Toward Supporting Multi-GPU Targets via Taskloop and User-Defined Schedules

Vivek Kale[1](✉), Wenbin Lu[2], Anthony Curtis[2], Abid M. Malik[1], Barbara Chapman[1,2], and Oscar Hernandez[3]

[1] Brookhaven National Laboratory, Upton, NY 11973, USA
vkale@bnl.gov
[2] Stony Brook University, Stony Brook, NY 11794, USA
[3] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA

Abstract. Many modern supercomputers such as ORNL's Summit, LLNL's Sierra, and LBL's upcoming Perlmutter offer or will offer multiple, e.g., 4 to 8, GPUs per node for running computational science and engineering applications. One should expect an application to achieve speedup using multiple GPUs on a node of a supercomputer over a single GPU of the node, in particular an application that is embarrassingly parallel and load imbalanced, such as AutoDock, QMCPACK and DMRG++. OpenMP is a popular model used to run applications on heterogeneous devices of a node and OpenMP 5.x provides rich features for tasking and GPU offloading. However, OpenMP doesn't provide significant support for running application code on multiple GPUs efficiently, in particular for the aforementioned applications. We provide different OpenMP task-to-GPU scheduling strategies that help distribute an application's work across GPUs on a node for efficient parallel GPU execution. Our solution involves using OpenMP's construct taskloop to generate OpenMP tasks containing target regions for OpenMP threads, and then having OpenMP threads assign those tasks to GPUs on a node through a schedule specified by the application programmer. We analyze the performance of our solution using a small benchmark code representative of the aforementioned applications. Our solution improves performance over a standard baseline assignment of tasks to GPUs by up to 57.2%. Further, based on our results, we suggest OpenMP extensions that could help an application programmer have his or her application run on multiple GPUs per node efficiently.

© Springer Nature Switzerland AG 2020
K. Milfeld et al. (Eds.): IWOMP 2020, LNCS 12295, pp. 295–309, 2020.
https://doi.org/10.1007/978-3-030-58144-2_19

Keywords: OpenMP · Tasks · Offload · Multi GPUs · AutoDock · Load balancing · User-defined scheduling · High-performance · Parallel

1 Introduction

Modern supercomputers for running computational science and engineering applications are often comprised of compute nodes with accelerator devices. Many of the supercomputers ranked on the latest Top500 list (http://www.top500.org) and the Green500 list (http://www.green500.org) in November 2019 are equipped with an accelerator component such as the NVIDIA GPU. A compute node in such a supercomputer often has multiple accelerators to further improve the power-to-performance ratio. For example, ORNL's supercomputer Summit has 6 NVIDIA Volta-100 GPUs per compute node [5], and LBL's Perlmutter will have nodes with 4 NVIDIA A100 GPUs and 2 AMD Milan GPUs [4]. This node architecture allows application programs to offload multiple computational kernels onto independent devices simultaneously and achieve significant performance. Supercomputers with such a node architecture are often considered as a leading candidate for running applications, in particular those applications that are embarrassingly parallel and load imbalanced, and employing Monte Carlo algorithms [3,26].

Accelerator programming languages such as CUDA, OpenCL, OpenMP, and OpenACC are often used to offload kernels to devices [28]. OpenMP is one of the most commonly used parallel programming models for on-node programming. The current version of OpenMP, OpenMP 5.0, provides GPU offloading support [1]. However, multi-GPU OpenMP offload is limited by mapping a target region for an accelerator to a specified device number of one of multiple devices on a node explicitly when a target region is created. How can we create suitable software for applications to take advantage of multiple GPUs on a node in a generic way, without mapping to specific devices, to leverage multiple GPUs on the node and improve performance portability on systems with a different number of devices per node?

This paper explores how to program multiple GPUs within a node by looking at different task-to-GPU scheduling strategies to map computations to multiple devices. Our solution involves using OpenMP's tasking construct `taskloop` to generate OpenMP tasks containing target regions for OpenMP threads, and then having OpenMP threads assign or schedule those tasks to GPUs on a node through a schedule specified by the application programmer, or a user such as a performance engineer helping optimize an application. We analyze the performance of our solution using a small OpenMP performance benchmark code representative of the applications with Monte Carlo methods, in particular AutoDock [15] and DMRG++ [13]. Applying our solution to our benchmark, we improve performance over a standard baseline assignment of tasks to GPUs up to 57.2%. Further, based on our results, we suggest OpenMP extensions that could help application programmers have their applications use multiple GPUs per node efficiently through OpenMP. We make the following contributions in this work:

1. OpenMP task-to-GPU scheduling strategies that help distribute an application's computations across GPUs on a node, which provide significant performance benefit over basic or naive approaches of assigning computations to GPUs;
2. a framework for developing user-defined task-to-GPU scheduling strategies;
3. OpenMP extension proposals to support our approach of programming multiple accelerators within a single node through taskloop and user-defined schedules.

