

The Minos Computing Library: Efficient Parallel Programming for Extremely Heterogeneous Systems

Roberto Gioiosa
Pacific Northwest National Lab
Richland, WA, US
roberto.gioiosa@pnnl.gov

Jeffrey S. Vetter
Oak Ridge National Lab
Oak Ridge, TN, US
vetter@ornl.gov

Burcu O. Mutlu
Pacific Northwest National Lab
Richland, WA, US
burcu.mutlu@pnnl.gov

Giulio Picierro
University of Rome Tor Vergata
Rome, Italy
giulio.picierro@uniroma2.it

Seyong Lee
Oak Ridge National Lab
Oak Ridge, TN, US
lees2@ornl.gov

Marco Cesati
University of Rome Tor Vergata
Rome, Italy
cesati@uniroma2.it

Abstract

Hardware specialization has become the silver bullet to achieve efficient high performance, from Systems-on-Chip systems, where hardware specialization can be “extreme”, to large-scale HPC systems. As the complexity of the systems increases, so does the complexity of programming such architectures in a portable way.

This work introduces the Minos Computing Library (MCL), as system software, programming model, and programming model runtime that facilitate programming extremely heterogeneous systems. MCL supports the execution of several multi-threaded applications within the same compute node, performs asynchronous execution of application tasks, efficiently balances computation across hardware resources, and provides performance portability.

We show that code developed on a personal desktop automatically scales up to fully utilize powerful workstations with 8 GPUs and down to power-efficient embedded systems. MCL provides up to 17.5x speedup over OpenCL on NVIDIA DGX-1 systems and up to 1.88x speedup on single-GPU systems. In multi-application workloads, MCL’s dynamic resource allocation provides up to 2.43x performance improvement over manual, static resources allocation.

CCS Concepts • **Computer systems organization** → **Parallel architectures**; • **Computing methodologies** → **Parallel programming languages**;

Keywords Heterogeneous systems, system software, task-based runtime, GPU, asynchronous runtime

ACM Reference Format:

Roberto Gioiosa, Burcu O. Mutlu, Seyong Lee, Jeffrey S. Vetter, Giulio Picierro, and Marco Cesati. 2020. The Minos Computing Library: Efficient Parallel Programming for Extremely Heterogeneous Systems. In *General Purpose Processing Using GPU (GPGPU '20)*, February 23, 2020, San Diego, CA, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3366428.3380770>

1 Introduction

The recent slowdown of growth in realized serial and multi-core performance of commodity microprocessors has forced vendors and users to consider more specialized architectures, including

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
GPGPU '20, February 23, 2020, San Diego, CA, USA

GPUs, FPGAs, and systems-on-a-chip (SoCs) [3, 9, 17]. Experts predict that this trend will continue into the foreseeable future [27], with even more specialized architectures, such as machine learning, neuromorphic, and quantum accelerators. In fact, the extent of heterogeneity in other areas, such as embedded systems and phones, can be extreme: SoCs include dozens of heterogeneous components. High-performance computing (HPC) systems have quickly evolved from systems with one accelerator (e.g., Cray Titan) to systems with multiple devices (e.g., IBM Summit). Although supercomputers are evolving in many areas, most of the complexity in next-generation large-scale installations is expected within a single node.

This “extreme” heterogeneity brings many challenges in terms of productivity, portability, and performance for system software, programming systems, and applications. In order to address this complexity, we hypothesize that new programming abstractions and runtime systems must be developed to program and execute applications on these heterogeneous systems. The programming abstractions must be rich enough to represent important application control flows and data structures, while hiding architecture complexity and facilitating performance portability. Meanwhile, the runtime system must provide a new level of support for discovering and scheduling heterogeneous resources that sometimes offer a performance difference of an order of magnitude or more for a given computation. Poor scheduling decisions at runtime on heterogeneous platforms may devastate overall performance.

The abundance of specialized hardware resources also brings new opportunities. For example, multiple applications can be co-scheduled within a single node (e.g., scientific simulation and in-situ data analytics), reducing the cost of moving data to/from permanent storage before the next stage of the workflows. The same supercomputer can be used to efficiently execute applications from different domains (e.g., scientific simulations, data analytics, or machine learning), amortizing the acquisition cost of a large system. New runtime systems and programming models need to be designed to take advantage of such new opportunities. Very often the specific hardware resource to utilize is hard-coded in the program source (e.g., platform 0, device 0 to indicate the first and only GPU). Scaling these applications to compute nodes with multiple devices requires manual modification of the code. Even worst is the case of co-scheduling applications independently developed by different users. Coordinating access to hardware resources between such applications is extremely problematic, if not unrealistic.

In this paper, we introduce the Minos Computing Library (MCL), a novel runtime system aimed at providing a new level of capability for efficient parallel programming of extremely heterogeneous systems. MCL consists of a node-level scheduler, a programming model, and a programming model runtime. The MCL scheduler orchestrates and coordinates access to computing resources from multiple, concurrent applications. The programming model provides a task-based programming abstraction that simplifies parallel programming and hides low-level architectural details. The MCL runtime supports asynchronous execution of tasks submitted by threads within an application and by multiple applications concurrently running in the system. MCL increases performance portability by transparently scaling applications to systems that feature heterogeneous resources and by enabling programmers to develop code on personal desktop computer and execute on large workstations or HPC compute nodes. MCL is designed to manage and schedule CPUs, GPUs, FPGAs, and SoCs efficiently within a compute node. Our proposal provides scalability and high throughput for co-scheduled applications at a high level of abstraction while respecting dependencies among tasks. Our results show that applications can efficiently scale from a desktop AMD GPU to a DGX-1 workstation with 8 NVIDIA Volta GPUs, automatically providing up to 17x speedup over single-GPU without any code modification. We envision MCL as a back-end of “domain-specific” languages, such as OpenMP [8] or TensorFlow [1]. When paired with a distributed programming model, such as MPI, MCL offers a complete solution for programming heterogeneous supercomputers.

The goal of this work is to introduce MCL design, key ideas, and programming model and to demonstrate MCL portability across a wide range of systems and architectures. We perform a detailed sensitivity analysis of representative benchmarks in various execution scenarios across the tested systems. Specifically, we make the following contributions:

1. describe MCL’s design goals and implementation, highlighting the asynchronous programming model, the scheduling framework, and the programming abstraction.
2. demonstrate performance and scalability across multiple platforms, including an NVIDIA DGX-1 workstation and an ARM/GPU embedded system. We show up to 17x performance improvement on the DGX-1 systems and up to 1.8x on a single-GPU compute node compared to OpenCL.
3. show that MCL can fully utilize the hardware resources in the system and provide automatic load balancing.

The rest of this paper is organized as follows: Section 2 describes MCL architectural design and implementation; Section 3 details our hardware and software test-beds; Section 4 demonstrates MCL scalability, flexibility, and performance improvement; Section 5 describes related work; finally, Section 6 concludes this work and suggests future research directions.

