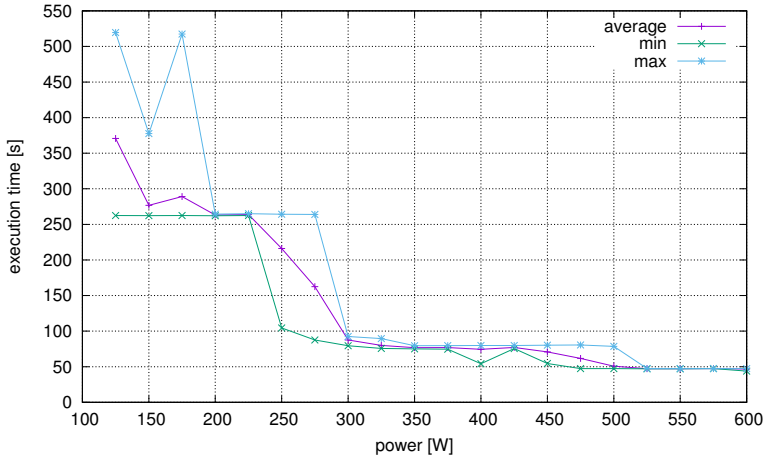

Figure 4. Pattern search application – execution time vs power limit, testbed 3

DOI: 10.1002/cpe

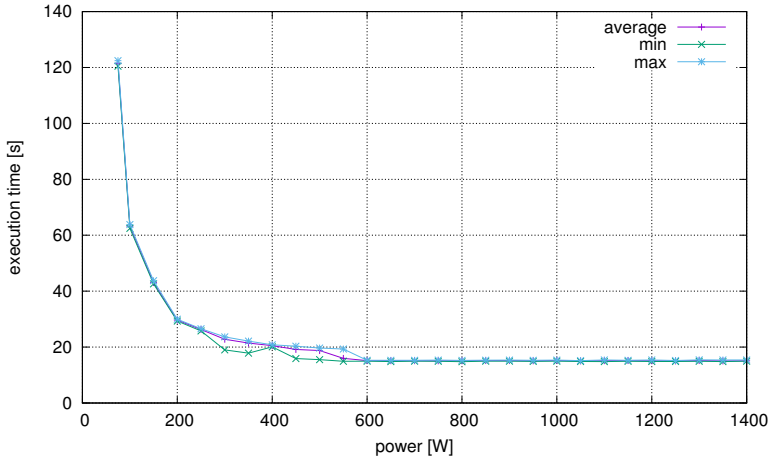

Figure 5. Collatz application – execution time vs power limit, testbed 1

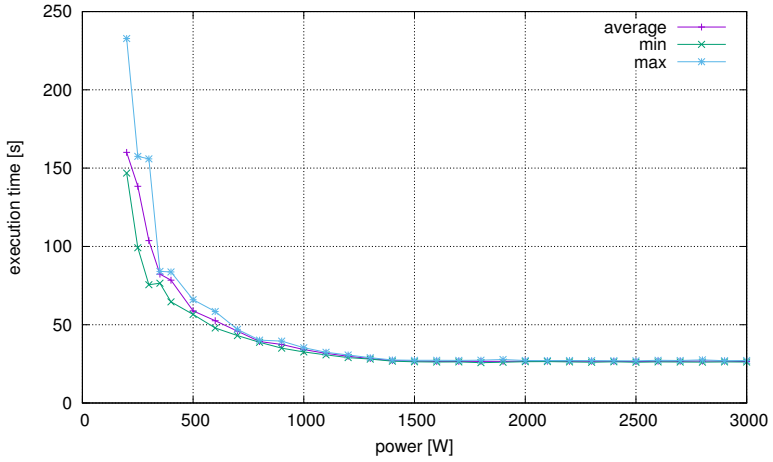

Figure 6. Collatz application – execution time vs power limit, testbed 2

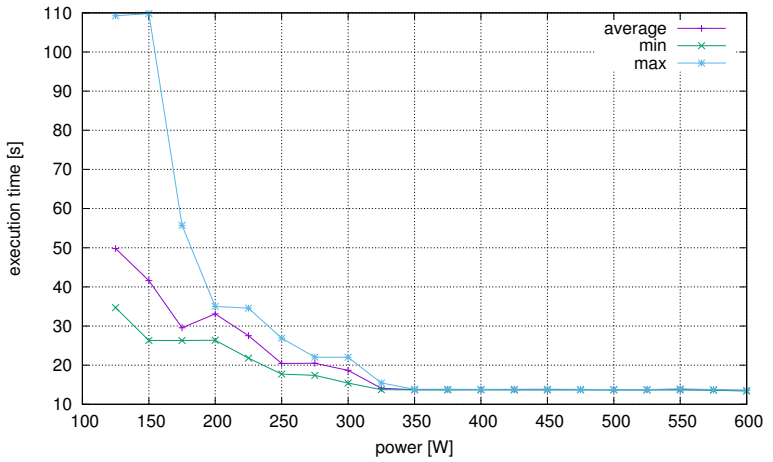

Figure 7. Collatz application – execution time vs power limit, testbed 3

Prepared using cpeauth.cls
DOI: 10.1002/cpe

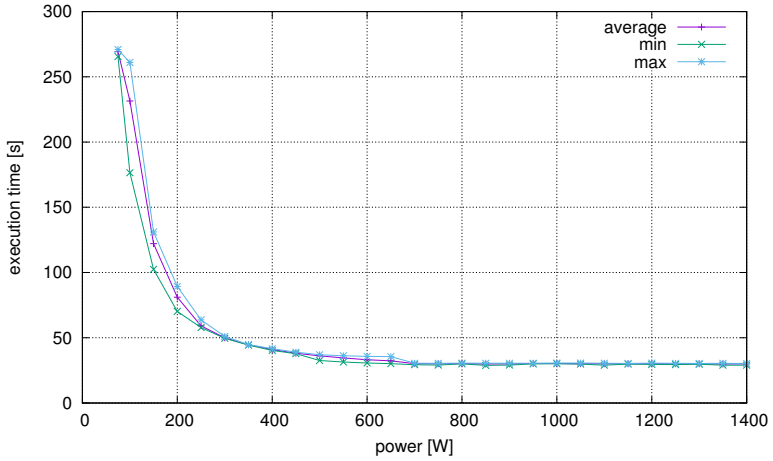

Figure 8. Vecmaxdiv application – execution time vs power limit, testbed 1

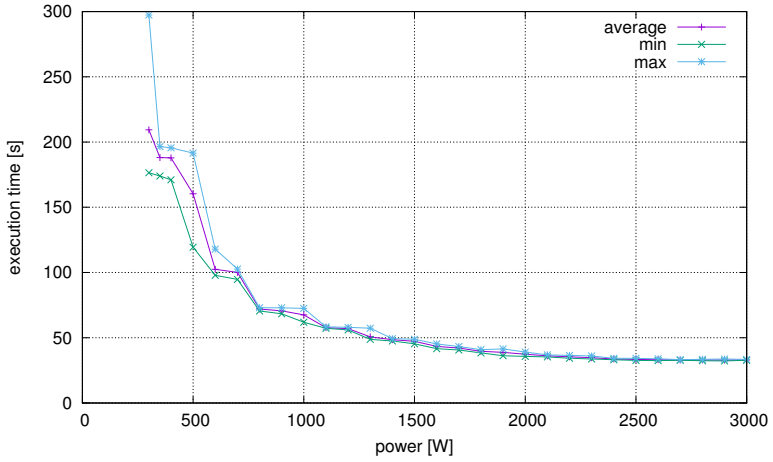