2 Motivation Through Use Case Applications

Distributing work and data across multiple GPUs on a node is challenging. We have discovered applications that require work decomposition, often exemplified by the computational motif, or pattern, of Monte Carlo Methods [3,26], to be straightforward to map to multiple devices get significant performance benefit from doing so. Below is a summary of applications that can benefit of a task decomposition approach to deal with load imbalances across multiple GPUs. We focus the first application due to its timeliness.

2.1 Autodock

In 2019, the emergence of a novel coronavirus (SARS-CoV-2) caused the Coronavirus Disease 2019 (COVID-19). This virus has become a major threat worldwide due to its highly contagious nature. Molecular docking is one of the important steps used in identifying candidate drugs against a virus like SARS-CoV-2.

AutoDock is a family of applications that perform this kind of docking. AutoDock 4 [15] is the sequential version that is the baseline for improvements due to parallelism. AutoDock Vina [24] achieves approximately two orders of magnitude speed-up over AutoDock 4 through threaded parallelism, while also significantly improving the accuracy of binding predictions. AutoDock 4.2 [22], or Autodock-GPU, is based on OpenCL and simulates the molecular docking process by predicting the ligand-receptor interactions. It uses a Lamarckian Genetic Algorithm (LGA) to perform docking by offloading independent LGA executions to a GPU.

During the docking of the receptor protein and the ligand molecule, Autodock-GPU searches for a pose that has a satisfyingly low energy state, which will be predicted by a scoring function. This is achieved by searching in the space of the receptor-ligand pair's conformational coordinates (position, orientation, and torsion), using LGA. The search stops automatically after either reaching a small standard deviation of the current best pose, a large number of generations, or a large number of scoring function evaluations, whichever comes first.

In Autodock-GPU, the receptors are rigid and are modeled by static grid maps. This limits the sizes of the search spaces, and makes the number of rotatable bounds in the ligand one of the most influential factors on the search difficulty. Due to the randomness of the genetic algorithm, experiments have shown that the larger search space is correlated with higher variations in the time it takes to find a good pose for a given receptor-ligand pair.

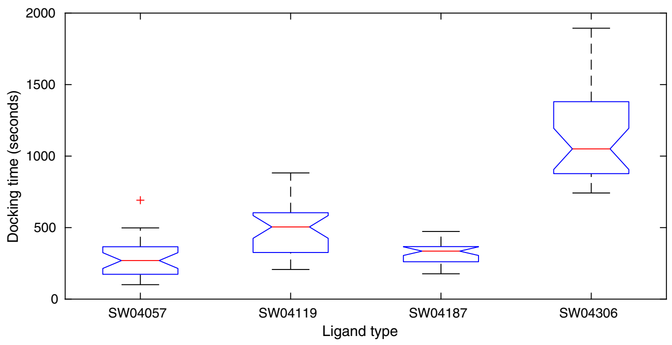


Fig. 1. Variation in docking time when running the OpenCL version of Autodock-GPU on NVIDIA Tesla V100, with local-search method ADADELTA.

We benchmarked Autodock-GPU on an NVIDIA Tesla V100 GPU, and recorded the docking times for four different ligands on the same receptor, 30 runs for each pair. In the results presented in Fig. 1, large variations are observed: the difference in run time for the same pair can vary from two times to more than four times. Similar results can be observed for ligands with fewer amounts of rotatable bounds, but with smaller variations.

Table 1. Mean, standard deviation and coefficient of variation of the docking time of the tested ligands. The result of ligand SW04057 includes several outliers that were not plotted in Fig. 1.

