Parallel Computing 58 (2016) 37–55

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL

Ashwin M. Aji[a,*], Antonio J. Peña[b], Pavan Balaji[c], Wu-chun Feng[d]

[a] AMD Research, Advanced Micro Devices, United States
[b] Barcelona Supercomputing Center, Spain
[c] Dept. of Mathematics and Computer Science, Argonne National Lab., United States
[d] Dept. of Computer Science, Virginia Tech, United States

A R T I C L E I N F O

Article history:
Received 19 November 2015
Revised 7 April 2016
Accepted 15 May 2016
Available online 1 June 2016

Keywords:
OpenCL
Runtime systems
Scheduling

A B S T R A C T

The OpenCL specification tightly binds a command queue to a specific device. For best performance, the user has to find the ideal queue-device mapping at command queue creation time, an effort that requires a thorough understanding of the underlying device architectures and kernels in the program. In this paper, we propose to add scheduling attributes to the OpenCL context and command queue objects that can be leveraged by an intelligent runtime scheduler to automatically perform ideal queuedevice mapping. Our proposed extensions enable the average OpenCL programmer to focus on the algorithm design rather than scheduling and to automatically gain performance without sacrificing programmability. As an example, we design and implement an OpenCL runtime for task-parallel workloads, called MultiCL, which efficiently schedules command queues across devices.

Our case studies include the SNU benchmark suite and a real-world seismology simulation. To benefit from our runtime optimizations, users have to apply our proposed scheduler extensions to only four source lines of code, on average, in existing OpenCL applications. We evaluate both single-node and multinode experiments and also compare with SOCL, our closest related work. We show that MultiCL maps command queues to the optimal device set in most cases with negligible runtime overhead.

1. Introduction

Coprocessors are being increasingly adopted in today's high-performance computing (HPC) clusters. In particular, production codes for many scientific applications, including computational fluid dynamics, cosmology, and data analytics, use accelerators for high performance and power efficiency. Diverse types of accelerators exist, including graphics processing units (GPUs) from NVIDIA and AMD and the Xeon Phi coprocessor from Intel. Compute nodes typically include CPUs and a few accelerator devices. In order to enable programmers to develop portable code across coprocessors from various vendors and architecture families, general-purpose parallel programming models, such as the Open Computing Language (OpenCL) [1], have been developed and adopted.

[*] Corresponding author.
E-mail addresses: ashwin.aji@amd.com, aaji@cs.vt.edu (A.M. Aji), apenya@mcs.anl.gov (A.J. Peña), balaji@mcs.anl.gov (P. Balaji), feng@cs.vt.edu (W.-c. Feng).

http://dx.doi.org/10.1016/j.parco.2016.05.006
0167-8191/© 2016 Elsevier B.V. All rights reserved.

OpenCL features workflow abstractions called *command queues* through which users submit read, write, and execute commands to a specific device. However, the OpenCL specification tightly couples a command queue with a specific single device for the entire execution with no runtime support for cross-device scheduling. For best performance, programmers thus have to find the ideal mapping of a queue to a device at command queue *creation time*, an effort that requires a thorough understanding of the kernel characteristics, underlying device architecture, node topology, and various data-sharing costs that can severely hinder programmability. Researchers have explored splitting data-parallel kernels across multiple OpenCL devices, but their approaches do not work well for scheduling task-parallel workloads with multiple concurrent kernels across several command queues.

To automatically gain performance in OpenCL programs without sacrificing programmability, we decouple the command queues from the devices by proposing scheduling policies to the OpenCL specification. We define new attributes to the `cl_context` and `cl_command_queue` objects, which denote global and local scheduling policies, respectively. While the context-specific global scheduling policy describes the queue–device mapping methodology, the queue-specific local policy indicates individual queue scheduling choices and workload hints to the runtime. Local queue scheduling policies may be applied for the entire lifetime of the command queues, implicit synchronization epochs, or any explicit code regions. We also propose an OpenCL API extension to specify per-device kernel execution configurations; this function enables the scheduler to dynamically choose the appropriate configuration at kernel launch time, thereby associating a kernel launch with a high-level command queue rather than the actual physical device. Our proposed hierarchical scheduling policies enable the average user to focus on enabling task parallelism in algorithms rather than device scheduling.

To demonstrate the efficacy of our proposed OpenCL extensions, we design and implement MultiCL, an example runtime system for task-parallel workloads that leverages the policies to dynamically schedule queues to devices. We build MultiCL on top of the SnuCL OpenCL implementation [2], which provides cross-vendor support. We design three runtime modules in MultiCL: an offline device profiler, an online kernel profiler, and an online device mapper. We also implement several optimizations to reduce the online kernel profiling overhead, including efficient device–to–device data movement for data-intensive kernels, minikernel profiling for compute-intensive kernels, and caching of profiled data parameters.

Our proposed API extensions are OpenCL version independent. By providing simple modular extensions to the familiar OpenCL API, we enable different schedulers to be composed and built into an OpenCL runtime. We do not aim to design the hypothetical one–size–fits–all ideal scheduling algorithm. Instead, users may choose the right set of runtime parameters to control scheduling decisions depending on the program's requirements and the user's programming skill level. Our solution thus enhances programmability and delivers automatic performance gains.

Our case studies include the SNU-NPB OpenCL benchmark suite and a real-world seismology simulation. We show that, on average, users have to modify only *four* source lines of code in existing applications in order to benefit from our runtime optimizations. We also compare the performance and programmability of MultiCL with SOCL, an OpenCL frontend to the StarPU runtime system, and discuss their tradeoffs. Our MultiCL runtime in most cases schedules command queues to the optimal device combination, with an average runtime overhead of 10% for the SNU-NPB benchmarks and negligible overhead for the seismology simulation. Our evaluation includes both single-node and multinode experiments, where we show how MultiCL adapts to the different algorithmic patterns and yields a highly efficient device scheduling.

The paper is organized as follows. In Section 2 we describe relevant details of the OpenCL programming model, and in Section 3 we discuss some related work. In Section 4, we describe our proposed OpenCL extensions. The MultiCL runtime design and optimizations are discussed in Section 5. Our experimental setup is explained in Section 6. We present and discuss our case studies in Sections 7 and 8. We present concluding thoughts in Section 9.

2. Background

In this section, we review the OpenCL programming model and describe the SnuCL framework and runtime system, which we extend in this work.

2.1. OpenCL programming model

OpenCL [1] is an open standard and parallel programming model for a variety of platforms, including NVIDIA and AMD GPUs, FPGAs, Intel Xeon Phi coprocessors, and conventional multicore CPUs, in which the different devices are exposed as accelerator coprocessors. OpenCL follows a kernel-offload model, where the data-parallel, compute-intensive portions of the application are offloaded from the CPU host to the coprocessor device.

OpenCL developers must pick one of the available platforms or OpenCL vendor implementations on the machine and create *contexts* within which to run the device code. Data can be shared only across devices within the same context. A device from one vendor will not typically be part of the same platform and context as the device from another vendor.

While OpenCL is a convenient programming model for writing portable applications across multiple accelerators, its performance portability remains a well-known and open issue [3]. Application developers may thus maintain different optimizations of the same kernel for different architectures and explicitly query and schedule the kernels to be executed on the specific devices that may be available for execution.

In OpenCL, kernel objects are created per context. However, the kernel launch configuration or work dimensions are set globally per kernel object at kernel launch, and per-device kernel configuration customization is possible only through custom conditional programming at the application level. Moreover, no convenient mechanism exists to set different kernel configurations for different kernel–device combinations dynamically. Therefore, the OpenCL interface and device scheduling are tightly coupled.

2.2. SnuCL

SnuCL [2] is an open source OpenCL implementation and runtime system that supports cross-vendor execution and data sharing in OpenCL kernels. This OpenCL implementation provides users with a unified OpenCL platform on top of the multiple separate vendor-installable client drivers. SnuCL features an optional cluster mode providing seamless access to remote accelerators using MPI for internode communications.

In our work we extend SnuCL's single-node runtime mode for cross-vendor execution. Although our optimizations can be applied directly to the cluster mode as well, these fall out of the scope of this paper.

3. Related work

The problem of scheduling among CPU and GPU cores has been extensively studied and falls broadly into two categories: interapplication scheduling and intra-application scheduling.

3.1. Interapplication scheduling

Interapplication schedulers [4,5] distribute entire kernels from different applications across the available compute resources. Their solutions are designed primarily for multitenancy, power efficiency, and fault tolerance in data centers.

Remote accelerator virtualization solutions such as the cluster mode of SnuCL, rCUDA [6,7], or VOCL [8] provide seamless access to accelerators placed on remote nodes. These address the workload distribution concern in different ways. SnuCL's cluster mode permits remote accelerator access in clusters only to those nodes within the task allocation, and users have to implement their own workload distribution and scheduling mechanisms. rCUDA enables global pools of GPUs within compute clusters, performing cross-application GPU scheduling by means of extensions to the cluster job scheduler [9]; as with SnuCL, users have to explicitly deal with load distribution and scheduling within the application. VOCL implements its own automatic scheduler, which can perform device migrations according to energy, fault tolerance, on-demand system maintenance, resource management, and load-balancing purposes [10,11]. This scheduling mechanism, however, is limited to performing transparent context migrations among different accelerators; it is not aimed at providing performance-oriented workload distribution and scheduling.