Figure 9. Vecmaxdiv application – execution time vs power limit, testbed 2

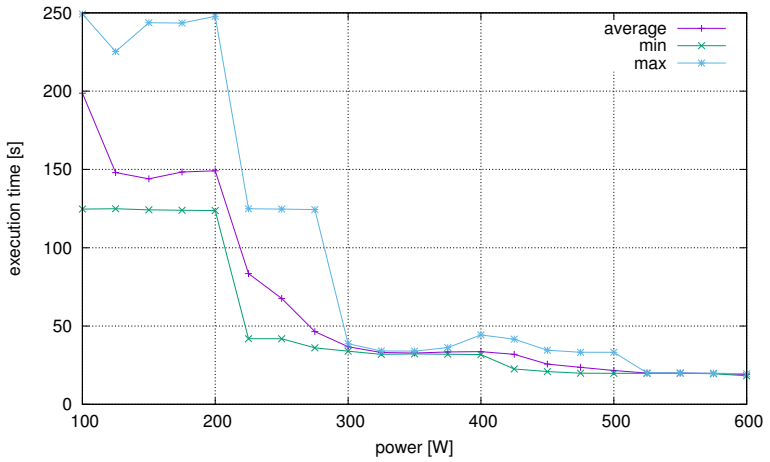

Figure 10. Vecmaxdiv application – execution time vs power limit, testbed 3

DOI: 10.1002/cpe

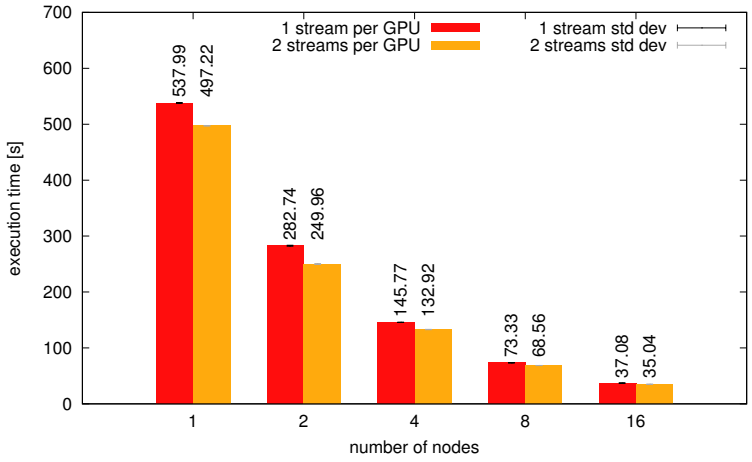

Figure 11. Pattern search application – execution time vs number of nodes, 1 or 2 streams per GPU, testbed 1

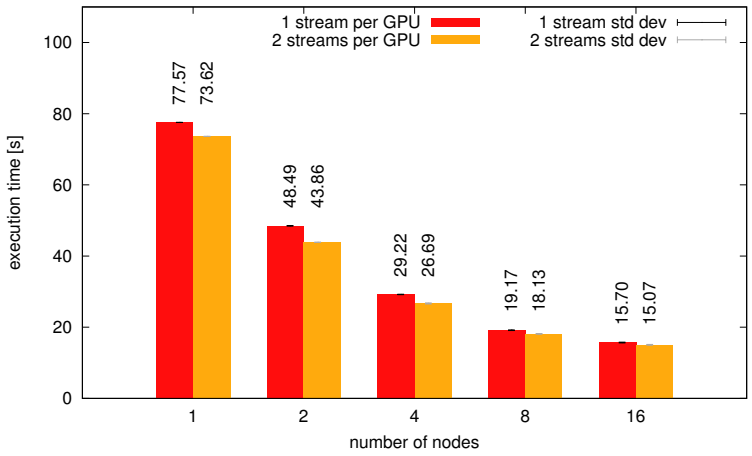

Figure 12. Collatz application – execution time vs number of nodes, 1 or 2 streams per GPU, testbed 1

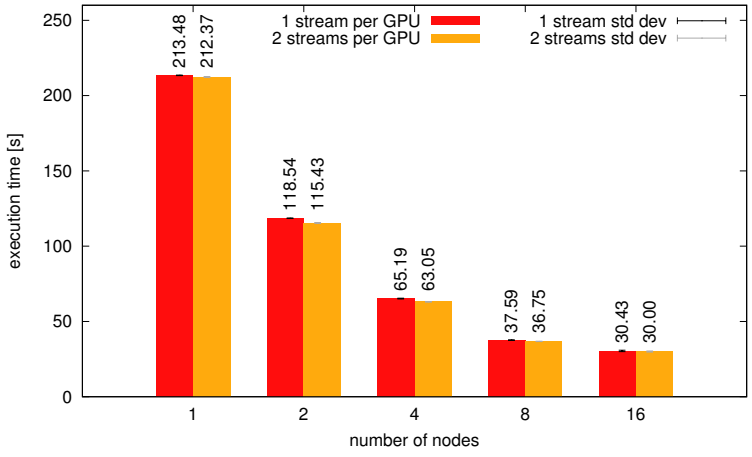

Figure 13. Vecmaxdiv application – execution time vs number of nodes, 1 or 2 streams per GPU, testbed 1