2 MCL Design and Implementation

The Minos Computing Library (MCL) is a system software library, a programming model, and a programming model runtime designed for extremely heterogeneous systems, i.e., systems that feature different classes of resources (e.g., GPUs, FPGAs, CPUs) and multiple devices within each class. MCL aims at simplifying programming heterogeneous systems and increasing performance portability while providing high performance and system utilization through

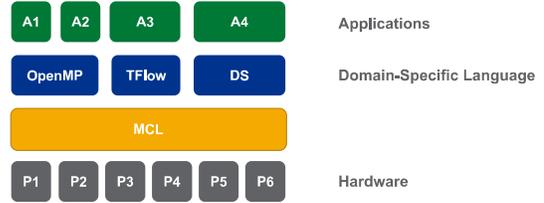


Figure 1. Example of hardware/software stacks based on MCL.

asynchronous execution, intelligent scheduling, and efficient resource allocation. MCL supports multiple applications running concurrently on the same compute node, effectively allocating hardware resources and dispatching tasks to computing devices.

MCL is not meant to replace current programming models used in high-performance computing, data analytics, or machine learning (ML). Rather, MCL is a back-end that enables higher level programming models to efficiently leverage heterogeneous hardware resources. Figure 1 shows examples of possible software stacks. We envision that future heterogeneous systems will have enough hardware resources to accommodate multiple applications co-scheduled in the same compute node. In HPC, this is desirable to minimize the cost of moving data to permanent storage and back. For example, scientific simulations and in-situ data analytics or a ML framework could be co-scheduled to avoid storing simulation results to remote storage and then move them back for data analysis. In other domains, such as autonomous driving, multiple, independent, and competing applications need to be efficiently executed on heterogeneous SoC hardware.

We assume that each application is written independently, potentially using different high-level programming languages. In Figure 1, for example, A1 and A2 are implemented using OpenMP, while A3 is implemented using TensorFlow. In general, we consider the high-level languages above MCL as domain-specific languages (e.g., OpenMP is a language for the HPC domain, while TensorFlow is a language for the ML domain). MCL enables programmers to implement their workflow applications independently using domain languages while the runtime efficiently and transparently manages heterogeneous hardware resources.

MCL consists of several components: a scheduler that orchestrates tasks and manages hardware resources, an asynchronous runtime, a well-defined and compact application programming interface (API), and a set of tools to analyze and debug applications. The scheduler is a persistent, external process that orchestrates task executions from multiple applications. The MCL runtime is a dynamic library linked to each application that implements the MCL APIs and asynchronously executes tasks on behalf the user threads. The MCL runtime is process-safe and thread-safe, thus several multi-threaded applications can co-exist on the same compute node at the same time. The debugger, tracer, and statistic engines are implemented both in the MCL scheduler and runtime and provide information, among others, about the tasks’ execution, resource utilization, and load balancing.

Currently, MCL uses OpenCL [26] as compatibility layer but raises the level of programming abstraction, hides architecture-specific details to programmers, performs automatic load balancing, and supports asynchronous task execution.¹

¹Not all OpenCL functionalities are currently supported in MCL.

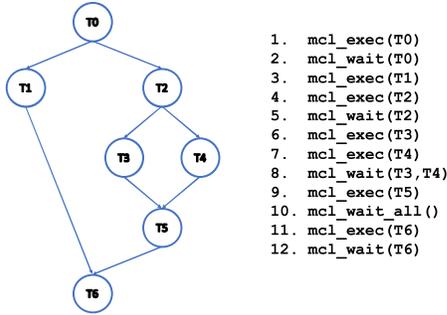


Figure 2. Task dependencies.

2.1 MCL Programming Abstractions

Although we envision MCL to be used as back-end for high-level programming languages, programmers can implement parallel applications directly with the MCL programming model. MCL provides a task-based programming abstraction similar to other task-based programming models, such as Cilk [6], OpenMP [8], OmpSs [7], or StarPU [2]. MCL task programming abstraction is closer to Cilk, in that independent tasks can be spawned recursively, and there are synchronization points that are not necessarily at loop boundaries. However, MCL tasks are coarse-grained while Cilk tasks are fine-grained, in the sense that MCL tasks exploit data parallelism while Cilk tasks are sequential. Also, MCL provides both global/local synchronization primitives, like the `sync` construct in Cilk used to block until all previously spawned have completed, and point-to-point synchronization mechanisms. The latter allows users to express data- and control-flow dependencies among tasks.

MCL task programming abstraction provides programmers with an interface to “implement” an application directed-acyclic-graph (DAG). Figure 2 shows a simple DAG example and the sequence of task spawn (`mcl_exec()`) and point-to-point (`mcl_wait()`) and global (`mcl_wait_all()`) synchronization pseudo-operations. Equivalent non-blocking interfaces (`mcl_test()`) and `mcl_test_all()` are also provided. This simple example shows that a programmer can easily express a dependency between task *T5* and tasks *T3* and *T4*, indicating that task *T5* should not start before *T3* and *T4* have completed. As we will explain in the next section, the MCL runtime executes tasks asynchronously. The `mcl_exec()` functions returns immediately after setting up a task for execution, thus the control flow returns quickly to the caller. Section 4 shows that this approach increases the task injection rate and overall performance.

2.2 MCL Asynchronous Execution Model

Application tasks are executed asynchronously by the MCL runtime. When a task is created, a *task handle* is associated to the task and returned to the user. Programmers can query the status of a task (e.g., `RUNNABLE`, `EXECUTING`, `TERMINATED`, `FAILED`) by examining the corresponding task handle. The handle is also used to implement task dependencies and to set task’s information (Table 1).

When a task is submitted, the MCL runtime sends a scheduling request to the MCL scheduler, as shown in Figure 3. The control returns immediately to the user program, which can submit other tasks or perform other work. The MCL scheduler decides when to execute a task and where it should be dispatched. As explained in the next Section, the MCL programming model does not allow users

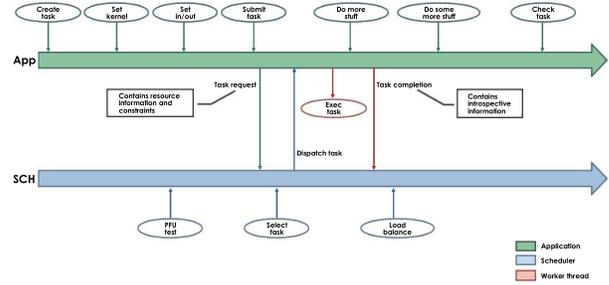


Figure 3. An example of MCL execution flow.

to indicate a specific resource device, such as GPU3, but only a class of devices. Eventually, the scheduler reserves hardware resources and dispatches the task to a specific device. It, then, sends an active message back to the original application. The task is executed *in the context of the user application* by MCL workers, parallel thread created by the MCL runtime at initialization time. MCL workers are in charge of executing active messages, off-loading computation to heterogeneous resources, checking the status of each task, and returning a task execution error code to the MCL scheduler. Executing tasks in the context of the application, rather than in the context of the MCL scheduler, eliminates the necessity of copying data back and forth between application and scheduler.

As Figure 3 shows, task execution is deferred to a later point in time when hardware resources are available and the scheduler has selected the task for execution. Programmers can use the `mcl_wait_all()` and `mcl_wait()` to force the application thread to block until specific tasks have completed their execution.