While some of these solutions provide scheduling support across applications, our solution provides scheduling capabilities across command queues *within* an application.

3.2. Intra-application scheduling

Intra-application scheduling strategies are programming model dependent. These distribute either loop iterations in directive-based applications or work groups in explicit kernel offload-based models; in other words, *work* can mean either loop iterations or work groups. These are essentially *device aggregation* solutions, where the scheduler tries to bring homogeneity to the heterogeneous cores by giving them work proportional to their compute power.

The possibility of presenting multiple devices as a single device to the OpenCL layer and performing workload distribution internally has been previously explored. Both [12] and [13] use static load-partitioning approaches for intrakernel workload distribution, and the authors of [12] leverage their own API. FluidiCL [14] performs work stealing to dynamically distribute work groups among CPU and GPU cores with low overhead. Maestro [15] is another unifying solution addressing device heterogeneity, featuring automatic data pipelining and workload balancing based on a performance model obtained from install-time benchmarking. Maestro's approach requires autotunable kernels that obtain the size of their workloads at runtime as parameters. Qilin [16] does adaptive mapping of computation to CPU and GPU processors by using curve fitting against an evolving kernel performance database. These approaches provide fine-grained scheduling at the kernel or loop level and exploit data parallelism in applications. In contrast, our work performs coarser-grained scheduling at the *command queue* level to enable task parallelism between kernels and command queues in applications.

SOCL [17] also extends OpenCL to enable automatic task dependency resolution and scheduling and performs automatic device selection functionality by performance modeling. It applies the performance modeling at kernel granularity, and this option is not flexible. In contrast, we perform workload profiling at synchronization epoch granularity. Our approach enables a more coarse-grained and flexible scheduling that allows making device choices for kernel groups rather than individual kernels. Also, our approach reduces the profile lookup time for aggregate kernel invocations, decreasing runtime overhead. In SOCL, dynamically scheduled queues are automatically distributed among devices, being bound for the entire duration of the program. In our work, conversely, we enable users to dynamically control the duration of queue–device binding for specific code regions for further optimization purposes. In addition, we enable scheduling policies at both the context level (*global*) and the command queue level (*local*). The latter may be set and reset during different phases of the program.

Table 1
Proposed OpenCL extensions.

CL function	CL extension	Parameter names	Option(s)
clCreateContext	New parameters and options	CL_CONTEXT_SCHEDULER	ROUND_ROBIN AUTO_FIT
clCreateCommandQueue	New parameters	SCHED_OFF	N/A
clSetCommandQueue-SchedProperty	New CL API	SCHED_AUTO_STATIC SCHED_AUTO_DYNAMIC SCHED_KERNEL_EPOCH SCHED_EXPLICIT_REGION SCHED_ITERATIVE SCHED_COMPUTE_BOUND SCHED_IO_BOUND SCHED_MEMORY_BOUND	
clSetKernelWorkGroupInfo	New CL API	N/A	N/A

Furthermore, our solution enables the launch configuration to be decoupled from the launch function, providing capabilities for customizing the kernel–device configuration.

In this paper we extend our previous work [18] by analyzing the applicability of our proposal to OpenCL v2.0 and beyond, comparing the performance and programmability of MultiCL with SOCL, and demonstrating the efficacy of the MultiCL approach for MPI+OpenCL applications.

4. Proposed OpenCL extensions

In this section we describe our proposed OpenCL API extensions (Table 1) to express global and local scheduling policies and to decouple kernel launches from the actual device.

4.1. Contextwide global scheduling

To express global queue scheduling mechanisms, we propose a new context property called CL_CONTEXT_SCHEDULER. This context property can be assigned to a parameter denoting the global scheduling policy. Currently, we support two global scheduler policies: *round robin* and *autofit*. The round-robin policy schedules the command queue to the next available device when the scheduler is triggered. This approach is expected to cause the least overhead but not always produce the optimal queue–device map. On the other hand, the autofit policy decides the most optimal queue–device mapping when the scheduler is triggered. The global policies, in conjunction with the local command queue specific options, will determine the final queue–device mapping.

4.2. Local scheduling options

While command queues that are created within the same context share data and kernel objects, they also share the context's global scheduling policy. We extend the OpenCL command queue to specify a local scheduling option that is queue-specific. The combination of global and local scheduler policies can be leveraged by the runtime to result in a more optimal device mapping. The command queue properties are implemented as bitfields, and so the user can specify a combination of local policies.

Setting the command queue scheduling property to either SCHED_AUTO_$*$ or SCHED_OFF determines whether the particular queue is opting in or out of the automatic scheduling, respectively. For example, an intermediate or advanced user may want to manually optimize the scheduling of just a subset of the available queues by applying the SCHED_OFF flag to them, while the remaining queues may use the SCHED_AUTO_DYNAMIC flag to participate in automatic scheduling. Static vs. dynamic automatic scheduling provides a tradeoff between speed and optimality, which is explained in Section 5. Command queue properties can also specify scheduler triggers to control the scheduling frequency and scheduling code regions. For example, the SCHED_KERNEL_EPOCH flag denotes that scheduling should be triggered after a batch of kernels (*kernel epoch*) is synchronized and not after individual kernels. The SCHED_EXPLICIT_REGION flag denotes that scheduling for the given queue is triggered between explicit start and end regions in the program, and the new clSetCommandQueueSchedProperty OpenCL command is used to mark the scheduler region and set more scheduler flags if needed. Queue properties may also be used to provide optimization hints to the scheduler. Depending on the expected type of computation in the given queue, the following properties may be used: SCHED_COMPUTE_BOUND, SCHED_MEM_BOUND, SCHED_IO_BOUND, or SCHED_ITERATIVE. For example, if the SCHED_COMPUTE_BOUND flag is used, the runtime performs minikernel profiling in order to reduce overhead (see Section 5).

The proposed command queue properties are meant to be used as fine-tuning parameters for scheduling; their use is not mandatory. Advanced users may choose to ignore all the properties completely and instead manually schedule all the queues, whereas intermediate users with some knowledge of the program may select a subset of properties as runtime hints (e.g., SCHED_COMPUTE_BOUND or SCHED_IO_BOUND) depending on the workload type. Inexperienced users may just use SCHED_AUTO_DYNAMIC and ignore the rest of the properties, so that the runtime decides everything, at the expense of a potentially higher runtime overhead and lower performance.

4.3. Device-specific kernel configuration

The parameters to the OpenCL kernel launch functions include a command queue, a kernel object, and the kernel's launch configuration. The launch configuration is often determined by the target device type, and it depends on the device architecture. Currently, per-device kernel configuration customization is possible only through custom conditional programming at the application level. The device-specific launch function forces the programmer to manually schedule kernels on a device, a process that leads to poor programmability.

We propose a new OpenCL API function called clSetKernelWorkGroupInfo to independently set unique kernel configurations to different devices. This function enables the programmer to separately express the different combinations of kernel configuration and devices beforehand so that when the runtime scheduler maps the command queues to the devices, it can also profile the kernels using the device-specific configuration that was set before. The clSetKernelWorkGroupInfo function may be invoked at any time before the actual kernel launch. If the launch configuration is already set before the launch for each device, the runtime simply uses the device-specific launch configuration to run the kernel on the dynamically chosen device. We do not change the parameters to the clEnqueueNDRangeKernel or to any other launch API, but the kernel configuration parameters are ignored if they are already set by using clSetKernelWorkGroupInfo.

4.4. Compatibility

Since our solution involves the most basic OpenCL objects (e.g., contexts, command queues, and kernels), our work is compatible with all OpenCL versions (1.0, 1.1, 1.2, 2.0, and 2.1). Our work does not rely on characteristics of particular OpenCL versions, thus making it a generic and forward-compatible solution. In this section, we discuss the compatibility of our proposal to certain niche OpenCL features like sub-devices and device queues.

4.4.1. Sub-devices

The function clCreateSubDevices from OpenCL 1.2 creates a group of cl_device_id subobjects from a parent device object. Our solution works seamlessly with cl_device_id objects that are returned either by the OpenCL platform or by the cl_device_id objects that are created by clCreateSubDevices. Our example scheduler handles all cl_device_id objects and makes queue–device mapping decisions uniformly.

4.4.2. Device queues

The OpenCL 2.0 release introduced an interesting feature called *device enqueue*, which enables users to enqueue new kernel launches from an in-flight kernel itself, with the limitation of enqueuing only on the device executing the given kernel. It is intended to avoid host intervention in enqueuing new kernel launches, hence enabling the possibility of eliminating unnecessary launch overheads. Device enqueue is intended mainly to benefit inherently recursive applications.

This feature is fully compatible with our proposal: one simply adapts the extensions we propose to the clCreateCommandQueue call to the new clCreateCommandQueueWithProperties. Since our workload estimation techniques work at the host-code level, device-code features do not affect them. The time spent executing device-enqueued kernels is seen as part of the main kernel execution time from the host code, hence obtaining the desired scheduling behavior.

This feature, however, does prevent our proposal from scheduling the device-enqueued kernel launches on different devices, as opposed to the original host-centric approach. The tradeoff between these two possibilities is application-dependent and should be explored on a case-by-case basis. This is worthy of a separate study and left for future work.

5. The MultiCL runtime system

In this section, we explain the design of the MultiCL runtime system and discuss key optimization tradeoffs.