Ligand	Rotatable bounds	μ	σ	Coefficient of variation
SW04057	36	531.51 s	818.66 s	1.54
SW04119	22	494.38 s	185.39 s	0.37
SW04187	17	324.31 s	67.518 s	0.21
SW04306	36	1131.1 s	300.50 s	0.27

When docking is performed in a multi-GPU setup, each GPU typically processes its own set of receptor-ligand pairs independently. Although this process is embarrassingly parallel, the distribution of the docking times shown in Table 1 suggests that we should expect variations of at least 20% of the mean docking time. In the absence of a load balancing scheduler that distributes the receptor-ligand pairs dynamically, the work could become unevenly distributed and thus result in inefficient utilization of the GPUs.

2.2 DMRG++

Density Matrix Renormalization Group (DMRG++) [13] is a condensed matter physics code which is used to study superconductive properties of materials. One of the main computations of the application is a Hamiltonian matrix-vector multiplication, where the elements of the Hamiltonian matrix contain vectors of different sizes with symmetrical values to the diagonal part of the matrix that are only known at runtime. The Hamiltonian matrix-vector operation can be significantly optimized through using `taskloop` as it's a sparse matrix-vector multiplication and using OpenMP target regions to accelerate the inner matrix-vector multiplications of each of element on multiple GPUs.

2.3 Formulating Our Problem with a Representative Benchmark

We develop an OpenMP benchmark kernel code in C that represents the aforementioned applications' computational pattern and that identifies their performance bottlenecks[1]. Through this benchmark code, we apply the technique that we want to experiment with. The benchmark code takes as input (1) a maximum problem dimension n of each computation and (2) the number of such computations C. The benchmark's work is to perform a set of C square rooted vector multiplications, each of which are on vectors of sizes chosen randomly from the set $\{1, 4, 9, \ldots, n * n\}$. The vector sizes are randomly generated and stored in an array of integers before the computations start. Each of these C computations resemble the multiplication of two matrices each of dimension n for protein-ligand docking pairs in Autodock. The matrix dimensions of the matrix multiplications in Autodock are of a variety of sizes and are generated at runtime.

We augment the benchmark kernel with OpenMP offload features as follows. OpenMP threads, each of which run on a core of a multi-core CPU, first randomly choose the vector sizes. Then, each OpenMP thread offloads its prepared work of the computations to a GPU. This offload is performed by enclosing the benchmark's computation in an OpenMP target region. Within each OpenMP task of computation, we allocate the same amount of data and the same data for running the computation on a GPU. The amount of data movement between CPU and GPU across target regions is uniform. This uniform amount of data movement is representative of the CPU-GPU data movement in the Autodock application code.

With this augmented benchmark kernel code, we make the following observations. First, each vector multiplication is independent of the other, making this

[1] A repository for the benchmark code, which includes the strategies in this paper, is accessible at https://tinyurl.com/omp-ad-bench.

computation embarrassingly parallel. Second, the code isn't using the remaining GPUs on the node. Doing so could significantly speed up the code's execution, especially considering the baseline performance numbers shown in Sect. 2.1. Third, even if the code did use all the GPUs on the node, the code wouldn't use the GPUs efficiently due to load imbalance caused by the differently sized computations, in particular given our observations from Sect. 2.1. Given these observations, our objective is to have an application code use all computational power of the node, specifically the GPUs, all the time, given the load imbalance due to the high standard deviations of the timings across the computations. The next section covers how we try to meet the objective of using all of the GPUs all the time.

3 Using OpenMP Offload on Multiple GPUs Efficiently

The key idea of the solution is to have OpenMP threads generate work in well-defined and standard units and then have one or more OpenMP threads work together to dynamically map these units of work to GPUs. This section explains our solutions and the baseline that we compare our solution to.

A basic way to run OpenMP offload code on multiple GPUs is by pre-assigning each target region of computational work of the application to a device, i.e., GPU, ID [21]. To run a set of 100 computations of our benchmark on nodes with 6 GPUs, we can have an OpenMP thread assign the first 17 computations to GPU 0, the next 17 to GPU 1, and so on. When running T computations on a node of G GPUs, an OpenMP thread assigns the x^{th} computation to device ID $\lceil \frac{x*G}{T} \rceil$ through adding the clause `device(x*G/T)` to the `target` construct. We call this strategy *compact*, and it is our baseline strategy.