MCL is a back-end programming model and runtime. It allows programmers and high-level domain languages to express tasks dependencies, but does not automatically determines data or control-flow dependencies. This design choice has been made for two reasons: first, to maintain the low-level runtime slender and efficient; second, because some of the information available to compilers and domain-specific languages are not available at MCL level. We envision that a domain-specific compiler and language can be used to determine task dependencies.

2.3 MCL Scheduling Framework

Designing efficient task scheduling algorithms is difficult and may require domain information. Many scheduling algorithms have been proposed in the literature, from random work stealing in Cilk and static/dynamic task scheduling in OpenMP, to the completely fair scheduler (CFS) used in modern Linux OS kernels. Task scheduling is even more complicated in the context of MCL, which is meant to facilitate porting and co-scheduling applications from different domains, each with its own characteristics and assumptions. Rather than developing a “holistic scheduler” that performs reasonably for all domains, we developed a *scheduling framework* in which developers can plug in their own scheduling algorithms, taking advantages of domain-specific knowledge and requirements. For example, a scheduler for self-driving cars may employ priorities to ensure that critical tasks meet their hard real-time deadlines, while a scheduler for the OpenMP runtime may consider that all tasks have the same priority, reducing design complexity.

Table 1. MCL APIs

Name	Description	Return
<code>mcl_init(nworkers, flags)</code>	Initialize MCL library	Error
<code>mcl_finit()</code>	Finalize MCL library	Error
<code>mcl_task_create()</code>	Create a new task	Handle
<code>mcl_task_set_kernel(hdl, src, name, nargs)</code>	Set task kernel	Error
<code>mcl_task_init(src, name, nargs)</code>	Create and initialize task	Handle
<code>mcl_task_set_arg(hdl, id, addr, size, flags)</code>	Set task argument	Error
<code>mcl_exec(hdl, pes, lpse, flags)</code>	Submit a task for execution	Error
<code>mcl_wait(hdl)</code>	Wait termination of a task	Error
<code>mcl_test(hdl)</code>	Test termination of a task	Status
<code>mcl_wait_all()</code>	Wait termination of all tasks	Error
<code>mcl_hdl_free(hdl)</code>	Release handle	Error
<code>mcl_dev_getn()</code>	Return the number of resources	Error
<code>mcl_dev_info(id, dev)</code>	Provide device characteristics	Error

Developers can implement sophisticated scheduling algorithms that leverage domain knowledge. MCL provides an application binary interface (ABI) which consists of a set of methods that need to be implemented by each scheduling algorithm. These include initialization/finalization of the algorithm data structures, the `next()` method to select the next task to dispatch, and the `put()` method to remove a completed task. Moreover, each algorithm can implement specific data structures, from simple FIFO queues to complex red-black trees, to queue runnable tasks. As an example and without loss of generality, in this work we implemented a FIFO scheduling algorithm under resource constraints. New tasks are queue in a general FIFO queue on the scheduler side. The `next()` method extracts the first task from the list and verifies that there are available resources. The task is scheduled if there are enough resources (memory and processing elements) available, otherwise the next task in the queue is analyzed. Although this algorithm is a best-effort implementation, Section 4 show promising results. Moreover, this best-effort implementation introduces negligible runtime overhead, which allows us to understand all other MCL sources of overhead.

We have taken a similar approach for the load balancing algorithm. In many cases, a two-level, round-robin algorithm works well to balance the load among device classes and among the devices in each class. However, since heterogeneous devices may be very different from one other, more sophisticated algorithms may achieve better performance. For example, one may want to distribute more work to GPU-class devices than CPU-class devices because of the larger number of processing elements, introducing some imbalance between the two classes but potentially achieving higher performance. Scheduling and load balancing frameworks are de-coupled but cooperate to achieve efficient resource management and fair scheduling. In this work, we paired the scheduling algorithm with a two-level round-robin algorithm that balances the load across classes first and then among devices of the same class.

2.4 MCL Programming Interface

MCL provides a compact set of application programming interface (APIs) through which programmers can create, execute, and check the status of a task. Compared to OpenCL, the list of MCL APIs is considerably shorter. This is intentional, as the user is not required (nor allowed) to specify architectural details, such as which GPUs should be used to execute a task. Table 1 shows the MCL API names, a brief description, and the return value. Functions are divided into classes: the first class (`mcl_init()` and `mcl_finit()`) includes utility functions to initialize and finalize MCL. Applications are required to insert these functions to register/de-register with the MCL scheduler before submitting any tasks for execution.

Table 2. Experimental Test-beds

System	Type	PEs	Mem	Dev
NVIDIA Xavier	ARM Cortex A57	8	16	1
	NVIDIA Volta	512	-	1
Apple iMac Pro	Intel Xeon W-2140B	16	32	1
	ATI Radeon P Vega	14,336	8	1
GPU Compute node	Intel Xeon E5-2680	20	768	2
	NVIDIA Tesla P100	3,584	12	8
NVIDIA DGX1 V100	Intel Xeon E5-2698	20	256	2
	NVIDIA Tesla V100	5,120	16	8

The second class consists of functions to setup tasks, while the third class includes functions to submit and monitor the execution of a task. A task is created through the `mcl_task_create()` function, which returns a new and unique task handle. The handle represents the task in the application space. Programmers can query the status of a task, or check whether or not the task has executed correctly, by monitoring the appropriate task handle. `mcl_task_set_kernel()` and `mcl_task_set_arg()` are used to setup the task’s kernel and arguments, respectively. The first function takes the source code, the name of the kernel, and the number of parameters. `mcl_task_set_arg()` takes the address and the size of argument `id`, as well as a set of flags that can be used to indicate whether the argument is an input, an output, an input/output, or whether is a scalar value or a buffer. Scalar values are passed by value to the MCL runtime, while buffers are passed by reference. This avoids expensive memory copies for large buffers between application and library space.

`mcl_exec()` submits a task to the MCL scheduler. Programmers can pass a set of flags that indicate which class of resources they want to use for the execution of the task (e.g., `MCL_CPU`, `MCL_GPU`, `MCL_ANY`). Specifying `MCL_CPU` or `MCL_GPU` poses a strict constraint to the MCL scheduler on the execution of a task, and it is meant for those cases where the programmer is certain that a class of resources is better suited for executing the task. Programmers can specify softer constraints by using `MCL_ANY` with a preference (e.g., `MCL_FAV_GPU`). This indicate to the MCL scheduler that the task can be executed on any resource but it would be better to execute it on a GPU-class resource.

`mcl_exec()` is not blocking and returns immediately after checking possible errors. Programmers can, thus, continue the execution and submit more tasks, perform other computation, or prepare the next input buffers. Note that programmers are not free to release input buffers after a call to `mcl_exec()` and until the task has been completed. In fact, the asynchronous nature of MCL implies that task execution is deferred to an unpredictable time in the future. Users can check the current status of the task by querying the task handle (see Figure 3). `mcl_test()` and `mcl_wait()` check the termination of a task. The difference between the two functions is that `mcl_wait()` blocks until the task has terminated. `mcl_wait_all()` is a variant of `mcl_wait()` that blocks until all submitted tasks have completed. Most API calls return an error code that expresses whether the function has completed correctly or why it has failed.