5.1. Design

The SnuCL runtime creates a *scheduler* thread per user process, but the default scheduler thread statically maps the incoming commands to the explicitly chosen target device—that is, manual scheduling. MultiCL is our extension of the SnuCL runtime, with the added functionality of automatic command queue scheduling support to OpenCL programs. MultiCL's design is depicted in the left portion of Fig. 1. The user's command queues that are created with the SCHED_OFF flag are statically mapped to the chosen device, whereas those that have the SCHED_AUTO flag are automatically scheduled by MultiCL. Further, the user-specified context property (e.g., AUTO_FIT) determines the scheduling algorithm for the pool

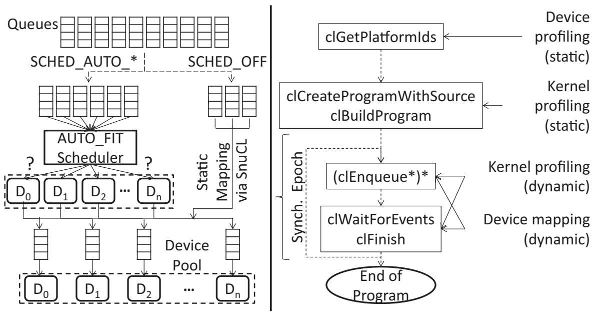

Fig. 1. Left: MultiCL runtime design and extensions to SnuCL. Right: Invoking MultiCL runtime modules in OpenCL programs.

of dynamically mapped command queues. Once a user queue is mapped to the device, its commands are issued to the respective device-specific queue for final execution.

The MultiCL runtime consists of three modules: (1) device profiler, where the execution capabilities (memory, compute, and I/O) of the participating devices are collected or inferred; (2) kernel profiler, where kernels are transformed and their execution times on different devices are measured or projected; and (3) device mapper, where the participating command queues are scheduled to devices so that queue completion times are minimal. The OpenCL functions that trigger the respective modules are shown in the right portion of Fig. 1.

5.1.1. *Device profiler*

The device profiler, which is invoked once during the `clGetPlatformIds` call, retrieves the static device profile from the profile cache. If the profile cache does not exist, then the runtime runs data bandwidth and instruction throughput benchmarks and caches the measured metrics as static per-device profiles in the user's file system. The profile cache location can be controlled by environment variables. The benchmarks are derived from the SHOC benchmark suite [19] and NVIDIA SDK and are run for data sizes ranging from being latency bound to bandwidth bound. The benchmarks measuring host-to-device (H2D) bandwidths are run for all the CPU socket–device combinations, whereas the device-to-device (D2D) bandwidth benchmarks are run for all the device–device combinations. These benchmarks are included as part of the MultiCL runtime. Bandwidth numbers for unknown data sizes are computed by using simple interpolation techniques. The instruction throughput of a device (or peak flop rate) can also be obtained from hardware specifications and manually included in the device profile cache. The benchmarks are run again only if the system configuration changes, for example, if devices are added or removed from the system or the device profile cache location changes. In practice, however, the runtime just reads the device profiles from the profile cache once at the beginning of the program.

5.1.2. *Kernel profiler*

Kernel execution times can be estimated by performance modeling or performance projection techniques, but these approaches either are done offline or are impractical because of their large runtime overheads. We follow a more practical approach in that we run the kernels once per device and store the corresponding execution times as part of the kernel profile. Arguably, this approach may cause potential runtime overhead to the current programs; in this section we therefore discuss several ways to mitigate the overhead. Our experiments (Section 6) indicate that, upon applying the runtime optimizations, the runtime overhead is minimal or sometimes negligible when the optimal device combinations are chosen for the given kernels. Static kernel transformations, such as minikernel creation, are performed during `clCreateProgramWithSource` and `clBuildProgram`, whereas dynamic kernel profiling is done at synchronization points or at user-defined code regions.

5.1.3. *Device mapper*

Each `clEnqueue-` command is intercepted by the device mapper, and the associated queue is added to a ready queue pool for scheduling. We use the per-queue aggregate kernel profiles and apply a simple dynamic programming approach that guarantees ideal queue–device mapping and, at the same time, incurs negligible overhead because the number of devices in present-day nodes is not high. Once the scheduler is invoked and maps the queue to a device, the queue is removed from the queue pool, and its commands are issued to the new target device. On the one hand, the scheduler can *actively* be invoked for every kernel invocation, but that approach can cause significant runtime overhead due to potential cross-device data migration. On the other hand, the runtime can simply aggregate the profiled execution costs for every enqueue command, and the scheduler can be invoked at synchronization epoch boundaries or at any other user-specified location in the program. The scheduler options discussed in the previous section can be used to control the frequency and location of invoking the scheduler, which can further control the overhead vs. optimality tradeoff.

```
__kernel void foo(...) {
 /* MultiCL inserts the below transformation code
    to run only the first workgroup (minikernel) */
 if(get_group_id(0)+get_group_id(1)+get_group_id(2)
     !=0)
  return;
 /* ... actual kernel code ... */
}
```

Fig. 2. Minikernel transformation example.

5.2. Static command queue scheduling

In the static command queue scheduling approach (with SCHED_AUTO_STATIC) we use the device profiler and device mapper modules of our runtime and do not perform dynamic kernel profiling; in other words, we statically decide the command queue schedules based only on the device profiles. Users can select this mode as an approximation to reduce scheduling overhead, but the optimal device may not be selected certain times. The MultiCL runtime uses the command queue properties (compute intensive, memory intensive, or I/O intensive) as the selection criterion and chooses the best available device for the given command queue.

5.3. Dynamic command queue scheduling

In the dynamic command queue scheduling approach (with SCHED_AUTO_DYNAMIC) we use the kernel profiling and device mapping modules of our runtime and selectively use the device profiling data. That is, we dynamically decide the command queue schedules based only on the kernel and device profiles. Users can choose runtime options to mitigate the runtime overhead associated with dynamic kernel profiling.

5.3.1. Kernel profile caching for iterative kernels

We cache the kernel profiles in memory as key–value pairs, where the key is the kernel name and the value is its performance vector on the devices. The cached kernel profiles are used to schedule future kernel invocations. We define a *kernel epoch* as a collection of kernels that have been asynchronously enqueued to a command queue. Synchronizing after a kernel epoch on the command queue will block until all the kernels in the epoch have completed execution. We also cache the performance profiles of kernel epochs for further overhead reduction. The key for a kernel epoch is just the set of the participating kernel names, and the value is the aggregate performance vector of the epoch on all the devices. The user can provide runtime options to batch schedule either kernel epochs or individual kernels. Our approach significantly reduces kernel profiling overhead.

Iterative kernels benefit the most because of kernel reuse. Arguably, some kernels may perform differently across iterations, however, or their performances may change periodically depending on the specific phase in the program. To address this situation, the user can set a program environment flag to denote the iterative scheduler frequency, which tells our scheduler when to recompute the kernel profiles and rebuild the profile cache. In practice, we have found iterative kernels to have the least overhead, because the overhead that is incurred during the first iteration or a subset of iterations is amortized over the remaining iterations.

5.3.2. Minikernel profiling for compute-intensive kernels

Although kernel profile caching helps iterative applications, noniterative applications still incur profiling overhead, a situation that is especially true for compute-intensive kernels. To select the best device, we need to know only the kernel's *relative* performances and not necessarily the absolute kernel performances. Therefore, we create a technique called *minikernel profiling*, which is conceptually similar to our miniemulation technique [20], where we run just a single workgroup of the kernel on each participating device and collect the relative performances in the kernel profiles. Our approach dramatically reduces runtime overhead, as discussed in Section 6. The minikernel profiling approach is best suited for those kernels whose workgroups often exhibit similar behavior and share similar runtime statistics, a situation typical of data-parallel workloads. Minikernel profiling is enabled by using the SCHED_COMPUTE_BOUND flag. However, users who want more profiling accuracy for the given workload can simply ignore the flag to enable full kernel profiling, but with some runtime overhead.

To implement minikernel profiling, we cannot simply launch a kernel with a single workgroup, because the kernel's work distribution logic may not guarantee a reduction in the profiling overhead. Instead, we modify the source kernel to create a *minikernel*, and we insert a conditional that allows just the first workgroup to execute the kernel and forces all the other workgroups to return immediately (e.g., Fig. 2). We profile the minikernel with the same launch configuration as the original kernel, so the kernel's work distribution does not change the amount of work done by the first workgroup. Our approach thus guarantees reduction in the profiling overhead.

We intercept the clCreateProgramWithSource call and create a minikernel object for every kernel. We build the program with the new minikernels into a separate binary by intercepting the clBuildProgram call. Although this method

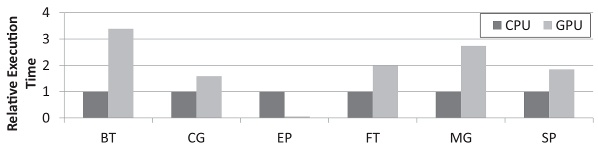

Fig. 3. Relative execution times of the SNU-NPB benchmarks on CPU vs. GPU.

doubles the OpenCL build time, we consider this to be an initial setup cost that does not change the actual runtime of the program. We note also that the minikernel profiling approach requires access to the kernel source in order to perform the optimization.