Through a static assignment of computations to GPUs described in the previous paragraph, the benchmark code and application codes of Sect. 2.3 can have load imbalance across, and an under-utilization of, the GPUs of a node. The benchmark and application codes can utilize the GPUs more efficiently if the computations are assigned dynamically to GPUs during the application's execution [10,12,17]. To assign, or schedule, computations to GPUs dynamically, we must find a way to encapsulate the computations in standardized units of work that can be managed by the OpenMP threads to distribute to the GPUs. We use the OpenMP tasking support already available in OpenMP for this purpose.

Figure 2 illustrates the dynamic OpenMP task-to-GPU scheduling strategy, showing how OpenMP threads on a CPU manage and schedule an OpenMP task to some GPU in the set of GPUs on a node. A `taskloop` construct is applied to the loop that performs the computation in independent outer iterations, each of which contains a target region. The red trapezoids in the figure are tasks generated from the `taskloop` construct, and the grey rectangle represents the queue of `taskloop`. Each OpenMP thread on the CPU offloads a task of computation in `taskloop` to a particular GPU by dequeueing the next available GPU from a GPU queue, which is stored on the host. This GPU queue does not perform cross-GPU synchronizations, thus avoiding GPU-to-GPU communication before each execution of a task.

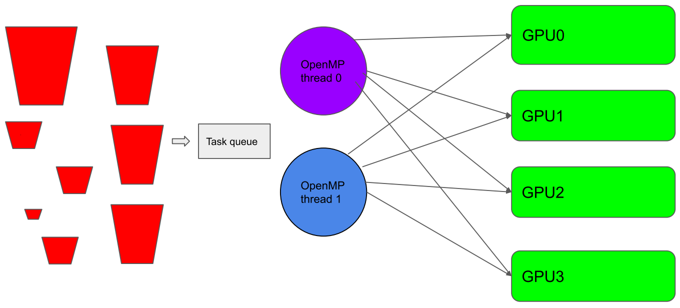


Fig. 2. Conceptual diagram of OpenMP threads scheduling tasks to GPUs.

Figure 3 shows the implementation strategy for our task-to-GPU scheduling technique. We wrap each OpenMP target region in an OpenMP task, as mentioned in the explanation of Fig. 2, and create a preceding and succeeding task for management of computational tasks for the GPU queue. These three OpenMP tasks are executed within each task in the `taskloop`. OpenMP threads of the parallel region assign the computational task of each task of the `taskloop` to GPUs through the function *gpu_scheduler_dyn()*. The function `doWork()` is the function for doing the square rooted vector multiplication computation in our benchmark.

Figure 4 shows the code change to a generic application for using the task-to-GPU dynamic scheduling strategy, which we implemented in the application code. An OpenMP thread running on the CPU invokes this function, and then waits in the `while` loop looking for an available GPU by repeatedly going through the array `occupancies` for the GPUs. Note that all the OpenMP threads are cycling through the same array `occupancies`, so atomic operations are used to avoid any data races/locks. If a GPU is busy, the thread just moves on and checks the next GPU. The thread keeps doing this until it sees the first GPU with occupancy of zero in the GPU queue. Other strategies, which may provide better load balance along with low overhead of data movement or coordination [11,14], can be defined and used by programmers or application developers alike.

4 Results

The benchmark was experimented with on SeaWulf, a cluster at Stony Brook University. We chose SeaWulf because it is representative of some of the modern supercomputers with multiple GPUs per node such as Summit, Sierra and the upcoming machine Perlmutter at NERSC, which run the applications discussed in Sect. 2. Also, the cluster was readily available for our experiments. SeaWulf has nodes with 8 NVIDIA K80 GPUs with 12 GB memory and CUDA 10.0.130 and a 28-core Intel Xeon E5-2683 v3 CPU (2 × 14-core). We use the clang/LLVM OpenMP compiler, cloned from the GitHub master branch, commit hash

```
#pragma omp parallel
{
#pragma omp single
  {
#pragma omp taskloop shared(success)
    for (int i = 0; i < numTasks; i++) {
      const int dev = gpu_scheduler_dyn(occupancies, ndevs);
      output[i] = 0;
#pragma omp task depend(out : success[i])
      {
        success[i] = 0;
      }
#pragma omp task depend(inout : success[i])
      {
#pragma omp target device(dev)                                  \
        map(to: a[0:arrSize], b[0:arrSize], c[0:arrSize])   \
        map(tofrom: success[i:1], output[i:1], taskWork[i:1],
     occupancies[dev:1])
        {
          devices[dev]++;
          if (taskWork[i] > probSize) taskWork[i] = probSize;
          const int NN = taskWork[i];
          output[i] = doWork(c, a, b, taskWork[i]);
          success[i] = 1;
        }
      }
#pragma omp task depend(in : success[i])
      {
#pragma omp atomic
        occupancies[dev]--;
      }
    }
  }
}
```

Fig. 3. Implementation for task-to-GPU scheduling.