3 Experimental Environment

This section describes the software and architecture environments used for the experiments presented in the next section. To demonstrate MCL portability and adaptability, we used a variety of different heterogeneous platforms, ranging from an embedded system

all the way up to powerful workstations. Table 2 lists the platforms used in this work and their heterogeneous characteristics.

The Xavier system is a power efficient system equipped with an 8-core ARM processor, a Volta GPU with 512 CUDA cores, and a 16GB of high-bandwidth on-chip memory (HBM) in a single SoC. NVIDIA does not support OpenCL on embedded platforms, hence we used the Portable Computing Library 2 (POCL) version 1.4 [12] with LLVM 8.0 to access the ARM cores as OpenCL devices. MCL and all benchmarks are compiled with GCC 4.8.

The Apple iMac Pro system is a desktop system equipped with one 16-core Intel W-2140B processor and one 14,336-core ATI Radeon GPU. MCL and all the benchmarks are compiled with Apple LLVM compiler version 10.0.0 and linked against the Apple OpenCL library, compliant with OpenCL 1.2.

The NVIDIA DGX-1 V100 is a powerful workstation developed by NVIDIA primary for machine and deep learning workloads. The system consists of two 20-core Intel Xeon E5-2698 processors attached to 256 GB of DRAM main memory. Eight NVIDIA Tesla V100 (Volta) are connected together through an NVLink bus [28]. Each GPU consists of 5,120 CUDA cores connected to 16 GB HBM stack. MCL and all the benchmarks are compiled with GNU GCC 5.4.0 and NVCC from the CUDA 9.1 SDK.

Finally, the GPU compute node is a traditional HPC compute node equipped with an NVIDIA P100 with 12GB of on-board memory and interconnected to Intel x86 processors through PCIe bus. The system features 768 GB of DRAM memory. We use the NVIDIA OpenCL library for the NVIDIA GPUs and POCL version 1.4 for the CPU sockets. POCL is supported by LLVM 8.0.

The tests performed in this paper are based on a double matrix-matrix multiplication (DGEMM), which serves as the basic block of many scientific and ML applications. This test is representative of an application that performs a large matrix-matrix multiplication and divides the work in T tiles of size $N \times N$. Unless otherwise specified, the execution time reported is taken between the time the first task is submitted and the time the last task has been executed (we do not account for the initialization time). The results reported in the next section are the average of ten runs.

The benchmark implementations do not change across platform. The MCL implementations naturally scale on each system and use all available resources. The MCL scheduler is in charge of scheduling tasks within each class of resource, or across all classes if the user has so specified. The OpenCL implementation is a “best-effort” implementation in the sense that kernels are submitted to the device queue as long as there are slots available in the OpenCL queue, after which the benchmark blocks. Buffers are properly managed in order to allow concurrent execution of multiple kernels, and just-in-time (JIT) compilation of the GEMM kernel is cached to reduce runtime execution overhead. However, we have not implemented any load balancing algorithm nor a dynamic scheduler that automatically discovers the available resources and effectively dispatches tasks. Although it is certainly possible to implement a sophisticated OpenCL benchmark, our goal is to compare a benchmark that has comparable programming complexity, and the current implementation of the OpenCL GEMM benchmark is already twice the size of the MCL implementation.

The MCL version is much simpler than its OpenCL counterpart and about half the size. Most of the host code for parsing arguments and computing metrics is shared between the two implementations. Additionally, the MCL version allows the user to specify the number

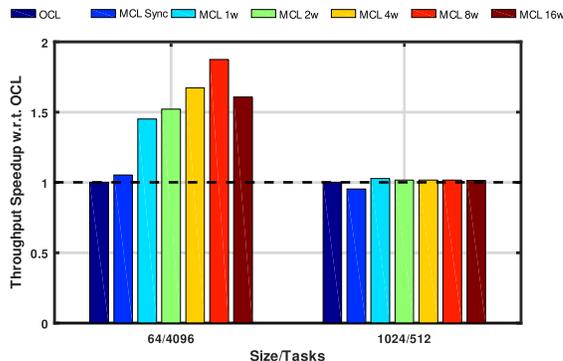


Figure 4. MCL performance compared to equivalent OpenCL implementation on NVIDIA Pascal P100 (higher is better).

of worker threads and whether the execution should be synchronous or asynchronous. Programming in MCL is much easier than OpenCL. There is no need to specify which device should be used when submitting tasks (as opposed to specifying the OpenCL device queue), nor to implement load balancing algorithms and resource discovery. Not only the MCL code is shorter and more readable, it also does not include architecture-specific information on the target device.

4 Experimental Results

This section evaluates MCL performance compared to equivalent OpenCL implementations on a variety of systems and scenarios. We highlight the benefits of using MCL and the MCL programming model and show how MCL scheduler adapts to the available hardware resources and achieves excellent utilization even with small tasks. This is done transparently to the user, which greatly increases program and performance portability. Our goal is to show MCL performance and flexibility across a variety of systems and scenarios. To reduce the number of experiments, we focus on one kernel that serves as the base of many scientific and ML applications. To make a fair comparison, we use the exact same OpenCL kernels code for both the OpenCL and MCL implementation.

MCL Performance Analysis In the first set of experiments, we analyze the overhead introduced by MCL over OpenCL when using similar hardware resources. The goal of this test is to evaluate MCL scheduling overhead, the parallelism exploited by the MCL workers, and the benefits of asynchronous execution. We conduct these tests on the GPU compute node platform, as this system only features one high-performance GPU, thus both OpenCL and MCL will use the same hardware resources.

Figure 4 shows MCL performance with respect to OpenCL (first bar) in terms of throughput (tasks executed/s). In order to highlight runtime overhead we perform tests with small and medium tasks. Runtime overhead is relatively higher with small computational tasks where the amount of computation is not enough to amortize the cost of moving input data to the device, setup the kernel execution, and move output data back to main memory.

We evaluate different scenarios in these experiments. In the first scenario, we configure MCL to execute synchronously and with only 1 worker thread. This means that no new task is submitted until the preceding task has terminated and that MCL cannot leverage the

parallelism provided by additional workers. This scenario restricts MCL's abilities and forces MCL to operate similar to the OpenCL version of the benchmark. As the second bar (MCL Sync) in Figure 4 shows, MCL synchronous performance is close to OpenCL. For larger tasks (1,024x1,024), where the execution time is dominated by computation, MCL performs slightly worse than OpenCL. For small tasks (64x64) where the execution time is dominated by data movement, MCL performs slightly better than OpenCL. This is due to the implicit parallelism introduced by the MCL workers: Even with one worker, the application thread and the MCL worker execute in parallel.