5.3.3. Data caching for I/O-intensive kernels

One of the steps in kernel profiling is to transfer the input data sets from the source device to each participating device before profiling them. Clearly, the data transfer cost adds to the runtime overhead. With n devices, the brute-force approach involves making D2D data transfers $n - 1$ times from the source device to every other device, followed by an intradevice data transfer at the source. However, the current vendor drivers do not support direct D2D transfer capabilities across vendors and device types. Thus, each D2D transfer is performed as a D2H-H2D double operation via the host memory, resulting in $n - 1$ D2H and $n - 1$ H2D operations.[1] Recognizing, however, that the host memory is shared among all the devices within a node, we optimize the data transfer step by doing just a single D2H copy from the source device to the host, followed by $n - 1$ H2D data transfers. In addition, we cache the incoming data sets in each destination device so that if our device mapper decides to migrate the kernel to a different target device, the required data is already present in the device. With this optimization, however, we trade off increased memory footprint in each device for less data-transfer overhead.[2]

6. Experimental setup

Our experimental compute node has a dual-socket oct-core AMD Opteron 6134 (Magny-Cours family) processor and two NVIDIA Tesla C2050 GPUs, thus forming three OpenCL devices (1 CPU and 2 GPU devices). Each CPU node has 32 GB of main memory, and each GPU has 3 GB of device memory. We use the CUDA driver v313.30 to manage the NVIDIA GPUs and the AMD APP SDK v2.8 to drive the AMD CPU OpenCL device. For the performance comparisons with SOCL, we use StarPU v1.2 as the corresponding runtime. The network interface is close to CPU socket 0, and the two NVIDIA GPUs have affinity to socket 1, which creates nonuniform host–device and device–device distances (and therefore data transfer latencies) depending on the core affinity of the host thread. The MultiCL runtime scheduler incorporates the heterogeneity in compute capabilities as well as device distances when making device mapping decisions.

We first evaluate the performance and programmability benefit of MultiCL by using the NAS Parallel Benchmarks. We then compare its performance with that of SOCL, our closest related work and an OpenCL frontend to the StarPU runtime scheduler. Next, we describe a seismology simulation application and explain its single-node OpenCL implementation. We show how MultiCL is used to automatically schedule kernels across the available devices with negligible overhead and minimal code changes. We further explore how a multinode MPI+OpenCL version of the same application changes the task–device mapping complexity, and we demonstrate that our MultiCL scheduler again automatically chooses a highly efficient mapping of tasks to devices, with no changes to the scheduling code from our single-node implementation.

7. Benchmark evaluation: NAS parallel benchmarks

The NAS Parallel Benchmarks (NPB) [23] are designed to help evaluate current and future parallel supercomputers. The SnuCL team recently developed the SNU-NPB suite [24], which consists of the NPB benchmarks ported to OpenCL. The SNU-NPB suite also has a multidevice version of the OpenCL code (SNU-NPB-MD) to evaluate OpenCL's scalability. SNU-NPB-MD consists of six applications: BT, CG, EP, FT, MG, and SP. The OpenCL code is derived from the MPI Fortran code that is available in the NPB3.3-MPI suite and is not heavily optimized for the GPU architecture. For example, Fig. 3 shows that for the single-device version, most of the benchmarks run better on the CPU but the degree of speedup varies, whereas EP runs faster on the GPU. These results reflect that the device with the highest theoretical peak performance and bandwidth—that is, the GPU—is not always the best choice for a given kernel.

Each SNU-NPB-MD benchmark has specific restrictions on the number of command queues that can be used depending on its data and task decomposition strategies, as shown in Table 2. Also, the amount of work assigned per command queue differs per benchmark. That is, some create constant work per application, and the work per command queue decreases for more queues; others create constant work per command queue, and the work per application increases for more queues. To

[1] GPUDirect for NVIDIA GPUs has markedly limited OpenCL support.

[2] The analysis of the performance of data transfers involving accelerators is out of the scope of this paper and has been covered in separate studies [21,22].

Table 2
SNU-NPB-MD benchmarks, their requirements, and our custom scheduler options.

Bench.	Classes	Cmd. queues	OpenCL scheduler option(s)
BT	S,W,A, B	Square: 1,4	SCHED_EXPLICIT_REGION, clSetKernelWorkGroupInfo
CG	S,W,A, B,C	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION
EP	S,W,A, B,C,D	Any: 1,2,4	SCHED_KERNEL_EPOCH, SCHED_COMPUTE_BOUND
FT	S,W,A	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION, clSetKernelWorkGroupInfo
MG	S,W,A, B	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION
SP	S,W,A, B,C	Square: 1,4	SCHED_EXPLICIT_REGION

use more command queues than the available devices, one could write a simple round-robin queue–device scheduler, but an in-depth understanding of the device architecture and node topology is needed for ideal scheduling. Also, some kernels have different device-specific launch configuration requirements depending on the resource limits of the target devices; and, by default, these configurations are specified only at kernel launch time. Moreover, such kernels are conditionally launched with different configurations depending on the device type (CPU or GPU). In order to *dynamically* choose the ideal kernel–device mapping, a scheduler will need the launch configuration information for all the target devices before the actual launch itself.

7.1. Evaluation with MultiCL

We enable MultiCL's dynamic command queue scheduling by making the following simple code extensions to each benchmark: (1) we set the desired scheduling policy to the context during context creation, and (2) we set individual command queue properties as runtime hints at command queue creation or around explicit code regions. In some kernels, we also use the `clSetKernelWorkGroupInfo` function to separately express the device-specific kernel launch configurations to the runtime, so that the scheduler can have the flexibility to model the kernel for a particular device along with the correct corresponding kernel launch configuration. These simple code changes, together with the MultiCL runtime optimizations, enable the benchmarks to be executed with ideal queue–device mapping. The application developer has to think only about the data-task decompositions among the chosen number of command queues and need not worry about the underlying node architecture.

Table 2 also shows our chosen MultiCL scheduler options for the different benchmarks. The iterative benchmarks typically have a "warmup" phase during the loop iterations, and we consider them to be ideal candidates for *explicit* kernel profiling because they form the most representative set of commands that will be consistently submitted to the target command queues. For such iterative benchmarks, we set the command queues with the `SCHED_EXPLICIT_REGION` property at creation time and trigger the scheduler explicitly around the warmup code region. We call `clSetCommandQueueSchedProperty` with the `SCHED_AUTO` and `SCHED_OFF` flags to start and stop scheduling, respectively. Other code regions were not considered for explicit profiling and scheduling because they did not form the most representative command set of the benchmark. We also did not choose the *implicit* `SCHED_KERNEL_EPOCH` option for iterative benchmarks because the warmup region spanned across multiple kernel epochs and the aggregate profile of the region helped generate the ideal queue–device mapping. On the other hand, the EP benchmark (random number generator) is known to be compute intensive and not iterative. At command queue creation time, we simply set the `SCHED_KERNEL_EPOCH` and `SCHED_COMPUTE_INTENSIVE` properties as runtime hints, which are valid for the queue's lifetime. In the BT and FT benchmarks, we additionally use our proposed `clSetKernelWorkGroupInfo` OpenCL API (see Section 4) to set CPU- and GPU-specific kernel launch parameters. The parameters that are later passed to `clEnqueueNDRangeKernel` are ignored by the runtime. This approach decouples the kernel launch from a particular device, thus enabling the runtime to dynamically launch kernels on the ideal device with the right device-specific kernel launch configuration.

We evaluate each benchmark with problem sizes from the smallest (S) to the largest problem size that fits on each available device, as specified in Table 2. Fig. 4 shows a performance comparison of automatic scheduling performed by MultiCL with manual round-robin techniques as the baseline. The benchmark class in the figure denotes the largest problem size for that application that could fit on the device memories, and each benchmark uses four command queues. One can schedule four queues among three devices (2 GPUs and 1 CPU) in 3^4 ways, but for our demonstration purpose we showcase five explicit schedules that we consider are likely to be explored by users: (1) CPU-only assigns all four command queues to the CPU; (2) GPU-only assigns all four command queues to one of the GPUs; (3) round-robin (GPUs) assigns two queues each to the two GPUs; (4) round-robin #1 assigns two queues to one GPU, one queue to the other GPU, and one queue to the CPU; and (5) round-robin #2 assigns two queues to the CPU and one queue to each GPU. Since five benchmarks perform

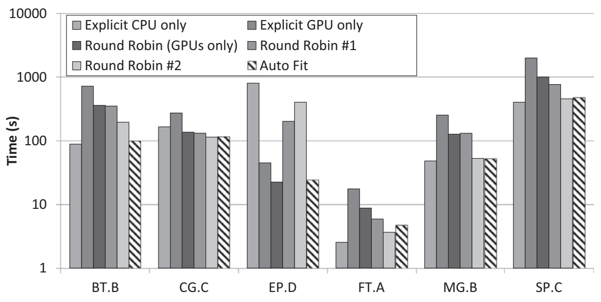

Fig. 4. Performance overview of SNU-NPB-MD for manual and our automatic scheduling. Number of command queues: 4; available devices: 1 CPU and 2 GPUs.