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Specifically, based on profiles of the workloads from testbed 1 shown in Figures A2, A3 and A4 in Appendix D, we can compute relative performance of the nodes including communication based on these samples. Those values, assuming then performance of node 0 $perf_0$ as well as the remaining N-1 nodes with $perf_1$, assuming ideal load balancing, can be used to assess ideal speed-ups. Specifically, data packets processed fast i.e. for approx. 0.05s in Figure A3 denote packets processed on a local GPU while the others on remote GPUs. Taking into account border cases of shortest remote times (and corresponding potentially best i.e. largest performance $perf_1$ regarded as inverse of the execution time) and longest local times (corresponding performance $perf_0$) we can compute (assuming ideal load balancing) theoretically best speed-ups on 16 nodes (assuming 1 stream) as $sp(16) = 1 + 15\frac{perf_1}{perf_0}$. Such comparisons of speed-ups obtained using either 1 or 2 streams per GPU to the theoretically ideal computed as suggested above are shown in Figures 14, 15 and 16 for pattern search, collatz and vecmaxdiv respectively. We can see that the applications differ in the potentially best speed-up considerably, which stems from their profiles and potential for load balancing which is visible in execution times of successive 200 samples, shown in the profiles above. Nevertheless, speed-ups obtained for 2 streams per GPU and 16 nodes are close to the best assessed theoretical ones: for pattern search – 89.3% of the theoretically ideal, for collatz – 89.6% of the theoretically ideal, for vecmaxdiv – 97.4% of the theoretically ideal.

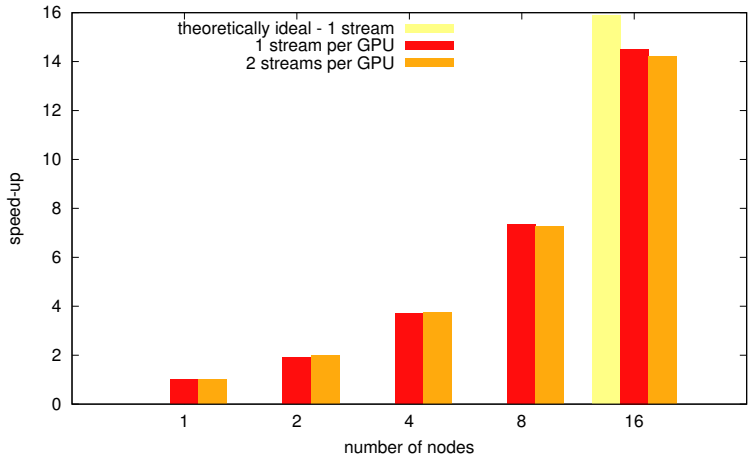

Figure 14. Pattern search application – speed-up vs number of nodes, 1 and 2 streams per GPU, testbed 1

4.4.4. Overhead of the framework and comparison to other results In order to fully evaluate usefulness and performance of a framework, it is necessary to assess the overhead of the latter compared to an implementation based solely on low level parallel programming APIs, presumably resulting in lowest practically obtained overheads. As an example, the performance of the KernelHive framework was compared to the performance of an MPI+OpenCL for a parallel geospatial interpolation application on up to 40 cluster nodes and 320 cores [23].

In this case, this has been done by comparison of times with the use of the proposed framework versus an implementation using just OpenMP+CUDA on 1 node. From Figures 17 through 19 we can see that the overheads of the proposed framework, considering 2 streams per GPU, are perfectly acceptable and speed-up drop versus the configuration on 1 node without the framework is as follows

DOI: 10.1002/cpe

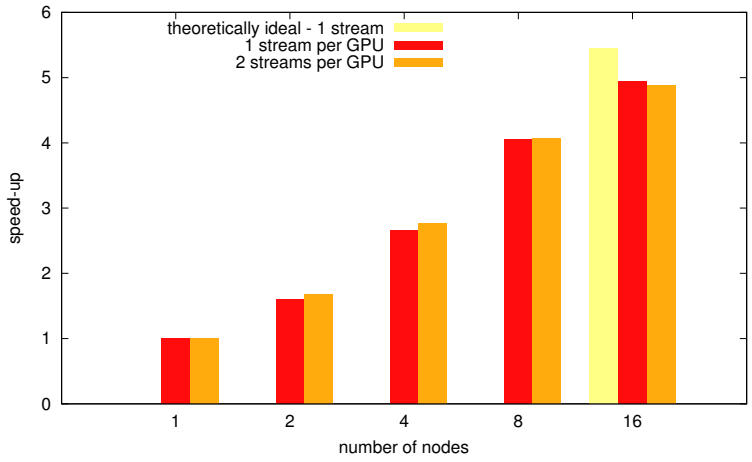

Figure 15. Collatz application – speed-up vs number of nodes, 1 and 2 streams per GPU, testbed 1

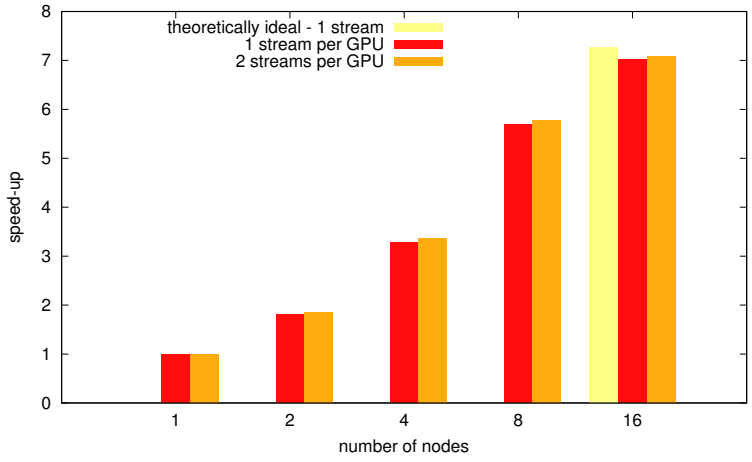

Figure 16. Vecmaxdiv application – speed-up vs number of nodes, 1 and 2 streams per GPU, testbed 1

using 16 nodes: for pattern search – 1.398 (9.9%), for collatz – 0.400 (8.2%), for vecmaxdiv – 0.377 (5.3%).