The next five bars on Figure 4 represent scenarios in which MCL executes tasks asynchronously with varying number of worker threads. When employing asynchronous execution, MCL does an excellent job at hiding the latency of runtime scheduling and data movement. More tasks can be dispatched per unit of time, hence, more tasks can be executed concurrently. For small tasks, MCL achieves 1.45x, 1.52x, 1.67x, 1.88x, and 1.61x speedups over OpenCL for 1, 2, 4, 8, and 16 workers, respectively. The graph shows that MCL efficiently hides data movement latency by executing tasks while moving data for the next computations. Hiding data movement latency is extremely important for small tasks, where data movement dominates the execution time. Our results clearly show that OpenCL suffers in this scenario, which means that users have to manually implement efficient mechanisms to overlap computation and data movement in the applications. MCL, instead, handles computation/communication overlap automatically.

For larger tasks, dominated by computation, OpenCL performs better. In this case, there is enough computation to amortize the cost of data movement. Although, the importance of overlapping data movement and computation is lower, MCL performance matches OpenCL, which indicates that the runtime overhead introduces is relatively small. We note that using more than 8 MCL workers, on this system, does not provide additional performance. As the plot shows, MCL performance with 16 workers is lower than with 8 workers. This happens for two reasons: first, 8 workers are sufficient to saturate the system and fully utilize the GPU. Second, in addition to the MCL workers, there is one application thread and two MCL scheduler threads (receiver and scheduler). Using 16 workers induces additional OS context switches among threads.

We remark that, although MCL leverages internal concurrent execution (Figure 3), both the OpenCL and MCL user benchmarks used in these experiments are sequential. While it is certainly possible to implement an OpenCL multi-threaded version, this introduces further complexity when programming heterogeneous systems, as discussed in the next section. MCL, instead, leverage the internal parallelism of the workers and the asynchronous execution while still allowing programmers to reason in terms of a sequential paradigm. On the other hand, MCL is thread-safe and process-safe, thus it is possible to implement a multi-threaded application where each thread submits MCL tasks, or even a multi-processes application.

In summary, the experiments shown in Figure 4 shows that MCL asynchronous execution and internal parallelism achieves considerable speedup (up to 1.88x with a single GPU) over OpenCL, high system utilization, and excellent overall performance.

System Utilization MCL's main objective is achieving full system utilization and efficient use of heterogeneous computing resources. In this section, we demonstrate that MCL indeed achieves

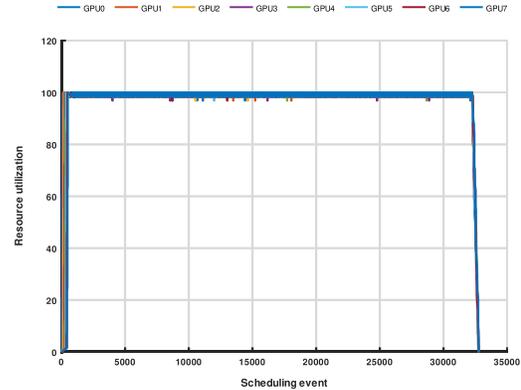


Figure 5. DGX-1 V100 system utilization running 16k tasks of size 1,024x1,024.

these goals. We use the NVIDIA DGX-1 V100 system for this test, as this is the system with the largest number of computing devices and, thus, the most challenging. To collect the necessary data, we configure MCL to provide execution statistics. In particular, the MCL scheduler reports the number of available logical processing elements (PEs) on each device after a scheduling event, i.e., a task is dispatched to a device or a task previously running on a device has terminated.

Figure 5 plots the system utilization as a function of time. The x-axis represents scheduling events while the y-axis indicates utilization. Each line in the plot represents a GPU device on the NVIDIA DGX-1 V100. The plot shows that, as more tasks as submitted by the application, the utilization of each resource increases until all PEs on each device have been allocated and the tasks are being executed. Notice that all the computing devices are fully utilized, as MCL effectively balances the load across the devices. As tasks terminate, new runnable tasks are dispatched to the devices with available PEs until there are no more runnable tasks.

We observed full resource utilization on the other systems as well, though the non-DGX1 systems only feature one GPU, thus there is no need to perform any load balancing. In the interest of space, we only report the most significant case.

Application Performance The graph in Figure 6 compares MCL and OpenCL performance in terms of throughput (tasks/s) on the systems described in Table 2. For the evaluation we used the GEMM application described in the previous section with matrix size 1,024x1,024 (Figure 6) and 64x64 (Figure 7), varying the total number of tasks. Both MCL and OpenCL versions submit tasks to GPU-class devices. The execution time is taken between the submission of the first task the time the last task has been executed.

The red and blue lines in Figure 6 show OpenCL and MCL throughput, respectively, as the number of tasks increases. The black lines in the plots show the achieved speedup (right y-axis) of MCL over OpenCL for a certain number of tasks. The plot in Figure 6a shows that the performance of the OpenCL version flattens already for 64 at 67 tasks/sec for the NVIDIA DGX-1 V100 system. This is because all hardware resources on the first GPU are used and the OpenCL queue is full. As tasks complete, the OpenCL runtime submits more work to the GPU, keeping its utilization around 100% throughout the execution of the application. The MCL