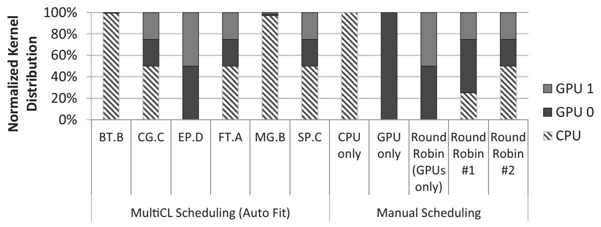

Fig. 5. Distribution of SNU-NPB-MD kernels to devices for manual and MultiCL's automatic scheduling. Number of command queues: 4; available devices: 1 CPU and 2 GPUs.

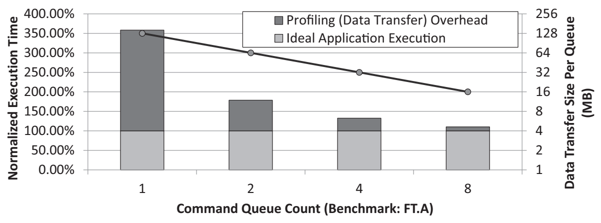

Fig. 6. Profiling (data transfer) overhead for the FT.A benchmark.

better on the CPU and EP works best on the GPU, we consider some of the above five schedules to form the best and worst queue–device mappings and expect the MultiCL scheduler to automatically find the best queue–device mapping.

We define the profiling overhead of our scheduler as the difference between the performance obtained from the ideal queue–device mapping and that obtained from the scheduler driven queue–device mapping, expressed as a percentage of the ideal performance, that is, $\frac{T_{scheduler\_map}-T_{ideal\_map}}{T_{ideal\_map}} \times 100$. Fig. 4 shows that automatic scheduling using the MultiCL runtime achieves near-optimal performances, which indirectly means ideal queue–device mapping. The geometric mean of the overall performance overhead is 10.1%. The overhead of FT is more than that of the other benchmarks, and we analyze this overhead in the next paragraph. Fig. 5 shows how the performance model in MultiCL's scheduler has distributed the kernels among the available devices. A close comparison with the benchmarks' CPU vs. GPU performance from Fig. 3 indicates that our scheduler maps queues to devices in a near-ideal manner. For example, Fig. 3 indicates that the BT and MG benchmarks perform much better on the CPU than on the GPU, and Fig. 5 indicates that our scheduler has assigned most of the kernels from all iterations to the CPU and almost none to the GPU. Similarly, EP performs best on the GPU (Fig. 3), and we see that our scheduler has assigned all the kernels to the GPU. The other benchmarks are still better on the CPU but to a lower degree; and thus we see that the CPU still gets a majority of the kernels but that the GPUs also get their share of work. We see similar trends for the other problem classes and other command queue numbers as well, but for brevity we have not included them in the paper.

7.1.1. Effect of data transfer overhead in scheduling

The FT benchmark distributes the input data among the available command queues; that is, the data per queue decreases as the number of queues increases. The MultiCL runtime performs kernel profiling only once per device for performance estimation; hence, the cost is amortized for more command queues, and our profiling overhead reduces. While Fig. 4 indicates that the profiling overhead in FT is about 45% when compared with the ideal queue–device mapping and when four command queues are used, Fig. 6 indicates that the profiling overhead decreases with increasing command queues. Further, Fig. 7 indicates that our data-caching optimization caches the profiled data on the host and reduces the D2D transfer overhead consistently by about 50% during kernel profiling. Although the other benchmarks work on similar data footprints in

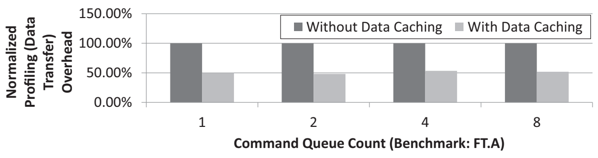

Fig. 7. Effect of data caching in reducing profiling overhead for the FT (Class A) benchmark.

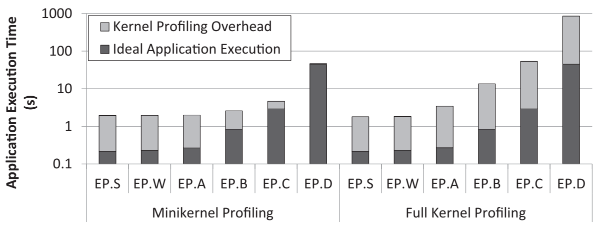

Fig. 8. Impact of minikernel profiling for the EP benchmark.

memory, they do not transfer as much data as FT does. Thus they exhibit apparently negligible data transfer overhead while scheduling.

7.1.2. Effect of minikernel profiling in scheduling

The EP benchmark does random number generation on each device and is highly compute intensive, and the CPU (non-ideal device) can be up to 20 × slower than the GPU (ideal device) for certain problem sizes. Since the full kernel profiling approach runs the entire kernel on each device before device selection, the runtime overhead compared with that of the ideal device combination can also be about 20 ×, as shown in Fig. 8. Moreover, running the full kernel means that the profiling overhead increases for larger problem sizes. On the other hand, our minikernel profiling approach just runs a single workgroup on each device, and we can see that it incurs a *constant* profiling overhead for any problem size. Minikernel profiling thus dramatically reduces the profiling overhead to only about 3% for large problem sizes, while making optimal device mapping. We perform minikernel profiling for all the other benchmarks as well; but since they are not as compute intensive as EP, the apparent benefits are negligible.

7.2. Performance comparison with SOCL

In this section, we evaluate SNU-NPB by using SOCL and compare its performance with MultiCL. We describe the SOCL extensions that were applied to the existing OpenCL programs in order to enable automatic command-queue scheduling with the StarPU runtime. We also demonstrate the shortcomings of the SOCL extensions for cross-device scheduling with device-specific kernels. In the rest of this text, SOCL and StarPU are used interchangeably.

We enabled automatic scheduling with SOCL by adding new parameters to just two OpenCL functions—`clCreateContext` and `clCreateCommandQueue`, that is, at the context and the command queue creation API, respectively. We chose the `dmda` (deque model data aware) scheduler by specifying it as a context property. We used a NULL device at command queue creation to tell the StarPU runtime to dynamically map the device to the queue. The code changes are minimal. While MultiCL also requires changes to the same OpenCL functions at a minimum, we provide additional OpenCL functions such as `clSetCommandQueueSchedProperty` and `clSetKernelWorkGroupInfo` for advanced programmer control and better performance.

These SOCL extensions trigger the runtime to perform automatic command queue scheduling. The SOCL runtime scheduler profiles kernels to build a performance model, which is then used to decide the ideal kernel–device mapping. SOCL profiles kernels in two stages: (1) it runs an instance of the kernel on all available devices and creates a per-kernel performance profile, and (2) it stores the kernel profile as part of the overall application profile in a known location on disk for future references. With SOCL, the same application has to be run multiple times to generate several kernel profiles. Once SOCL collects sufficient kernel profiles, it calculates the average performance of the kernel for the given data size in order to predict the performance of a new instance of the same kernel with the same data size without actually running the kernel on any device. The number of historical kernel profile measurements is a tradeoff point between accuracy and performance overhead and can be configured via the `STARPU_CALIBRATE_MINIMUM` environment variable. We used the default value of `STARPU_CALIBRATE_MINIMUM=10` for our experiments. However, this means that to use SOCL's performance model with low overhead, we must run every application multiple times in an online profiling or calibration phase before it may be used in production. If a system configuration changes (hardware or software), we have to calibrate the performance models again before the productions runs. The calibration may be performed offline once before the production runs or forced to be done online during the production runs. On the other hand, with MultiCL, we do not have separate calibration and

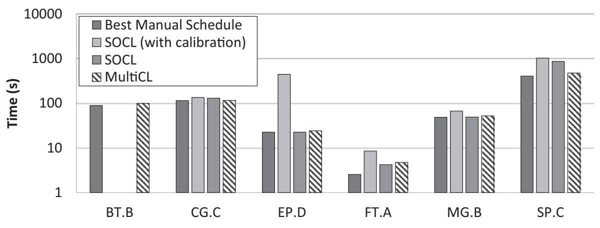

Fig. 9. Performance comparison between MultiCL and SOCL. SOCL does not have an interface to support the per-device kernels of BT, which is represented by the missing bars.

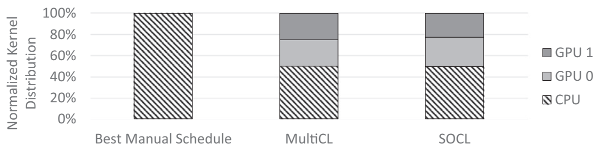

Fig. 10. Kernel-to-device distribution of the FT.A benchmark for MultiCL, SOCL and the best manual scheduling. Number of command queues: 4; available devices: 1 CPU and 2 GPUs.

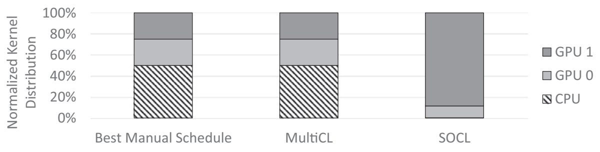

Fig. 11. Kernel-to-device distribution of the SP.C benchmark for MultiCL, SOCL and the best manual scheduling. Number of command queues: 4; available devices: 1 CPU and 2 GPUs.

production phases. Instead, we perform online kernel profiling for every production run of the application. We have developed overhead minimization techniques to mitigate the kernel execution costs and data transfer costs, thus making MultiCL more amenable to systemwide changes in hardware or software.