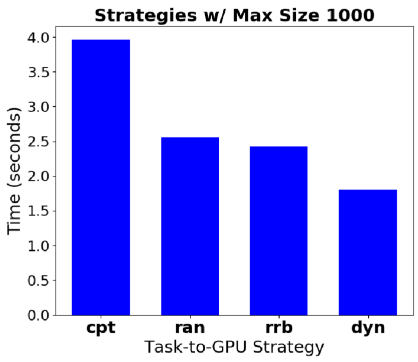
86e3abc9. We use clang/LLVM due to its support of OpenMP features for tasking and devices. In performing our experiments, we aim to answer the question of whether sophisticated task-to-GPU scheduling on a node with multiple GPUs provides a performance benefit over the baseline approach of statically assigning tasks to GPUs.

In our experiments, we show results for four strategies of assigning OpenMP target regions to GPUs. We show the *compact* strategy which involves a straightforward static assignment of target regions to GPUs, as discussed in Sect. 3. We show a *round-robin* task-to-GPU scheduling strategy that has OpenMP threads of the `taskloop` assigning tasks to GPUs in a round-robin fashion, with a scheduler function named `gpu_scheduler_rrb()` returning `taskID % ngpus`, where `taskID` is the task number taken as another input parameter

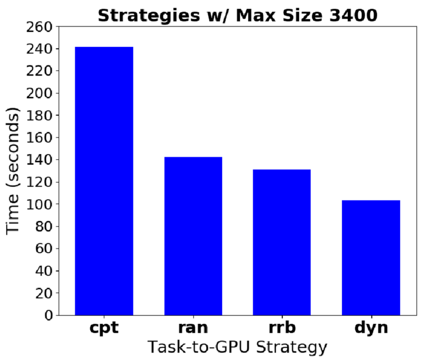
```c
unsigned gpu_scheduler_dyn(unsigned *occupancies, int ngpus)
{
  short looking = 1;
  unsigned chosen;
  while (looking) {
    for (unsigned i = 0; i < ngpus; i++) {
      unsigned occ_i;
#pragma omp atomic read
      occ_i = occupancies[i];
      if (occ_i == 0) {
        chosen = i;
#pragma omp atomic
        occupancies[chosen]++;
        looking = 0;
        break;
      }
    }
  }
  return chosen;
}
```

Fig. 4. Implementation of user-defined task-to-GPU schedule.

by the scheduler and ngpus is the number of GPUs on a node. Additionally, we show a *random* scheduling strategy in which an OpenMP thread assigns a task to a GPU by choosing a GPU randomly, with a scheduler function named gpu_scheduler_ran() returning rand() % ngpus. Finally, we show the *dynamic* scheduling strategy described through Fig. 4.



(a) Max problem size 1000.



(b) Max problem size 3400.

Fig. 5. Execution times for different task-to-GPU scheduling strategies on 8 GPUs of a node of SeaWulf.

4.1 Impact of Task-to-GPU Scheduling Strategies

We assess the impact on performance of the four different task-to-GPU assignment strategies we designed, by applying them to the benchmark code presented in Sect. 3.

Figure 5a shows results of the benchmark for 500 target regions for vector sizes from 1 to 1000^2 and run using the 8 GPUs on a SeaWulf node. We see that when we use *random*, the execution time reduces significantly from 3.96 s to 2.54 s, an improvement of 35.49%. Compared to *random*, *round robin* offers a comparable but larger improvement, 38.89% relative to compact (the baseline). When we use *dynamic*, the time decreases further, providing the best performance improvement over our baseline, 54.61%.

Figure 5b shows the results of the same square root vector addition benchmark on the same platform, this time with 500 target regions for various vectors between size 1 and 3400^2. Here, the *round-robin* strategy provides a 45.8% improvement (reducing from the baseline of 241.22 s to 130.76 s). When we use *dynamic*, the time decreases significantly, improving performance by 57.2%.