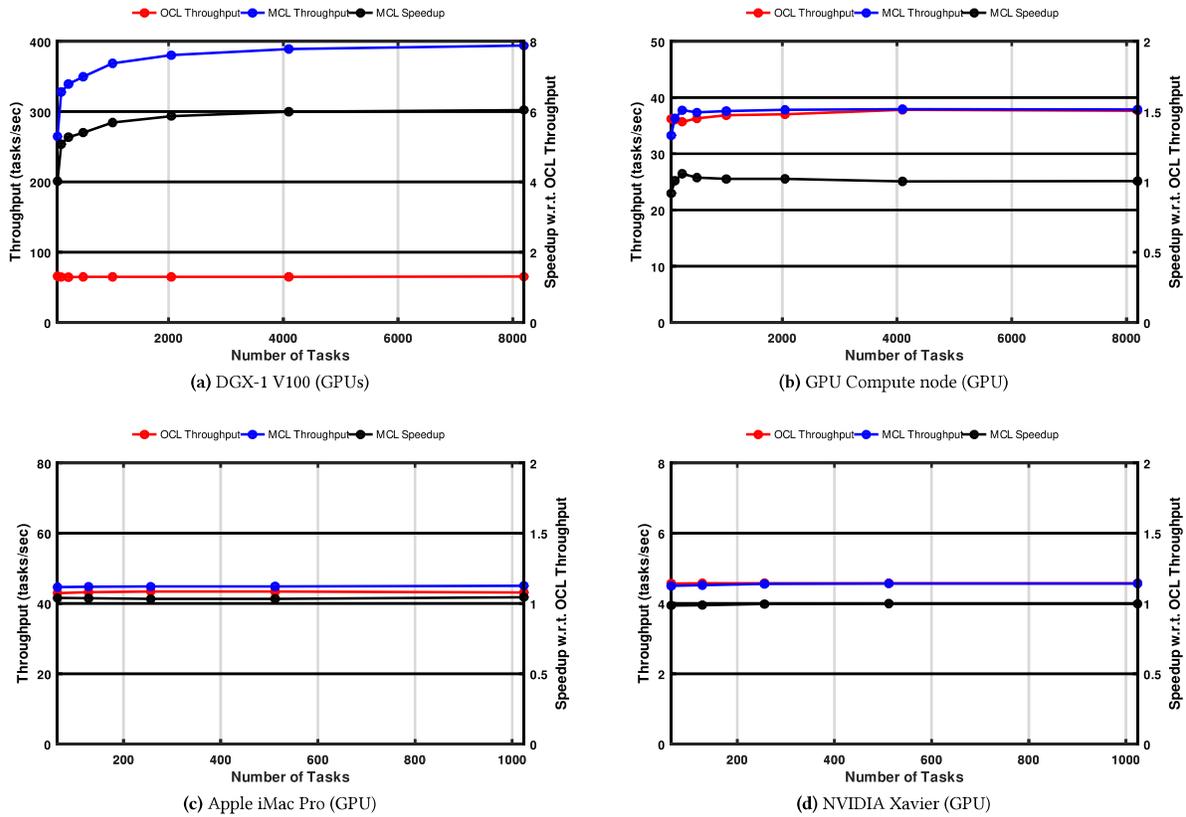


Figure 6. MCL throughput (tasks/sec, left y-axis) when executing GEMM tasks with size 1,024x1,024 and varying number of tasks. The right y-axis reports MCL speedup over OpenCL. In both cases, higher is better.

performance, instead, increases thanks to the ability of the MCL scheduler to distribute tasks across *all* available GPU devices. Performance eventually flattens for the MCL version as well when all GPUs are fully utilized (around 4k tasks) but the achieved throughput is much higher (about 400 tasks/sec) compared to the OpenCL version. MCL achieves 6x speedup over OpenCL when submitting 8k tasks on the NVIDIA DGX-1 V100 system. We remark that this is the same MCL code that runs on a laptop or desktop and that we have made no additional effort to leverage the eight NVIDIA V100 GPUs available in this system. The plot in Figure 6 shows another interesting point: MCL is able to achieve good performance with a relative small number of tasks submitted to the system. For example, Figure 6a shows that MCL achieves an average of 368.52 tasks/s (5.40x speedup over OpenCL) already with 512 tasks, which on a system with eight GPUs means an average of 64 tasks executed per GPU. Combined with the result presented in Figure 4, this excellent performance at small scale demonstrates that 1) MCL runtime overhead is contained and can be amortized by asynchronous execution, 2) current MCL scheduler and load balancing algorithms already provide good performance and load balancing, and 3) MCL is capable to achieve excellent performance even when the ratio computation/communication is low, which is a traditionally difficult point to optimize when running on heterogeneous devices.

Similar results are observed on the other systems. On the Apple iMac desktop (Figure 6c), the Xavier embedded system (Figure 6d),

and cluster compute node (Figure 6b) (all systems feature only one GPU), MCL performs similarly to OpenCL, with MCL running a little faster than OpenCL on the desktop (1.04x speedup). In all cases, the computation is large enough to saturate the hardware resources even with 64 tasks.

The experiments in Figure 6 are conducted with large computational tasks (1,024x1,024 GEMM). With large kernels the overhead of moving data to/from a device and setting up a kernel for execution can be amortized by the length of the computation. However, when the computation is small, such overheads cannot be easily amortized. Small tasks are common in irregular and/or sparse computations and pose important challenges. Figure 7 shows the same experiments performed in the previous plots but with 64x64 GEMM tasks. The results show that, as the computation becomes memory-bound, MCL greatly outperforms OpenCL. On the compute cluster node, MCL achieves up to 1.88x speedup compared to OpenCL (Figure 7b) with 1,024 tasks, before slowing down until 1.73x with 4096 tasks. The results on the Figure 7a show how important is to properly manage data transfer, computation/communication overlap, task scheduling, and load balancing on heterogeneous systems with multiple devices. On the 8-GPUs DGX-1 V100, MCL achieves 17.75x speedup over OpenCL, reaching up to 17,410.02 tasks/s compared to 980.68 tasks/s achieved by the equivalent OpenCL implementation. This super-linear speedup shows the effectiveness of MCL

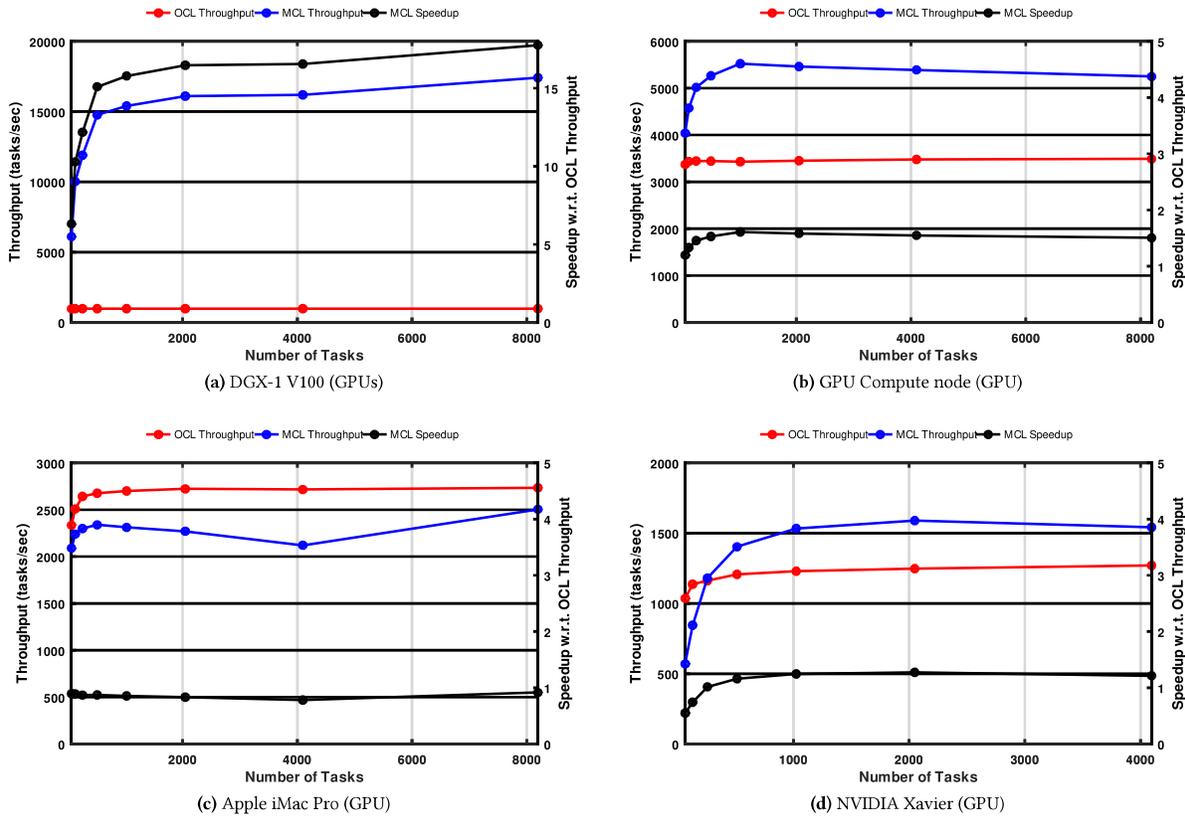


Figure 7. MCL throughput (tasks/sec, left y-axis) when executing GEMM tasks with size 64x64 and varying number of tasks. The right y-axis reports MCL speedup over OpenCL. In both cases, higher is better.

asynchronous execution and load balancing and their impact on overall performance.