Fig. 9 describes the performance of SNU-NPB benchmarks when used with SOCL and compares it with MultiCL. We choose the best manual schedule from Fig. 4 as the baseline for comparison. To analyze SOCL, we measure its performance with and without the calibration (online profiling) cost. We can see that for CG, EP, FT, and MG, the performance of MultiCL is comparable to that of SOCL after calibration, which means that both frameworks map the command queues to the same set of devices with similar performance modeling overhead.

The EP benchmark's performance on the CPU is 20 × slower than that on the GPU, and both SOCL and MultiCL choose the GPU to execute the EP kernels. Therefore, its performance with calibration is about 20 × slower than its production run after calibration because the SOCL profiler executes the same kernel on the CPU *and* GPU during calibration. On the other hand, we perform minikernel profiling in MultiCL in our production runs, which reduces almost all the profiling overhead, as explained previously in Fig. 8 and can also be seen in Fig. 9.

For the FT benchmark, SOCL performs better than MultiCL but worse than the manual schedule. With MultiCL, one of the reasons for the performance gap is due to the data movement overhead during profiling, as explained in Section 7.1.1. However, SOCL will not have this overhead because the performance models will have been calibrated *before* the production run. Fig. 10 depicts the distribution of kernels to devices for the FT.A benchmark. We see that both SOCL and MultiCL have mapped half of the kernels to the CPU and distributed the rest among the GPUs, whereas the ideal mapping is to schedule all the kernels on the CPU. However, the difference in the performance between SOCL's and MultiCL's mapping and the CPU-only mapping is only about 25%, as shown in Fig. 4. The relatively low performance gap between the device mapping strategies amount to the performance difference between the ideal schedule and SOCL's mapping in Fig. 10. For the SP benchmark, SOCL performs worse than MultiCL after calibration as well. For SP.C, SOCL does not map the kernels to the right set of devices (see Fig. 11), whereas MultiCL determines the ideal mapping and hence achieves the best performance. Moreover, we have chosen the `dmda` scheduler for SOCL, which creates an offline performance model as well as takes into account the data transfer costs while scheduling. To build our performance model, we record kernel-specific profiles and data transfer costs with the actual input data and do not extrapolate our model based on the initial runs, which is a reason for a more accurate kernel-device mapping by MultiCL.

The BT benchmark consists of different sets of kernels that are optimized for CPUs and GPUs. The CL source code for BT as well as the kernel launch configurations are specified for the target architecture by using conditional compilation techniques like `#ifdef` followed by a compilation flag. With minor modifications to the application source, one can pass platform-specific build options to `clBuildProgram`, compile the same source code differently for each architecture, and store the respective kernel objects. With SOCL and with native OpenCL, however, one can specify only a single set of kernel

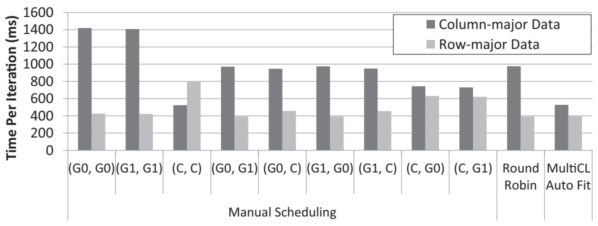

Fig. 12. FDM-Seismology performance overview.

launch parameters to `clEnqueueNDRangeKernel`. This approach caused BT to crash because the SOCL runtime scheduler applied a single set of launch parameters to both CPUs and GPUs, where they were executing architecture-aware kernel objects. On the other hand, with MultiCL, we used our proposed `clSetKernelWorkGroupInfo` API to decouple the launch parameters from the actual kernel launch itself, with which we could specify architecture-specific launch parameters for the respective architecture-aware kernel objects. SOCL does not have an interface to completely support per-device kernels, as is represented by the missing bars in Fig. 9.

7.3. Summary of programmability benefits

We add new parameters to at most four OpenCL functions in existing benchmarks and trigger the MultiCL runtime to automatically schedule the command queues and map them to the ideal combination of devices. We choose the autofit global scheduler for the context, while the command queues choose either the explicit region or kernel epoch local scheduler options (Table 2). MultiCL also has the capability to specify per-kernel launch parameters for architecture-specific optimized applications. The MultiCL scheduler performs static device profiling to collect the device distance metrics, performs dynamic kernel profiling to estimate the kernel running costs, and then computes the aggregate cost metric from the data transfer and kernel execution costs. We derive the data transfer costs from the offline device profiles and the kernel execution costs from the online kernel profiles. We use the aggregate cost metric to compute the ideal queue–device mapping.

8. Production evaluation: seismology modeling

FDM-Seismology is an application that models the propagation of seismological waves based on the finite-difference method by taking the Earth's velocity structures and seismic source models as input [25]. The application implements a parallel velocity-stress, staggered-grid finite-difference approach for propagation of waves in a layered medium. In this method, the domain is divided into a three-dimensional grid, and a one-point integration scheme is used for each grid cell. Since the computational domain is truncated in order to keep the computation tractable, absorbing boundary conditions are placed around the region of interest to keep the reflections minimal when boundaries are impinged by the outgoing waves. This strategy helps simulate unbounded domains. The simulation iteratively computes the velocity and stress wavefields within a given subdomain. Moreover, the wavefields are divided into two independent regions, and each region can be computed in parallel. The reference code of this simulation is written in Fortran [26].

8.1. OpenCL implementation for a single node

For our experiments, we extend an existing OpenCL implementation [27] of the FDM-Seismology simulation as the baseline. The OpenCL implementation divides the kernels into velocity and stress kernels, where each of these sets computes the respective wavefields at its two regions. The velocity wavefields are computed by using 7 OpenCL kernels, 3 of which are used to compute on region-1 and the other 4 kernels to compute on region-2. Similarly, the stress wavefields are computed by using 25 OpenCL kernels, 11 of which compute on region-1 and 14 kernels compute on region-2. We have two OpenCL implementations of the simulation depending on the data layout: (1) *column-major data*, which directly follows Fortran's column major array structures, and (2) *row-major data*, which uses row major array structures and is more amenable for GPU execution. Moreover, since the two wavefield regions can be computed independently, their corresponding kernels are enqueued to separate command queues. In our experimental system, the two command queues can be scheduled on the three OpenCL devices in 3^2 different ways. Fig. 12 demonstrates the performance of both versions of the kernels on different device combinations. We see that the column-major version performs best when all the kernels are run on a single CPU and performs worst when all of them are run on a single GPU; the performance difference between the two queue–device mappings is 2.7×. On the other hand, the row-major version is best when the command queues are distributed across two GPUs and is 2.3× better than the performance from the worst-case mapping of all kernels on a single CPU.

We compare the performance of two global contextwide schedulers, *round robin* and *autofit*, by simply setting the context property to either the `ROUND_ROBIN` or `AUTO_FIT` values, respectively. FDM-Seismology has regular computation per iteration, and each iteration consists of a single synchronization epoch of kernels. Thus, as our local scheduler, we can either choose the implicit `SCHED_KERNEL_EPOCH` at queue creation time or choose the `SCHED_EXPLICIT_REGION` and

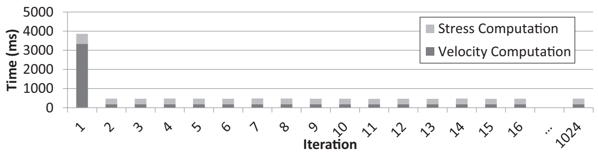

Fig. 13. FDM-Seismology performance details. Profiling overhead decreases asymptotically with more iterations.

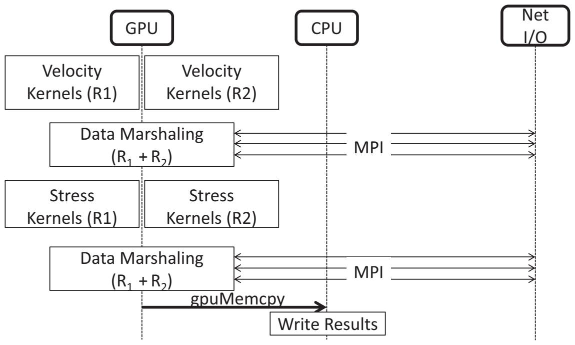

Fig. 14. Data marshaling with OpenCL in FDM-Seismology. Data dependency: stress and velocity kernels work concurrently with independent regions on separate command queues, whereas the data marshaling step works with both regions on a single command queue.

turn on automatic scheduling explicitly just for the first iteration by using `clSetCommandQueueSchedProperty`. We use the `SCHED_KERNEL_EPOCH` option in our experiments, but the final mapping and profiling overhead is expected to be the same for the other option as well. Fig. 12 shows that the autofit scheduler maps the devices optimally for both code versions. The performance of the autofit case is similar to the CPU-only case for the column-major code and is similar to the dual-GPU case for the row-major version of the code, with a negligible profiling overhead of less than 0.5%. On the other hand, the round-robin scheduler always chooses to split the kernels among the two GPUs and hence does not provide the best combination for the column-order version of the code. Fig. 13 shows that for the autofit scheduler, although the first iteration incurs runtime overhead, the added cost gets amortized over the remaining iterations.