We can compare results obtained in this paper to those from KernelHive, described in Section 2 and referenced above. This comparison is valid since both systems follow a similar dynamic master-slave strategy for distribution of data packets and integration and both can utilize GPUs across several nodes of a system which can be homogeneous or heterogeneous. The difference is that KernelHive uses OpenCL kernels which at a similar programming abstraction level as CUDA but the Java technology for integration of nodes than MPI. While in paper [23] KernelHive was benchmarked with a MD5 password-breaking application, best results could be compared to the best scaling application in this paper i.e. pattern search application as both feature a relatively large compute/communication ratio and relatively balanced execution times per data chunk. In terms of speed-ups on the same 16 nodes (the same lab), we can see the speed-up exceeding 14 for 16 nodes in this paper versus approx. 12 for the KernelHive system for the best data partitioning configuration.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

This is expected due to the aforementioned software stacks and at the same time allows to assess the proposed CUDA+OpenMP+MPI framework favorably, in terms of scalability.

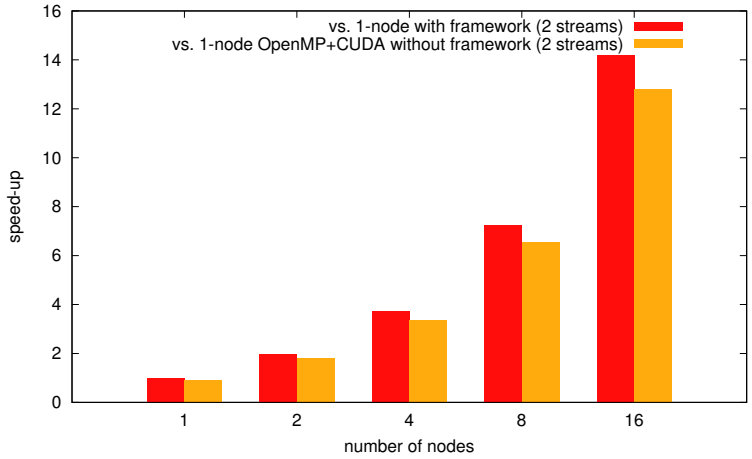

Figure 17. Pattern search application, comparison of speed-up vs number of nodes, vs 1 node using the framework or vs 1 node using OpenMP+CUDA only, 2 streams per GPU, testbed 1

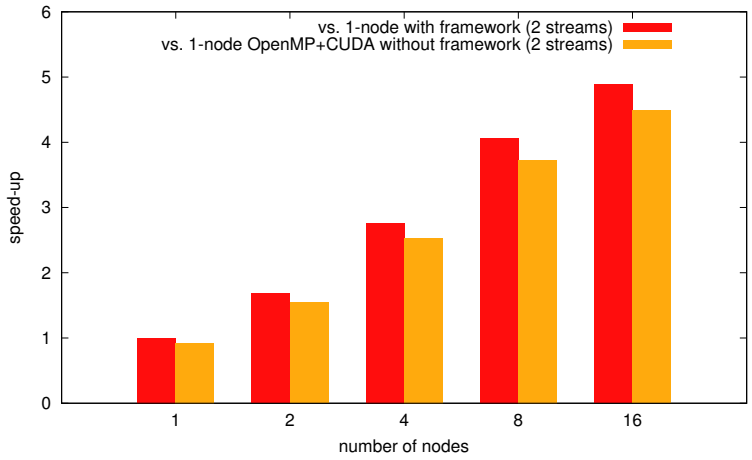

Figure 18. Collatz application, comparison of speed-up vs number of nodes, vs 1 node using the framework or vs 1 node using OpenMP+CUDA only, 2 streams per GPU, testbed 1

5. SUMMARY AND FUTURE WORK

In the paper, we have proposed an API and implementation of combined OpenMP+CUDA framework for a GPU cluster that hides internode communication behind CUDA APIs by using MPI, specifically taking advantage of the `MPI_THREAD_MULTIPLE` mode for multithreaded communication and management of GPUs as well as multiple CUDA streams. We have shown scalability of the solution for 3 workloads including finding the largest divisors, checking the Collatz

DOI: 10.1002/cpe

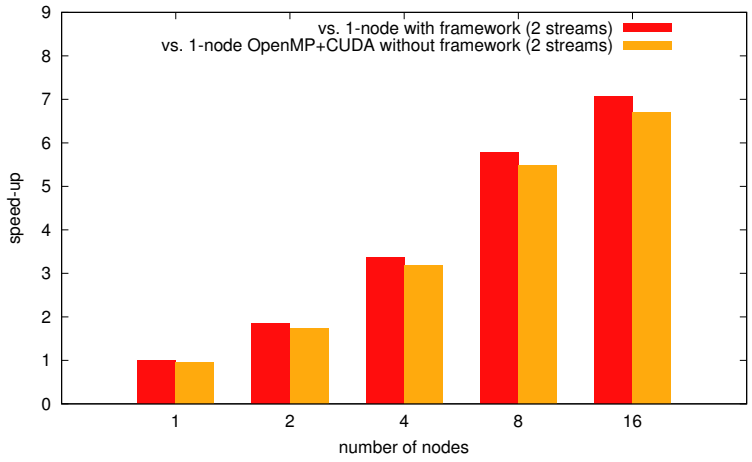

Figure 19. Vecmaxdiv application, comparison of speed-up vs number of nodes, vs 1 node using the framework or vs 1 node using OpenMP+CUDA only, 2 streams per GPU, testbed 1

conjecture and finding patterns in vectors, tested on 3 various systems: a cluster of 16 nodes with NVIDIA GTX 1060 GPUs; a system with 2 powerful nodes, one with: 8x NVIDIA Quadro RTX 6000 GPUs, second with 4x NVIDIA Quadro RTX 5000 GPUs; a heterogeneous environment with several nodes: one with 2x NVIDIA RTX 2080 and 2 nodes with NVIDIA GTX 1060 GPUs. We showed scaling of execution times versus imposed power caps, benefits from using 2 streams per GPU in the framework. We further showed that the speed-ups of the framework, compared to a theoretically ideal implementation is within 89.3% to 97.4% of the theoretically ideal ones for 16 nodes. Consequently, we have confirmed that the framework is a viable and efficient solution to the problem of dynamic power-aware load balancing in a multi-node GPU environment.

In the future, benchmarking and extending the solution towards other types of GPUs will be of interest, specifically mobile and embedded type of compute devices in multi-node systems. It has been demonstrated that such devices offer less power demanding computing [40, 41] and exploration of performance-power trade-offs using power capping might result in non-trivial configurations under more strict power limitations as compared to powerful GPUs in the traditional servers and HPC systems. Additionally, extending the implementation to use available CPU cores for computations will be performed, according to the concept presented in Appendix E.