The other two single-GPU systems (Figure 7c and Figure 7d) do not show similar results. Although on the Xavier system MCL and OpenCL perform similarly, this value is far from what observed on the other systems. A single ARM core does not provide sufficient ingestion rate for MCL to completely hide data transfer latency and fully utilize the GPU. On the desktop system, instead, MCL performs worse than OpenCL. We have analyzed MCL internal execution events and traces and realized that the Apple OpenCL libraries is more sophisticated than the other ones used and already performs some of the optimizations implementd in MCL.

Overall, the results in Figures 6 and 7 show that MCL does not suffer large slowdown on systems with one GPU compared to OpenCL implementations thanks to its asynchronous task engine. On multi-GPU systems, MCL manages the architecture-specific features, thus user code automatically scales to use all available resources and automatically provides performance improvements, without the need of modifying the existing source code.

Multi-application workload Both MCL and OpenCL support multi-application workload execution, i.e., multiple independent processes sharing the same pool of hardware devices. However, OpenCL code must explicitly be designed to avoid hardware contention with other applications, for example if all applications attempt to execute work on device 0. We designed a multi-application

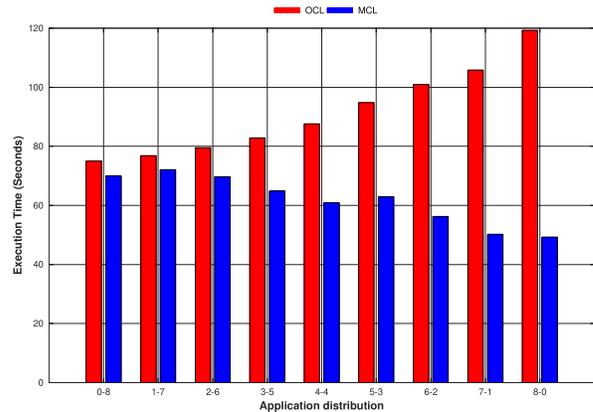


Figure 8. Multi-application workload overall execution time (lower is better): 8 processes concurrently submit tasks of different sizes.

workload experiment in which multiple, independent processes submit tasks of different sizes. Figure 8 shows the results obtained on the NVIDIA DGX1 V100 system (8 GPUs) when executing eight processes in parallel. In this experiment there are two type of processes: small-task processes submit 8,192 64x64 GEMM computations while large-task processes submit 512 1,024x1,024 GEMM

computations. We have timed the number of tasks/process so that the single-process execution takes about the same time. The pairs on the x-axis in Figure 8 represent the number of small- and large-task processes, respectively, e.g., 2-6 indicates that there were 2 processes submitting small computations and 6 processes submitting large tasks. For the OpenCL case, each process is statically associated to a separate GPU. This means that for the cases 0-8 and 8-0, the load for the OpenCL experiment is perfectly balanced across the available GPUs. All processes are submitted at the same time and the the y-axis in Figure 8 reports the total execution time of the experiments, i.e., the execution time of the last process.

The plot in Figure 8 shows several interesting points. First, the overall execution time of the OpenCL case increases when increasing the number of processes that submit small tasks. We have performed several additional experiments (not shown here for brevity) and verified that the contention on the NVLink bus increases significantly with a higher tasks ingestion rate. This is, obviously, more evident with a larger number of processes that submit small computations, as the number of tasks submitted per second increases. Second, we notice that OpenCL and MCL perform similarly for the 0-8 case (all large-tasks processes), which is consistent with what shown in Figure 6 for single-GPU systems. Third, overall, MCL performs better than OpenCL and, in some case, MCL outperforms OpenCL by a factor of 2.43x (case 7-1). For the case 8-0 (all small-task processes), MCL outperforms OpenCL by a factor of 2.71x, which is consistent with the results shown in Figure 7. There are two main reasons that contribute to this performance improvement. MCL asynchronous execution enables better communication/computation overlap, resulting in higher performance especially when executing small tasks (see Figure 4). Moreover, MCL scheduler dynamically allocates computing resources on demand and performs automatic load balancing. This means that 1) if a process has terminated its execution, that GPU can be used to execute tasks submitted by other processes and 2) MCL scheduler may decide to co-schedule tasks submitted by multiple processes on the same computing resource, possibly leaving enough room on other GPUs for the execution of larger tasks that, otherwise, might have to be queued. Finally, we notice that, to the contrary of the OpenCL case, MCL suffers less from resource contention on the NVLink bus. As we are conducting experiments on the same hardware platform, contention on the bus still occurs, but MCL does a better job at hiding it and removing it from the critical path.

Summary and Performance Portability The experiments in this section demonstrate that MCL can efficiently leverage hardware devices in heterogeneous systems, balance the load across such devices, and run on a variety of very different systems, from small, power-efficient embedded systems, to powerful distributed workstation. While it is always difficult to quantify portability and we do not offer a precise metric in this work, we highlight that in our development model we write MCL applications on a personal desktop (the Apple iMac). We then re-compile the code on the other systems and let the MCL manage heterogeneous hardware resources. Our applications automatically scale on all computing devices. We also remark that it might be possible to achieve better performance than OpenCL, for example using the CUDA [18] language and runtime. However, using a vendor-specific language considerably reduces portability, which is one of our main goals.

5 Related Work

There exist many task-based programming models [2, 4–7, 10, 13–16, 20–25]. Cilk [6] is a task-focused programming model based on a fork-join parallelism and a work-stealing scheduler. Threading Building Blocks (TBB) [11] is a C++ template library that manages and schedules tasks to be executed in parallel. Both Cilk and TBB target shared memory systems such as traditional multicore CPUs and Intel Xeon Phi manycore processors. StarPU [2] targets asynchronous heterogeneous architectures; similar to MCL, applications written in StarPU submit computational tasks implemented for a given target device, and StarPU schedules these tasks and handles necessary data transfers among the target devices. Legion [4] is a data-centric programming model and runtime system that expresses both locality and independence of program data using logical regions, from which necessary decision about data placement and task scheduling is made. On the other hand, MCL offers simple global/local synchronization primitives, which allow users to express dependencies between any two tasks. Like other distributed programming models that use OpenCL or CUDA back-ends, Legion is orthogonal to MCL. In fact, MCL can be used as back-end of Legion, enabling users to co-schedule several distributed applications on the same cluster, with multiple applications within each compute node. MCL has the distinct capability of allowing several multi-threaded applications to co-exist on the same compute node. As shown in our experiments, this is an important point, especially when applications developed by different programmers need to be coordinated.