8.2. MPI+OpenCL implementation for multiple nodes

Our MPI-based parallel version of the seismology application divides the input model into submodels along different axes such that each submodel can be computed on different CPUs (or nodes). This technique, also known as domain decomposition, allows the computation in the application to scale to a large number of nodes. Each processor computes the velocity and stress wavefields in its own subdomain and then exchanges the wavefields with the nodes operating on neighbor subdomains, after each set of velocity or stress computation. Furthermore, each processor updates its own wavefields after receiving wavefields generated at the neighbors. In summary, each computation phase is followed by an MPI communication and an update of local wavefields, and a small postcommunication computation on local wavefields. At the end of each iteration, the updated local wavefields are written to a file.

The velocity and stress wavefields are stored as large multidimensional arrays on each node. In order to optimize the MPI computation between neighbors of the domain grid, only a few elements of the wavefields, those needed by the neighboring node for its own local update, are communicated to the neighbor, rather than whole arrays. Hence, each MPI communication is surrounded by data marshaling steps, where the required elements are packed into a smaller array at the source, communicated, and then unpacked at the receiver to update its local data.

Our previous work found that the data marshaling phase performs better when they are implemented on the GPU itself rather than the CPU [28]. To accomplish MPI communication directly from the OpenCL device, we used MPI-ACC [28], a GPU-aware MPI framework—based on the MPICH MPI implementation [29]—that performs point-to-point communication among OpenCL devices across the network. Moreover, as a consequence to performing data marshaling on the device, the host-device bulk data transfers before and after each velocity-stress computation kernel are completely avoided. The need to explicitly bulk transfer data from the device to the host arises only at the end of the iteration, when the results are transferred to the host to be written to a file (Fig. 14).

We have implemented all the data marshaling code pieces as 65 different data marshaling kernels in OpenCL for this purpose. In the single node OpenCL implementation, the region-1 and region-2 kernels were computed concurrently by using two separate command queues. The data marshaling kernels are consecutive, however, and work on both regions simultaneously; that is, there is data dependency between regions 1 and 2. To address this data dependency, we create a third new command queue that is used to invoke only the data marshaling kernels. We discuss the row-major data execution

```
// Initialize command queues
cl_command_queue queue1, queue2, queue3;
// Mini-application loop
for (...) {
  // Two independent compute loops
  compute_data_1(data1, queue1);
  compute_data_2(data2, queue2);
  sync(queue1);
  sync(queue2);
  // Data resolution happens before marshaling
  marshal_data(data1, data2, queue3);
}
```

Fig. 15. Pseudo-code of the Seismology miniapplication.

mode in this section, but the column-major scenario works similarly. In summary, the application uses three command queues and we show how the MultiCL runtime schedules them to the available devices within each node.

After every computation phase, the marshaling queue synchronizes with the computation queues, a process that involves both execution and data synchronizations. The data marshaling phase requires explicit cross-queue synchronization before and after marshaling. The SnuCL runtime automatically migrates the required data to and from the target device around the marshaling phase. While splitting the velocity and stress kernels across two devices reduces the computation cost, the marshaling kernels execute on a single device and incur *data resolution* cost. In this paper, data resolution means to resolve the data incoherence, or to make the data coherent across the different devices.

Therefore, on the one hand, we can map the compute command queues across two devices to reduce computation cost, but incur data resolution cost. On the other hand, we can map all the queues to a single device to avoid the data resolution cost, but with increased computation cost. Choosing between two devices vs. a single device is a non-trivial tradeoff, because the better configuration depends on the correlation between the problem size with the following: (1) computation cost, (2) data marshaling cost, and (3) data resolution cost. To understand this tradeoff further, we implement and study a seismology “miniapplication” that captures the core computation–communication pattern of FDM-Seismology as explained in Section 8.2.1. We choose to study a miniapplication because we can rapidly and comprehensively explore the application design space, which can lead to a concrete discussion on the performance model and runtime design tradeoffs. In Section 8.2.2, we apply our lessons learned to get better insights into the queue-device mapping choices made by the MultiCL scheduler for the full seismology application. While we performed our experiments on a small 4-node cluster, we evaluate the effects of MultiCL among the OpenCL devices within a single node. The performance of the application is uniform across all the nodes in the cluster.

8.2.1. Analysis with a miniapplication

Fig. 15 describes the pseudo-code of the seismology miniapplication. It begins with a compute phase that consists of a compute-intensive kernel performing a series of multiple and multiply–add operations on two separate data sets on two independent command queues. The compute phase is followed by a data marshaling phase, which consists of a lightweight kernel running on a single command queue and takes negligible cycles to compute. However, the marshaling kernel references both the computed data sets to induce data dependency among the command queues. In summary, the compute phase is compute-intensive and works independently on two data sets, whereas the marshaling phase is not compute-intensive but has to resolve dependencies and incurs a data resolution cost. The compute and marshaling phases are executed for multiple iterations, and the average running time is reported. We have parameterized the miniapplication so that we can control the relative computation and data resolution costs by modifying the loop count and working data set sizes, respectively. However, it is algorithmically identical to the full application.

MultiCL’s AUTO_FIT scheduler optimizes for minimal total execution time. If the concurrent kernel execution time on two devices (T_K2), in addition to the data resolution time (T_DR) is less than the consecutive kernel execution time on a single device (T_K1), then the scheduler maps the compute command queues to separate devices. If the consecutive kernel execution time on a single device (T_K1) is less, then the scheduler maps all the command queues to a single device.

In our experiment, we keep the compute phase constant and vary only the working data set size, so that the data resolution cost increases for larger working data sets, whereas the compute phase time remains constant for all working data sets. Fig. 16 shows the time decomposition for kernel execution and data resolution costs for all queue–GPU combinations. We ignore the CPU device in this case because the GPUs are about 100 $\times$ faster than the CPU for this specific miniapplication example and are irrelevant in our scheduler analysis. We see from a detailed analysis in Fig. 17 that for smaller data sizes, $T\_K1 > T\_K2 + T\_DR$ and two devices are better to schedule the queues despite the data resolution cost. For larger data sizes, $T\_K1 < T\_K2 + T\_DR$ and the data resolution cost of using two devices outweighs the advantage of concurrent processing, and running everything on a single device performs better overall. As the data sizes grow, the case for a single device schedule becomes stronger as well.

8.2.2. Analysis with FDM-seismology

Fig. 18 shows the time decomposition for kernel execution and data resolution costs for all queue–GPU combinations. We note that while the case for a single GPU may become better for smaller data sizes, the application has a non-negligible se-

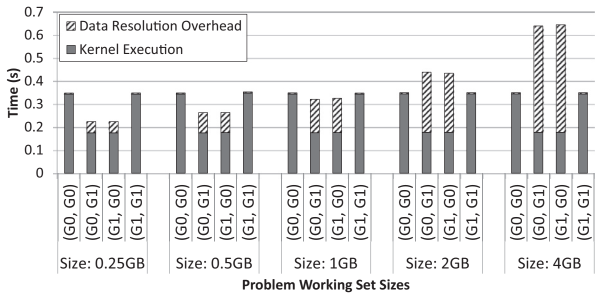

Fig. 16. Seismology Miniapplication performance overview on all queue–GPU combinations. Two GPUs are better for smaller data sizes, whereas a single GPU is better for larger data sizes.

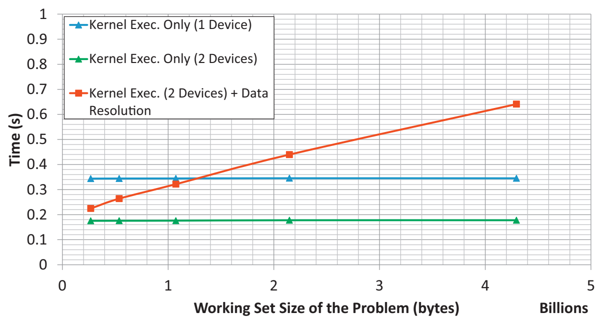

Fig. 17. Seismology Miniapplication performance analysis for single-device vs. two-device configurations. For smaller data sizes, $T\_K1 > T\_K2 + T\_DR$; that is, two devices are better. For larger data sizes, $T\_K1 < T\_K2 + T\_DR$; that is, a single device is better.

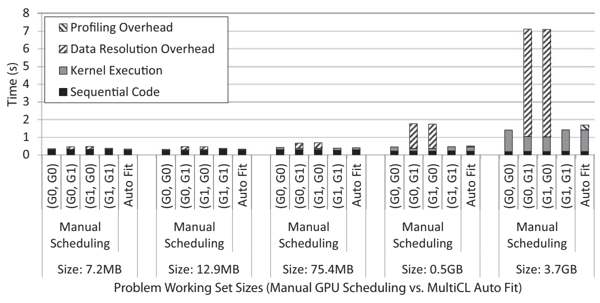

Fig. 18. FDM-Seismology application performance overview on all queue–GPU combinations. For all data sizes, a single GPU is better than using two GPUs. The sequential code cost also provides less incentive to move to multiple GPUs.

quential code, which limits the overall parallelism of the application and using multiple GPUs make lesser sense. Fig. 18 also shows that our MultiCL scheduler correctly identifies these trends in the device mapper module of the runtime and assigns all command queues to a single device for all data sets, while incurring a small profiling overhead.

While our miniapplication was controlled to have a constant compute phase for all working data sets, the compute phase may vary in real applications. We see from a detailed analysis in Fig. 19 that for all data sizes, $T\_K1 < T\_K2 + T\_DR$, and the data resolution cost of using two devices outweighs the advantage of concurrent processing, and running everything on a single device performs better overall. The reason is that the compute phase time also varies with the working data set size and there is no tradeoff point in the positive x-axis where two devices would be better than a single device. As the data sizes grow, the case for a single-device schedule becomes stronger as well.