References

1. Czarnul P. *Parallel Programming for Modern High Performance Computing Systems*. Chapman and Hall/CRC Press/Taylor & Francis, 2018. URL https://www.routledge.com/Parallel-Programming-for-Modern-High-Performance-Computing-Systems/Czarnul/p/book/9781138305953.
2. Tian S, Chesterfield J, Doerfert J, Chapman B. Experience report: Writing a portable gpu runtime with openmp 5.1. *OpenMP: Enabling Massive Node-Level Parallelism*, McIntosh-Smith S, de Supinski BR, Klinkenberg J (eds.), Springer International Publishing: Cham, 2021; 159–169.
3. Yang CT, Huang CL, Lin CF. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications* 2011; **182**(1):266–269, doi:https://doi.org/10.1016/j.cpc.2010.06.035. URL https://www.sciencedirect.com/science/article/pii/S0010465510002262,

computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009.
4. Czarnul P, Proficz J, Drypczewski K. Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. *Sci. Program.* 2020; **2020**:4176 794:1–4176 794:19, doi:10.1155/2020/4176794. URL https://doi.org/10.1155/2020/4176794.
5. Kocot B, Czarnul P, Proficz J. Energy-aware scheduling for high-performance computing systems: A survey. *Energies* 2023; **16**(2), doi:10.3390/en16020890. URL https://www.mdpi.com/1996-1073/16/2/890.
6. Borghesi A, Bartolini A, Lombardi M, Milano M, Benini L. Scheduling-based power capping in high performance computing systems. *Sustainable Computing: Informatics and Systems* 2018; **19**:1–13, doi:https://doi.org/10.1016/j.suscom.2018.05.007. URL https://www.sciencedirect.com/science/article/pii/S2210537917302317.
7. Chiesi M, Vanzolini L, Mucci C, Franchi Scarselli E, Guerrieri R. Power-aware job scheduling on heterogeneous multicore architectures. *IEEE Transactions on Parallel and Distributed Systems* 2015; **26**(3):868–877, doi:10.1109/TPDS.2014.2315203.
8. Czarnul P, Rościszewski P. Optimization of execution time under power consumption constraints in a heterogeneous parallel system with gpus and cpus. *Distributed Computing and Networking*, Chatterjee M, Cao Jn, Kothapalli K, Rajsbaum S (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2014; 66–80.
9. Bratek P, Szustak L, Wyrzykowski R, Olas T, Chmiel T. Heterogeneous voltage frequency scaling of data-parallel applications for energy saving on homogeneous multicore platforms. *Euro-Par 2021: Parallel Processing Workshops*, Chaves R, B Heras D, Ilic A, Unat D, Badia RM, Bracciali A, Diehl P, Dubey A, Sangyoon O, L Scott S, *et al.* (eds.), Springer International Publishing: Cham, 2022; 141–153.
10. Khaleghzadeh H, Fahad M, Shahid A, Manumachu RR, Lastovetsky A. Bi-objective optimization of data-parallel applications on heterogeneous hpc platforms for performance and energy through workload distribution. *IEEE Transactions on Parallel and Distributed Systems* 2021; **32**(3):543–560, doi:10.1109/TPDS.2020.3027338.
11. Krzywaniak A, Czarnul P, Proficz J. Extended investigation of performance-energy trade-offs under power capping in hpc environments. *2019 International Conference on High Performance Computing Simulation (HPCS)*, 2019; 440–447, doi:10.1109/HPCS48598.2019.9188149.
12. Krzywaniak A, Czarnul P, Proficz J. Gpu power capping for energy-performance trade-offs in training of deep convolutional neural networks for image recognition. *2022 International Conference on Computational Science (ICCS)*, 2022. Accepted, in press.
13. Krzywaniak A, Czarnul P. Performance/energy aware optimization of parallel applications on gpus under power capping. *Parallel Processing and Applied Mathematics*, Wyrzykowski R, Deelman E, Dongarra J, Karczewski K (eds.), Springer International Publishing: Cham, 2020; 123–133.
14. Choi J, Fink Z, White S, Bhat N, Richards DF, Kale LV. Accelerating communication for parallel programming models on gpu systems. *Parallel Computing* 2022; **113**:102 969, doi:https://doi.org/10.1016/j.parco.2022.102969. URL https://www.sciencedirect.com/science/article/pii/S0167819122000606.
15. Potluri S, Wang H, Bureddy D, Singh A, Rosales C, Panda DK. Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012; 1848–1857, doi:10.1109/IPDPSW.2012.228.
16. De Wael M, Marr S, De Fraine B, Van Cutsem T, De Meuter W. Partitioned global address space languages. *ACM Comput. Surv.* may 2015; **47**(4), doi:10.1145/2716320. URL https://doi.org/10.1145/2716320.
17. Nowicki M, Gorski L, Grabrczyk P, Bala P. Pcj - java library for high performance computing in pgas model. *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014; 202–209, doi:10.1109/HPCSim.2014.6903687.
18. Nowicki M, Gorski L, Bala P. Pcj – java library for highly scalable hpc and big data processing. *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018; 12–20, doi:10.1109/HPCS.2018.00017.
19. Nowicki M, Górski L, Bała P. Performance evaluation of java/pcj implementation of parallel algorithms on the cloud. *Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020, Revised Selected Papers*, Springer-Verlag: Berlin, Heidelberg, 2020; 213–224, doi:10.1007/978-3-030-71593-9_17. URL https://doi.org/10.1007/978-3-030-71593-9_17.
20. Sharma V, Chow P. A pgas communication library for heterogeneous clusters 2021, doi:10.48550/ARXIV.2104.12350. URL https://arxiv.org/abs/2104.12350.
21. Kaiser H, Diehl P, Lemoine AS, Lelbach B, Amini P, Berge A, Biddiscombe J, Brandt SR, Gupta N, Heller T, *et al.*. HPX - the C++ standard library for parallelism and concurrency. *J. Open Source Softw.* 2020; **5**(53):2352, doi:10.21105/joss.02352. URL https://doi.org/10.21105/joss.02352.
22. Breitbart J. OpenMP for next generation heterogeneous clusters. USENIX Association: Berkeley, CA, 2010.