OpenMP [8] is a directive-based programming model, which consists of compiler directives, library routines, and environment variables. The directive-based approach in OpenMP allows to incrementally parallelize existing sequential programs. OpenMP was initially designed as an open standard for portable shared memory parallelization but has recently adopted offloading constructs to support accelerator-based heterogeneous computing, which may be a suitable candidate to be used as a front-end programming model for our MCL program system. OmpSs [7] is a programming model that extends OpenMP with task-based asynchronous parallelism supporting data dependencies, targeting heterogeneous architectures. Directive extensions in OmpSs allow to automatically handle task dependencies and necessary memory transfers between the host and device. Similar to OpenMP, OpenACC [19] is the first standardization effort to provide portability across device types and compiler vendors using directive-based high-level programming. Both OpenACC and OpenMP can be used to program heterogeneous systems, but they are designed as an intra-node programming model based on the host-accelerator computing model. Therefore, efficiently utilizing multiple, heterogeneous devices at the same time can be cumbersome and error-prone. Like most programming systems, StarPU, OmpSs, OpenMP, and OpenACC target a single application with exclusive ownership of all hardware resources, while MCL supports concurrent tasks from multiple applications.

Sycl and Intel DPC++/OneAPI are recent efforts to develop single-source compiler toolchain for C++ code that needs to be executed on heterogeneous resources leveraging the OpenCL stack. While the compiler approach greatly simplify writing applications, programmers are still required to manage platforms, devices, and queues. Compared to MCL, applications do not automatically scale on multi-device systems, there is no support for multi-applications workloads

beyond what OpenCL provides, and asynchronous execution and load balancing need to be orchestrated by the programmer.

CUDA [18] and OpenCL [26] are two dominant low-level accelerator programming models for heterogeneous computing; both provide a set of APIs to offload compute regions into heterogeneous devices and manage the device memory, providing similar execution and memory models. While CUDA is a vendor-specific model targeted toward to NVIDIA GPUs, OpenCL is an architecture-independent model, offering functional portability across diverse architectures. These low-level accelerator programming models require significant rewriting and restructuring of existing applications written for traditional CPU-only machines. Moreover, CUDA is specific to NVIDIA GPUs. MCL uses OpenCL as the compatibility layer and abstracts library interfaces to hide most of the low-level programming complexities of OpenCL for the host side.

We have demonstrated that MCL can efficiently execute tasks on a variety of very different systems, provide performance portability and high throughput.

6 Conclusions and Future Work

This work introduces the Minos Computing Library (MCL), a novel system software to facilitate programming extremely heterogeneous systems that increases performance and application portability. MCL leverages asynchronous task execution to automatically overlap computation with data transfer and provide a framework to develop and integrate custom task schedulers and load balancing algorithms. Compared to current state-of-the-art task-based programming models, MCL perform dynamic resource allocation across applications, which further increases performance in multi-application workloads. We showed that code developed on a normal personal desktop automatically scales up on multi-GPU systems and down to embedded systems. Our results show speedups up to 17.5x over OpenCL for multi-GPU systems and even up to 1.88x on a single-GPU system. As future work, we plan to perform leverage MCL to investigate novel task schedulers, resource managers, and data orchestration algorithms, as well as demonstrate how MCL can be integrated with MPI applications by replacing OpenMP.

Acknowledgements

This research is supported by the Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) LAB 19-2119 and Defense Advanced Research Projects Agency (DARPA) HR001117S0055, Program Area Domain-Specific System on Chip (DSSoC).

References

- [1] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. <http://tensorflow.org/> Software available from tensorflow.org.
- [2] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.
- [3] D. F. Bacon, R. Rabbah, and S. Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63. <https://doi.org/10.1145/2436256.2436271>
- [4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing locality and independence with logical regions. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2012). <https://doi.org/10.1109/SC.2012.71>
- [5] M. E. Belviranlı, L. N. Bhuyan, and R. Gupta. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization* 9, 4 (2013), 1–20. <https://doi.org/10.1145/2400682.2400716>
- [6] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM New York, NY, USA, 207–216.
- [7] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. 2012. Productive Programming of GPU Clusters with OpenMP. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. 557–568. <https://doi.org/10.1109/IPDPS.2012.58>
- [8] L. Dagum and R. Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science AND Engineering* 5, 1 (1998), 46–55.
- [9] B. Dally. 2010. GPU Computing to Exascale and Beyond.
- [10] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press New York, NY, USA, 212–223.
- [11] Intel. [n. d.]. Threading Building Blocks. [Online]. Available: <https://software.intel.com/en-us/intel-tbb>. (Accessed Feb. 1, 2019).
- [12] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming* 43, 5 (01 Oct 2015), 752–785.
- [13] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. 2014. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.
- [14] L. V. Kale and S. Krishnan. 1993. *CHARM++: a portable concurrent object oriented system based on C++*. Vol. 28. ACM.
- [15] J. Kim, H. Kim, J. H. Lee, and J. Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. In *16th ACM symposium on Principles and practice of parallel programming*. ACM, San Antonio, TX, USA, 277–288. <https://doi.org/10.1145/1941553.1941591>
- [16] M. Kotsifakou, P. Srivastava, M.D. Sinclair, R. Komuravelli, V. Adve, and S. Adve. 2018. HPVM: heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Vienna, Austria, 68–80. <https://doi.org/10.1145/3178487.3178493>
- [17] S. Mittal and J. S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *Comput. Surveys* 47, 4 (2015), 1–35. <https://doi.org/10.1145/2788396>
- [18] J. Nickolls and I. Buck. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*.
- [19] OpenACC. 2015. OpenACC: Directives for Accelerators.
- [20] J. Planas, R.M. Badia, E. Ayguadé, and J. Labarta. 2013. Self-Adaptive OpenMP Tasks in Heterogeneous Environments. In *IEEE 27th International Symposium on Parallel and Distributed Processing*. 138–149. <https://doi.org/10.1109/IPDPS.2013.53>
- [21] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. 2013. Semi-automatic restructuring of offloadable tasks for many-core accelerators. (2013), 1–12. <https://doi.org/10.1145/2503210.2503285>
- [22] C.J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, Cascais, Portugal, 233–248. <https://doi.org/10.1145/2043556.2043579>
- [23] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. 2013. Dandelion. (2013), 49–68. <https://doi.org/10.1145/2517349.2522715>
- [24] T. R. W. Scogland, B. Rountree, W. C. Feng, and B. R. De Supinski. 2012. Heterogeneous task scheduling for accelerated OpenMP. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012* (2012), 144–155. <https://doi.org/10.1109/IPDPS.2012.23>
- [25] K. Spafford, J. Meredith, and J. S. Vetter. 2010. Maestro: Data Orchestration and Tuning for OpenCL Devices. In *Euro-Par 2010 - Parallel Processing*, Pasqua D’Ambra, Mario Guarracino, and Domenico Talia (Eds.), Vol. 6272. Springer Berlin Heidelberg, 275–286. https://doi.org/10.1007/978-3-642-15291-7_26
- [26] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering* 12, 3 (2010), 66–73.
- [27] J.S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Paterka, M. Strout, and J. Wilke. 2018. *Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity*. Technical Report. USDOE Office of Science (SC) (United States). <https://doi.org/10.2172/1473756>
- [28] Wikipedia. [n. d.]. NVLink. [Online]. Available: <https://en.wikipedia.org/wiki/NVLink>. (Accessed Feb. 1, 2019).