From these results, we make a few observations. First, for both problem sizes, a large amount of the performance gains come from using a task-based approach in which OpenMP threads distribute tasks across GPUs through the sophisticated task-to-GPU scheduling strategies, specifically, *random*, *round-robin* and *dynamic*. Second, *round-robin* performs slightly better than *random*, showing that a calculated and predefined strategy rather than a randomized strategy is important when using tasking, though it isn't tremendously significant. Third, *dynamic* shows a more pronounced benefit over *random* and *round-robin*, telling us that a dynamic task-to-GPU scheduling strategy, in particular one which is carefully implemented to maintain low coordination overhead, can provide a significant performance benefit.

4.2 Detailed Profiling

To understand utilization of all GPUs and overhead of the runtime and task-to-GPU scheduler, we did manual instrumentation with the CUDA Profiling Tools Interface (CUPTI), specifically through using the SOLLVE V&V suite's [2] timing implementation and interface for CUPTI. Through CUPTI, CUDA invokes user-defined callbacks to record start/finish timestamps of various events. There are many high-level activity categories: DEVICE, CONTEXT, DRIVER, RUNTIME, MEMCPY, MEMSET, KERNEL, OVERHEAD. We inserted a timing function from the interface at the beginning and at the end of the task of the target region. To understand GPU utilization and overhead, we focus on the DRIVER, OVERHEAD, and MEMCPY activity, by searching the resulting output of the SOLLVE V&V interface. We experiment with just the smaller of the two problem sizes, as for the larger of the two problem sizes, CUPTI created large amounts of overhead, making it difficult for us to understand the benefits of our approach.

Table 2 shows the timings, in nanoseconds, obtained through the CUPTI Activity API. We see that the DRIVER operations, which includes CUDA context/stream activities and synchronizations, dominate the execution time, and this is also the main source of reduction in execution time, as timings in this category are orders of magnitude larger than those in other categories. There are some variations in the MEMCPY category, which could be the result of combining GPU-to-socket locality issues and our locality-unaware schedulers. The OVERHEAD category captures the driver compiler's activity, buffer flush overhead and the instrumentation overhead. We have yet to identify the source of reductions in this category due to perturbation from the instrumentation itself. From these timings, we see that our sophisticated OpenMP task-to-GPU scheduling strategies are relevant not just from a standpoint of load balancing, but also from a standpoint of GPU resource management and reducing overhead.

Table 2. Execution activity breakdown using CUPTI (nanoseconds)

Scheduler	DRIVER	MEMCPY	OVERHEAD
Compact	23654161998	7393960	189489499
RoundRobin	16634552748	8526436	248582086
Random	17742577658	8085590	208329746
Dynamic	14229831142	6374549	171728914

5 Discussion on Results and Proposed Extensions

A key question that arises from our implementation and results is how we extend OpenMP to support of task scheduling for multi-GPUs, for ease of use by application programmers. There are different aspects to this, and we cover them in this section.

First, we need support for a single OpenMP construct to offload asynchronous target regions on multiple devices. In our implementation, we partition work across GPUs by leveraging the `taskloop` construct, associating one thread with a task of `taskloop` and then having that thread assign the task to a GPU. We used `taskloop` in our implementation for the strategy because we know through the OpenMP Community [1] that there is a potential that tasking constructs and target constructs will be unified. However, right now, a problem exists with executing each of the tasks on any of the GPUs and keeping track of the correct device contexts used by the OpenMP tasks that offload the different target regions on multiple GPUs. For example, if a CPU task is scheduled a different thread, then after a target region with a `nowait` clause is offloaded, there could be issues with the GPU contexts when running on multiple devices. We found this issue with some OpenMP implementations, where we had to comment out the `nowait` clause on the `target` construct to make our benchmark work on multiple devices. We are aware of ongoing efforts in the LLVM OpenMP community [6] to improve the support of `nowait` with tasks, in particular in the context of multiple GPUs. We expect that our results can further improve with these improvements in the OpenMP library implementation.

Second, our approach could have less overhead, and could also require less programmer effort, if OpenMP's `taskloop` construct was extended so that it could handle the scheduling of target regions to GPUs and avoid additional levels of nested tasks. In our current implementation, we have two levels of tasking. The lower level is for the target and the higher level is for coordination of the tasks of `taskloop`. We could reduce the possible overhead and reduce the programming complexity by eliminating one of these two levels. A possible extension of `taskloop` is to create a `target taskloop` construct that will automatically manage the assignment of tasks to GPUs using `grainsize` or `num_tasks` as scheduling strategies supported by OpenMP implementations and that can be specified in the construct. We may also want to extend the `taskloop` construct to support pipelines where the body of the `taskloop` can contain dependent tasks that execute on both the CPU and GPU.