23. Rościszewski P, Czarnul P, Lewandowski R, Schally-Kacprzak M. Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. *Concurrency and Computation: Practice and Experience* 2016; **28**(9):2586–2607, doi:https://doi.org/10.1002/cpe.3719. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3719.
24. Jarzabek L, Czarnul P. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *J. Supercomput.* 2017; **73**(12):5378–5401, doi:10.1007/s11227-017-2091-x. URL https://doi.org/10.1007/s11227-017-2091-x.
25. Li H, Liang T, Lin Y. An openmp programming toolkit for hybrid CPU/GPU clusters based on software unified memory. *J. Inf. Sci. Eng.* 2016; **32**(3):517–539. URL http://www.iis.sinica.edu.tw/page/jise/2016/201605_01.html.
26. Lee J, Tran MT, Odajima T, Boku T, Sato M. An extension of xcalablemp pgas lanaguage for multi-node gpu clusters. *Euro-Par 2011: Parallel Processing Workshops*, Alexander M, D'Ambra P, Belloum A, Bosilca G, Cannataro M, Danelutto M, Di Martino B, Gerndt M, Jeannot E, Namyst R, *et al.* (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2012; 429–439.
27. Silla F, Iserte S, Reaño C, Prades J. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience* 2017; **29**(13), doi:10.1002/cpe.4072. URL https://doi.org/10.1002/cpe.4072.
28. Kim J, Seo S, Lee J, Nah J, Jo G, Lee J. Opencl as a programming model for gpu clusters. *Languages and Compilers for Parallel Computing*, Rajopadhye S, Mills Strout M (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2013; 76–90.
29. Liang TY, Li HF, Lin YJ, Chen BS. A distributed ptx virtual machine on hybrid cpu/gpu clusters. *Journal of Systems Architecture* 2016; **62**:63–77, doi:https://doi.org/10.1016/j.sysarc.2015.10.003. URL https://www.sciencedirect.com/science/article/pii/S1383762115001174.
30. Kim J, Jo G, Jung J, Kim J, Lee J. A distributed opencl framework using redundant computation and data replication. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, Association for Computing Machinery: New York, NY, USA, 2016; 553–569, doi:10.1145/2908080.2908094. URL https://doi.org/10.1145/2908080.2908094.
31. Yviquel H, Cruz L, Araujo G. Cluster programming using the openmp accelerator model. *ACM Trans. Archit. Code Optim.* aug 2018; **15**(3), doi:10.1145/3226112. URL https://doi.org/10.1145/3226112.
32. Boiński T, Czarnul P. Optimization of Data Assignment for Parallel Processing in a Hybrid Heterogeneous Environment Using Integer Linear Programming. *The Computer Journal* 02 2021; **65**(6):1412–1433, doi:10.1093/comjnl/bxaa187. URL https://doi.org/10.1093/comjnl/bxaa187.
33. Czarnul P. Investigation of parallel data processing using hybrid high performance CPU + GPU systems and CUDA streams. *Comput. Informatics* 2020; **39**(3):510–536, doi:10.31577/cai_2020_3_510. URL https://doi.org/10.31577/cai_2020_3_510.
34. Prades Gasulla J. Improving performance and energy efficiency of heterogeneous systems with rcuda 2021.
35. Iserte S, Prades J, Reaño C, Silla F. Improving the management efficiency of gpu workloads in data centers through gpu virtualization. *Concurrency and Computation: Practice and Experience* 2021; **33**(2):e5275, doi:https://doi.org/10.1002/cpe.5275. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5275, e5275 cpe.5275.
36. Wang F, Zhang W, Lai S, Hao M, Wang Z. Dynamic GPU energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems* 2022; :1–1doi:10.1109/tpds.2021.3137867. URL https://doi.org/10.1109%2Ftpds.2021.3137867.
37. Kang DK, Lee KB, Kim YC. Cost efficient gpu cluster management for training and inference of deep learning. *Energies* 2022; **15**(2), doi:10.3390/en15020474. URL https://www.mdpi.com/1996-1073/15/2/474.
38. Georgiou Y, Glesser D, Hautreux M, Trystram D. Power adaptive scheduling September 2015. SLURM User Group 2015, https://slurm.schedmd.com/SLUG15/Power_Adaptive_final.pdf.
39. Azimi R, Jing C, Reda S. Powercoord: Power capping coordination for multi-cpu/gpu servers using reinforcement learning. *Sustainable Computing: Informatics and Systems* 2020; **28**:100 412, doi:https://doi.org/10.1016/j.suscom.2020.100412. URL https://www.sciencedirect.com/science/article/pii/S2210537920301396.
40. Daher AW, Rizik A, Muselli M, Chible H, Caviglia DD. Porting rulex software to the raspberry pi for machine learning applications on the edge. *Sensors* 2021; **21**(19), doi:10.3390/s21196526. URL https://www.mdpi.com/1424-8220/21/19/6526.
41. Lapegna M, Balzano W, Meyer N, Romano D. Clustering algorithms on low-power and high-performance devices for edge computing environments. *Sensors* 2021; **21**(16), doi:10.3390/s21165395. URL https://www.mdpi.com/1424-8220/21/16/5395.

A. LIST OF API FUNCTIONS

The following list contains C functions available within the framework along with corresponding descriptions of performed actions.

```
void __cudampi__setglobalpowerlimit(float powerlimit); // sets a global power
    limit for GPU computing devices
float __cudampi__gettotalpowerofselecteddevices(); // gets total power of
    currently enabled devices
int __cudampi__selectdevicesforpowerlimit_greedy() { // adopts a greedy
    strategy for selecting devices
// returns 1 if successful, 0 otherwise - if not all devices have been recorded
     power
int __cudampi__getnextchunkindex(long long *globalcounter,int batchsize,long
    long max); // wrapper for
// __cudampi__getnextchunkindex_enableddevices(globalcounter,batchsize,max);
int __cudampi__getnextchunkindex_enableddevices(long long *globalcounter,int
    batchsize,long long max);
  // for a given thread (GPU) return the next available data chunk - enabled
      devices considered
  // max is the vector size
int __cudampi__getnextchunkindex_alldevices(long long *globalcounter,int
    batchsize,long long max);
  // for a given thread (GPU) return the next available data chunk - all
      devices considered
  // max is the vector size
int __cudampi__isdeviceenabled(int deviceid); // returns whether the device has
     been enabled (1) or not (0)
cudaError_t __cudampi__cudaGetDeviceCount(int *count); // wrapper for
  // __cudampi__getCUDAdevicescount(count);
void __cudampi__getCUDAdevicescount(int *cudadevicescount); // returns how many
     GPUs in total are available (on all considered nodes)
void __cudampi__initializeMPI(int argc,char **argv); // initialize the multi-
    node MPI environment, arguments are then
// passed to  MPI_Init_thread with MPI_THREAD_MULTIPLE
void __cudampi__terminateMPI(); // terminate processing, shut down the multi-
    node MPI environment
int __cudampi__gettargetGPU(int device); // gets target GPU id on a given node
    for device number (global)
int __cudampi__gettargetMPIrank(int device); // gets target MPI rank based on
    local GPU id
cudaError_t __cudampi__cudaMalloc(void **devPtr, size_t size); // allocate
    space of a given size
// for current device (locally or remotely depending on location of the device)
cudaError_t __cudampi__cudaFree(void *devPtr); // free memory at given pointer
// for current device (locally or remotely depending on location of the device)
cudaError_t __cudampi__cudaDeviceSynchronize(void); // invoke
    cudaDeviceSynchronize()
// on the current device and measure power as well as time
// if the given thread is a manager invoke
    __cudampi__selectdevicesforpowerlimit_greedy()
cudaError_t __cudampi__cudaSetDevice(int device); // set the current device to
    device (argument)
// locally or remotely
```