In summary, we showed in this section that when the single-node OpenCL program was scaled to multiple nodes with MPI+OpenCL, the number of required command queues increased from two to three to perform computations and data marshaling. We showed that our MultiCL runtime adapts to the changed configuration and automatically schedules the command queues to the optimal device set within each node.

8.3. Programmability benefits

For all our experiments, we modified about four lines of code in the entire program, on average. The user is required to add new context properties to set the global scheduling policy and set the command queue properties for local policies

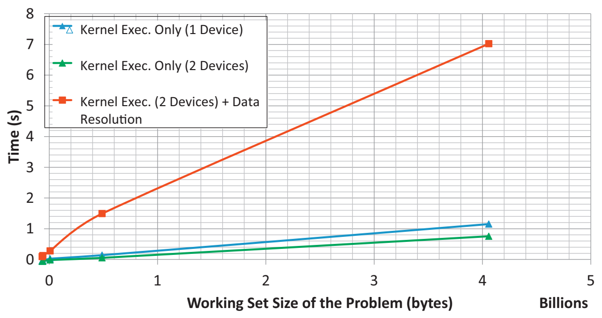

Fig. 19. FDM-Seismology application performance analysis for single-device vs. two-device configurations. For all data sizes, $T\_K1 < T\_K2 + T\_DR$; that is, a single device is better.

and runtime scheduler hints. The remaining runtime features are optional, such as using `clSetCommandQueueProperty` to explicitly control the local policy and `clSetKernelWorkGroupInfo` to specify device-specific kernel launch configurations. We have shown that with minimal code changes to a given OpenCL program, our scheduler can automatically map the command queues to the optimal set of devices, thereby significantly enhancing the programmability for a wide range of benchmarks and real-world applications. Our scheduler is shown to incur negligible overhead for our seismology simulation test cases.

9. Conclusion

We have proposed extensions to the OpenCL specification to control the scheduling both globally at the context level and locally at the command queue level. We have designed and implemented MultiCL, a runtime system that leverages our OpenCL scheduling extensions and performs automatic command queue scheduling capabilities for task-parallel workloads. Our runtime scheduler includes static device profiling, dynamic kernel profiling, and dynamic device mapping. We have designed novel overhead reduction strategies including minikernel profiling, reduced data transfers, and profile data caching. Our experiments on the NPB benchmarks and a real-world seismology simulation (FDM-Seismology) demonstrate that the MultiCL runtime scheduler maps command queues to the optimal device combination in most cases, with an average runtime overhead of 10% for the NPB benchmarks and negligible overhead for the FDM-Seismology application, both on its single and multiple node versions. Our comparisons with SOCL, a closely related approach, reveal similar or better performance for MultiCL, but also the relevance of supporting per-device kernel configurations. Our proposed OpenCL extensions and the associated runtime optimizations enable users to focus on application-level data and task decomposition rather than device-level architectural details and device scheduling.

Acknowledgment

This work was supported in part by the DOE contract DE-AC02-06CH11357, DOE GTO via grant EE0002758 from Fugro Consultants, VT College of Engineering SCHEV grant, NSF grants CNS-0960081, an NVIDIA Graduate Fellowship, NVIDIA Professor Partnership, and CUDA Research Center at Virginia Tech. This research used the HokieSpeed heterogeneous computing resource at Virginia Tech, which is supported by the National Science Foundation under contract CNS-0960081. We thank Kaixi Hou for his efforts in porting the seismology simulation code to OpenCL.

References

[1] A. Munshi, The OpenCL Specification, 22, Khronos OpenCL Working Group, 2014.
[2] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, J. Lee, SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters, in: ACM International Conference on Supercomputing, 2012.
[3] M. Daga, T. Scogland, W. Feng, Architecture-aware mapping and optimization on a 1600-core GPU, in: 17th IEEE International Conference on Parallel and Distributed Systems, 2011.
[4] Y. Wen, Z. Wang, M. O'Boyle, Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms, in: International Conference on High Performance Computing (HiPC), 2014.
[5] V.T. Ravi, M. Becchi, W. Jiang, G. Agrawal, S. Chakradhar, Scheduling concurrent applications on a cluster of CPU–GPU nodes, Future Gener. Comput. Syst. 29 (8) (2013) 2262–2271.
[6] A.J. Peña, Virtualization of Accelerators in High Performance Clusters, Universitat Jaume I, 2013 Ph.D. thesis.
[7] A.J. Peña, C. Reaño, F. Silla, R. Mayo, E.S. Quintana-Orti, J. Duato, A complete and efficient CUDA-sharing solution for HPC clusters, Parallel Comput. 40 (10) (2014) 574–588.
[8] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, W. Feng, VOCL: an optimized environment for transparent virtualization of graphics processing units, Innovative Parallel Computing, 2012.
[9] S. Iserte, A. Castelló, R. Mayo, E.S. Quintana-Orti, F. Silla, J. Duato, C. Reano, J. Prades, SLURM support for remote GPU virtualization: implementation and performance study, International Symposium on Computer Architecture and High Performance Computing, 2014.
[10] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, W. Feng, Transparent accelerator migration in a virtualized GPU environment, in: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2012, pp. 124–131.

[11] P. Lama, Y. Li, A.M. Aji, P. Balaji, J. Dinan, S. Xiao, Y. Zhang, W. Feng, R. Thakur, X. Zhou, pVOCL: power-aware dynamic placement and migration in virtualized GPU environments, in: International Conference on Distributed Computing Systems (ICDCS), 2013, pp. 145–154.
[12] C.S. de la Lama, P. Toharia, J.L. Bosque, O.D. Robles, Static multi-device load balancing for OpenCL, in: 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 675–682.
[13] J. Kim, H. Kim, J.H. Lee, J. Lee, Achieving a single compute device image in OpenCL for multiple GPUs, 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), 2011.
[14] P. Pandit, R. Govindarajan, Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices, International Symposium on Code Generation and Optimization (CGO), 2014.
[15] K. Spafford, J. Meredith, J. Vetter, Maestro: data orchestration and tuning for OpenCL devices, Euro-Par – Parallel Processing, 2010.
[16] C. Luk, S. Hong, H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, International Symposium on Microarchitecture, 2009.
[17] S. Henry, A. Denis, D. Barthou, M. Counilh, R. Namyst, Toward OpenCL automatic multi-device support, in: Euro-Par – Parallel Processing, Springer, 2014, pp. 776–787.
[18] A.M. Aji, A.J. Peña, P. Balaji, W. Feng, Automatic command queue scheduling for task-parallel workloads in OpenCL, IEEE Cluster, 2015.
[19] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010.
[20] L.S. Panwar, A.M. Aji, J. Meng, P. Balaji, W. Feng, Online performance projection for clusters with heterogeneous GPUs, in: International Conf. on Parallel and Distributed Systems (ICPADS), 2013.
[21] A.J. Peña, S.R. Alam, Evaluation of inter-and intra-node data transfer efficiencies between GPU devices and their impact on scalable applications, in: The 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2013, pp. 144–151.
[22] J. Duato, A.J. Peña, F. Silla, R. Mayo, E.S. Quintana-Orti, Modeling the CUDA remoting virtualization behaviour in high performance networks, First Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.
[23] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, et al., The NAS parallel benchmarks, Int. J. High Perf. Comput. Appl. 5 (3) (1991) 63–73.
[24] S. Seo, G. Jo, J. Lee, Performance characterization of the NAS Parallel Benchmarks in OpenCL, IEEE Intl. Symposium on Workload Characterization (IISWC), 2011.
[25] S. Ma, P. Liu, Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite-element methods, Bull. Seismol. Soc. Am. 96 (5) (2006) 1779–1794.
[26] Pengcheng Liu, DISFD in Fortran, (Personal Copy).
[27] Kaixi Hou, FDM-Seismology in OpenCL, (Personal Copy).
[28] A. Aji, L. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. Bisset, J. Dinan, W. Feng, J. Mellor-Crummey, X. Ma, R. Thakur, MPI-ACC: accelerator-aware MPI for scientific applications, IEEE Trans. Parallel Distr. Syst. PP (99) (2015) 1.
[29] P. Balaji, W. Bland, W. Gropp, R. Latham, H. Lu, A.J. Peña, K. Raffenetti, S. Seo, R. Thakur, J. Zhang, MPICH User's Guide, Argonne National Laboratory, 2014.

Ashwin M. Aji received his Ph.D. and M.S. degrees from Dept. of Comp. Science, Virginia Tech, and is now a post-doc at AMD Research.

Antonio. J. Peña received his Ph.D. from the Jaume I University of Castellón, Spain, and is a postdoctoral fellow at Argonne National Laboratory.

Pavan Balaji is a computer scientist at Argonne National Laboratory, a Fellow of the Northwestern-Argonne Inst. of Science and Engineering at Northwestern University, and a Research Fellow of the Computation Inst. at the Univ. of Chicago.

Wu-chun Feng is a professor and Elizabeth & James E. Turner Fellow in the Dept. of Comp. Science, Dept. of Electrical and Computer Engineering, and Virginia Bioinformatics Institute at Virginia Tech.