Third, in this work, we showed the need for a specialized task-to-GPU scheduling strategy using atomics. Such a specialized strategy can be implemented with an OpenMP user-defined task-to-GPU schedule in a similar fashion as has been proposed in [18]. The user could define her task-to-GPU schedule in an application by implementing a function `gpu_scheduler_X()`, with a pointer to a record as a parameter to the function. The user could then specify the schedule `X` in a clause of `taskloop`. We note that better atomic instructions such as compare-and-swap can also help make developing such schedules easier.

Lastly, our approach can benefit from an `affinity` clause for `taskloop` which could work hand-in-hand with the proposed user-defined task-to-GPU schedules. The affinity clause would reduce data movement from CPU to GPU, and significantly improve performance when data has been mapped to a specific device. We need to assess various design issues given the application for this kind of affinity which include a study of task-to-data, task-to-device and thread-to-device affinity. For example, for Autodock, the tasks doing docking for a ligand will be assigned to the GPU on which that data already resides through the affinity clause hint, which would allow for improved locality through reuse of the data.

6 Related Work

Existing accelerator programming libraries support only a single accelerator. However, several methods deal with multiple accelerators by using the libraries together with parallel programming libraries for hosts. One example method is to spawn multiple threads using OpenMP on host, and each thread deals with one accelerator [9,16,27]. Another is to spawn multiple processes using MPI on host, with each process dealing with one accelerator [7,27].

Xu et al. [28] propose an OpenACC extension to support multiple accelerators. Although the OpenACC extension supports communication between accelerators, dividing data and tasks manually is needed. Komoda et al. [20] propose another OpenACC extension that supports dividing data and tasks into multiple accelerators. Furthermore, its compiler has a mechanism to keep data consistency on the accelerator memory automatically. However, the OpenACC extension can be used only before the loop statement. So, the OpenACC extension cannot offload data to an arbitrary device, as in our work. Scogland [25] developed directive extensions to support scheduling work on multiple GPUs and multi-cores using the a runtime called coreTStar. The extension partitions loop iterations and its data across multiple devices and CPU threads.

Matsumura et al. [8] develop an OpenACC compiler system to generate an OpenACC code for multiple accelerators from an OpenACC code for a single accelerator automatically. However, there are some limitations. For example, a loop statement that can be divided is composed only of affine access. Nakao et el. [23] develop an XcalableACC directive-based language for accelerated clusters which gives an ability to use multiple accelerators on a single node. In contrast, our work allows for sophisticated scheduling strategies that the user to define within the application code.

7 Conclusions

In this work, we presented methods to use all GPUs of a node of an HPC cluster efficiently through OpenMP, particularly for applications that are embarrassingly parallel and load imbalanced, which are characteristics of the computational pattern of Monte Carlo Methods and exemplified by the applications Autodock and DMRG++. Our solution involves encapsulating each OpenMP target region containing a computation within an OpenMP task, and then having OpenMP threads assign the OpenMP tasks to GPUs on a node through a user-level task-to-GPU schedule. Through experimenting with our approach, our results provide up to a 57.2% performance improvement. Our results suggest the usefulness of OpenMP tasking across GPUs on a node.

Our technique focuses on scheduling tasks across GPUs rather than scheduling of Thread Blocks to Stream Multiprocessors (SMs), i.e., scheduling within a GPU. An extension to our approach that combines scheduling across GPUs and scheduling within GPUs will be written for future work. We will also incorporate our techniques within relevant application codes, e.g., Autodock, DMRG++ [13,19,22]. We will work to propose new extensions in OpenMP, particularly implementing them in the LLVM OpenMP compiler and supporting OpenMP implementations that allow users to easily use our approach. We will look at the impact of and tune the taskloop's grain-size. Finally, we will look at using or adapting the affinity clause to easily reduce data movement overheads of our task-to-GPU scheduling strategies. The affinity clause will give a hint to the task scheduler about placing a task on the most appropriate GPU based on the GPU context.