Prepared using cpeauth.cls
DOI: 10.1002/cpe

```
cudaError_t __cudampi__cudaMemcpy(void *dst, const void *src, size_t count,
enum cudaMemcpyKind kind); // copy data as in cudaMemcpy(dst,src,count,kind)
// execute either locally or remotely (to, from device, using MPI)
cudaError_t __cudampi__cudaMemcpyAsync(void *dst, const void *src, size_t count
    ,
enum cudaMemcpyKind kind,cudaStream_t stream); // launch asynchronous copying
    as in
// cudaMemcpyAsync(dst,src,count,kind,stream);
// execute either locally or remotely (to, from device, using MPI)
void launchkernelinstream(void *devPtr,cudaStream_t stream); // function
    executing
// a kernel, allows to configure the grid (dimensions, sizes of thread blocks
    and the grid)
void __cudampi__kernelinstream(void *devPtr,cudaStream_t stream); // function
    for kernel launch
// from application code, launches the kernel invoked in launchkernelinstream
    (...) in
// the given stream
void launchkernel(void *devPtr); // wrapper calling launchkernelinstream(...)
    for the default (0) stream
void __cudampi__kernel(void *devPtr); // function for kernel launch from
    application code
// launches the kernel invoked in launchkernel(...) in the default stream
cudaError_t __cudampi__cudaStreamCreate (cudaStream_t *pStream); // create a
    new stream
cudaError_t __cudampi__cudaStreamDestroy(cudaStream_t stream); // destroy the
    given stream
```

B. KERNEL INVOCATION DETAILS

Each kernel is invoked from the application source code with a call to: `void __cudampi__kernelinstream(void *devPtr,cudaStream_t stream)` or `void __cudampi__kernel(void *devPtr)` that launch functions: `void launchkernelinstream(void *devPtr,cudaStream_t stream)` or `void launchkernel(void *devPtr)` (stream 0) respectively. The latter two invoke the kernel body provided by the programmer in function `void appkernel(void *devPtr)`. Details of these functions are as follows:

```
extern "C" void launchkernelinstream(void *devPtr,cudaStream_t stream) {
  dim3 blocksingrid(gridsize);
  dim3 threadsinblock(blocksize);

  appkernel<<<blocksingrid,threadsinblock,0,stream>>>(devPtr);
  if (cudaSuccess!=cudaGetLastError())
    printf("Error during kernel launch in stream");
}
```

and

```
__global__
void appkernel(void *devPtr) {
```


DOI: 10.1002/cpe

```
...
}
```

C. CODE STRUCTURE AND DEPENDENCIES

As outlined in Figure A1, the solution has a modular architecture with the following key source files:

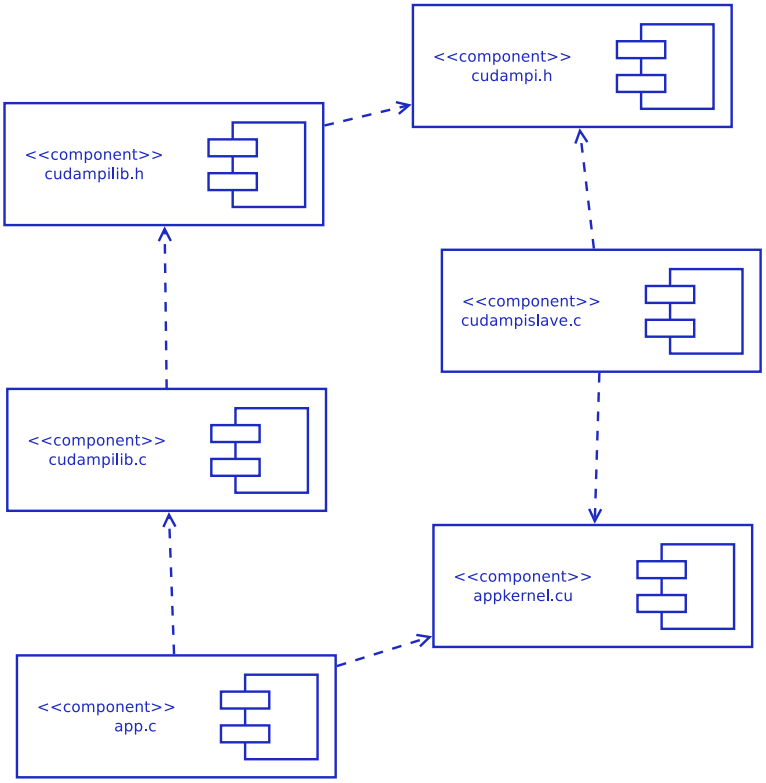

Figure A1. Source component diagram

`cudampi.h` – constants corresponding to labels (MPI tags) of messages exchanged between process 0 (the master) and the other slave/worker processes,

`cudampilib.h` – declaration of functions and constants used exposed by `cudampilib.c` and used by the implementation of process 0 of the MPI application – master/manager handling communication with the other processes but also launching computations on local GPUs,

`cudampilib.c` – library of functions used by the implementation of process 0,

`appkernel.cu` – implementation of a kernel with the syntax as well as grid configuration in the code actually launching a kernel on a GPU described in Appendix B,

`app.c` – code of the actual application with the multithreaded structure according to the model described in Section 3.1,

DOI: 10.1002/cpe

`cudampislave.c` – implementation of MPI processes with ranks larger than 0 running on other nodes with GPUs.