Acknowledgements. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy

Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVm for Exascale performance and portability (SOLLVE). It is also supported in part by NSF project 1409946 "Compute on Data Path". This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the high-performance SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492). We want to thank Jeremy Smith and Ada Sedova, from Oak Ridge National Laboratory, for providing a small sample of input sets for the Autodock-GPU experiments to help us study the application workload. We acknowledge the QMCPACK team at ORNL for discussing their code with respect to application load imbalances.

References

1. OpenMP 5.0 Reference Guide. https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-1119-01-TSK-web.pdf
2. OpenMP Verification and Validation Suite. https://github.com/SOLLVE/sollve_vv
3. Parallel Computational Pattern: Monte Carle Methods. https://patterns.eecs.berkeley.edu/?page_id=186
4. Perlmutter User Guide. https://www.nersc.gov/systems/perlmutter/
5. Summit User Guide. https://docs.olcf.ornl.gov/systems/summit_user_guide.html
6. The LLVM Compiler Infrastructure. http://llvm.org/
7. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication (2012)
8. Matsumura, K., Sato, M., Boku, T., Podobas, A., Matsuoka, S.: MACC: an OpenACC transpiler for automatic multi-GPU use. In: Yokota, R., Wu, W. (eds.) SCFA 2018. LNCS, vol. 10776, pp. 109–127. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-69953-0_7
9. Beyer, J., de Supinski, B.R.: IWOMP 2016 tutorial: OpenMP accelerator model (2016). http://iwomp2016.riken.jp/wp-content/uploads/2016/10/tutorial-accelerator.pdf
10. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1995)
11. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105, Lund, Sweden (1999)
12. Ciorba, F.M., Iwainsky, C., Buder, P.: OpenMP loop scheduling revisited: making a case for more schedules. ArXiv arxiv:1809.03188 (2018)
13. Criado, J., et al.: Optimization of condensed matter physics application with OpenMP tasking model. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 291–305. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_20

14. Donfack, S., Grigori, L., Gropp, W.D., Kale, V.: Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 496–507 (2012)
15. Huey, R., Morris, G.M., Olson, A.J., Goodsell, D.S.: A semiempirical free energy force field with charge-based desolvation. J. Comput. Chem. **28**, 1145–1152 (2007)
16. Guan, J., Yan, S., Jin, J.M.: An OpenMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems. IEEE Trans. Antennas Propag. **61**(7), 3607–3616 (2013)
17. Kalé, L., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Paepcke, A. (ed.) Proceedings of OOPSLA 1993, pp. 91–108. ACM Press (September 1993)
18. Kale, V., Iwainsky, C., Klemm, M., Müller Korndörfer, J.H., Ciorba, F.M.: Toward a standard interface for user-defined scheduling in OpenMP. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 186–200. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_13
19. Kim, J., et al.: QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. J. Phys.: Condens. Matter **30**(19), 195901 (2018). https://doi.org/10.1088/1361-648x/aab9c3
20. Komoda, T., Miwa, S., Nakamura, H., Maruyama, N.: Integrating multi-GPU execution in an OpenACC compiler. In: 2013 42nd International Conference on Parallel Processing, pp. 260–269 (2013)
21. Leopold Grinberg, C.B., Haque, R.: Hands on with openmp4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (Part ii) (2017)
22. Morris, G.M., et al.: Autodock4 and AutoDockTools4: automated docking with selective receptor flexibility. J. Comput. Chem. **30**(16), 2785–2791 (2009)
23. Nakao, M., Murai, H., Iwashita, H., Tabuchi, A., Boku, T., Sato, M.: Implementing lattice QCD application with XcalableACC language on accelerated cluster, pp. 429–438 (2017)
24. Trott, O., Olson, A.J.: AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading. J. Comput. Chem. **31**(2), 455–461 (2010)
25. Scogland, T.R.W., Feng, W., Rountree, B., de Supinski, B.R.: CoreTSAR: adaptive worksharing for heterogeneous systems. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 172–186. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_11
26. Tandon, P., Rosner, D.E.: Monte Carlo simulation of particle aggregation and simultaneous restructuring. J. Colloid Interface Sci. **213**(2), 273–286 (1999)
27. Wolfe, M.: Scaling OpenACC applications across multiple GPUs (2014)
28. Xu, R., Tian, X., Chandrasekaran, S., Chapman, B.: Multi-GPU support on single node using directive-based programming models (January 2016)