D. PROFILES OF THE TESTED WORKLOADS

Figures A2, A3 and A4 depict processing times of 200 successive data packets from testbed 1, for each application, respectively. These times include communication for non-local GPUs. Shortest observed times correspond to data packets processed locally.

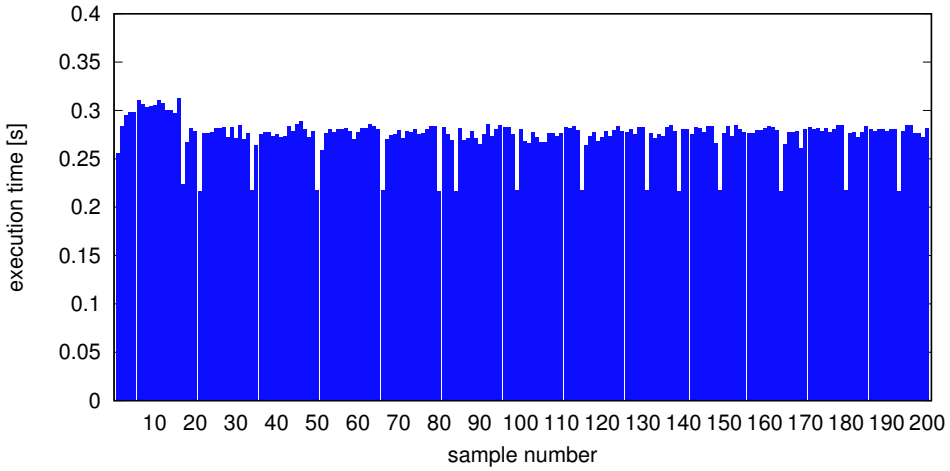

Figure A2. Pattern search application – data packet processing times, 16 nodes, testbed 1

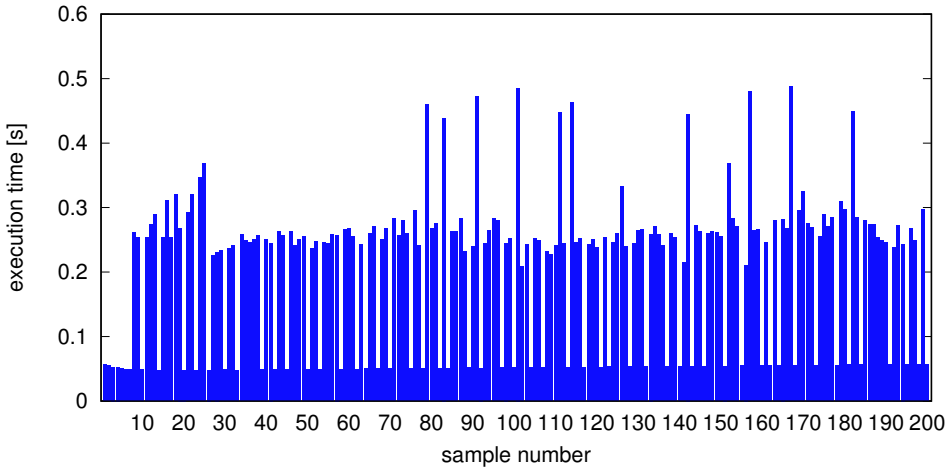

Figure A3. Collatz application – data packet processing times, 16 nodes, testbed 1

Prepared using cpeauth.cls
DOI: 10.1002/cpe

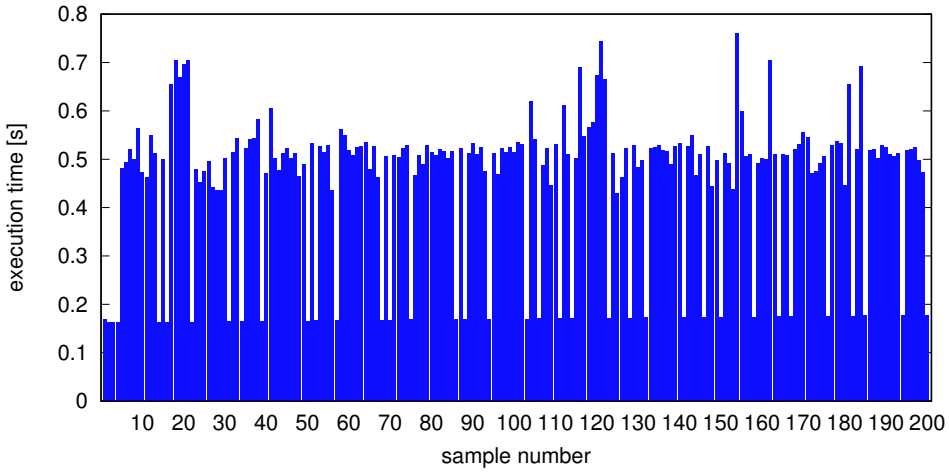

Figure A4. Vecmaxdiv application – data packet processing times, 16 nodes, testbed 1

E. SUPPORT FOR HETEROGENEOUS ENVIRONMENTS

It shall be noted that the proposed model and the implementation support load balancing, assuming the number of data chunks is sufficiently large compared to the number of compute devices, among GPUs in a heterogeneous environment i.e. various numbers of GPUs per node and various GPUs across the system, as shown by the results for testbed 3. This is possible because threads managing various GPUs can fetch subsequent data chunks and schedule kernel calls while other GPUs are active.

While the current implementation is meant for a GPU cluster in which computations are assumed to be performed by the GPUs and the CPUs are used for GPU management, it is possible to extend the implementation to benefit from the available CPU cores for computations as well. Specifically, within MPI's process 0 additional OpenMP threads can be launched for managing CPU cores on particular nodes. Similarly to the current implementation with GPUs, where there is one thread responsible for management of a GPU, one additional OpenMP thread shall be responsible for managing CPU cores (physical, logical) on a particular node (either local or remote). Further parallelization on the target CPU's end, reached with MPI messages, can be performed among the CPU cores with OpenMP nested parallelism and data can be partitioned among the threads in a way analogous to how it is implemented for a GPU. Instead of NVIDIA NVML, power readings can be done using Intel RAPL for Intel CPUs or more universally via PAPI. At the level of the framework, power readings can be considered in a way analogous to those of the GPUs for either engaging or not engaging the particular CPU cores on a given node, as for the GPUs. Similarly, performance/power ratios for CPUs would be considered as in Listing 2.

DOI: 10.1002/cpe