2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)

Static Graphs for Coding Productivity in OpenACC

Leonel Toledo*, Pedro Valero-Lara†, Jeffrey Vetter‡ and Antonio J. Peña§
*Barcelona Supercomputing Center (BSC) — Fundaciò i2CAT
Email: leonel.toledo@i2cat.net
†Oak Ridge National Laboratory
Email: valerolarap@ornl.gov
‡Oak Ridge National Laboratory
Email: vetter@ornl.gov
§Barcelona Supercomputing Center (BSC)
Email: antonio.pena@bsc.es

***Abstract*—The main contribution of this work is to increase the coding productivity for GPU programming by using the concept of Static Graphs. To do so, we have combined the new CUDA Graph API with the OpenACC programming model. We use as test cases a well-known and widely used problems in HPC and AI: the Particle Swarm Optimization. We complement the OpenACC functionality with the use of CUDA Graph, achieving accelerations of more than one order of magnitude, and a performance very close to a reference and optimized CUDA code. Finally, we propose a new specification to incorporate the concept of Static Graphs into the OpenACC specification.**

***Index Terms*—Coding Productivity, Tasking, Data Dependencies, Static Graph, OpenACC, Particle Swarm Optimization**

I. Introduction

It is undeniable that GPU capabilities have been increasing significantly in terms of performance and memory capacity. However, some applications are facing problems in terms of scalability and some algorithms seem to limit the amount of work that one GPU can perform at a single time [1]. This is mainly due to the assignment of hardware resources and the occupancy of the device, which makes it difficult to benefit from the whole GPU capacity. NVIDIA developed CUDA Graph API, as a potential solution to improve scalability. In CUDA Graph API, it is possible to represent the workflow as a graph, as an alternative for submitting kernels. These graphs are built from a series of operations that could range from kernel invocations to memory copies, as well as host code or calls to libraries, such as CUBLAS and CUSPARSE. Every call inside the graph is represented as a node, and each node is connected by dependencies.

The contribution of this work is to increase the coding productivity of GPU programming by using Static Graphs. As test cases, we use one very well-known and widely used algorithm in high-performance computing (HPC) and artificial intelligence (AI), the Particle Swarm Optimization. By combining OpenACC and CUDA Graph we are able to accelerate the OpenACC-only version and attain a very similar performance to the CUDA optimized code. To the best of our knowledge, this is the first time that CUDA Graph has been integrated with OpenACC and effectively adapted to the algorithm used as test case in this work: Particle Swarm Optimization. Finally, given the important benefits attained by using CUDA Graph, we propose new specifications into the OpenACC Standard, to introduce the concept of "static graph".

The rest of this document is organized as follows: Section II describes the OpenACC programming model and the most important concepts about the new CUDA API to implement Static Graphs. Section III presents a detailed analysis of the implementations and performance attained by using CUDA Graph in combination with OpenACC on the Particle Swarm Optimization algorithm. In Section V, we propose a new specification to use Static Graphs in the OpenACC Standard. Section VI presents the most relevant state–of–the–art references. Section VII concludes with the most important remarks and discusses future directions.

The first two authors contributed equally to this work.

II. Background

A. OpenACC

OpenACC is a high-level, directive-based programming model which supports C, C++ and Fortran. It was developed to allow programmers to interact with heterogeneous high-performance computing architectures without the effort that requires to fully understand all the low-level programming details and underlying hardware features [2]. This programming model allows developers to insert hints into their code that help the compiler interpret how to parallelize the code. In this way, the compiler is responsible for the transformation of the code to parallelize it, which is completely transparent to the programmer.

OpenACC defines a mechanism to offload programs to an accelerator in a heterogeneous system [3]. Because OpenACC is a directive-based programming model, the code can be compiled serially, ignoring the directives and still produce correct results, allowing a single code to be portable across different platforms [4].

This simple model allows non-expert programmers to easily develop code that benefits from accelerators [5]. Currently, OpenACC compilers support several platforms such as x86 multicore CPUs, accelerators (GPUs, FPGAs), OpenPOWER processors, KNLs and ARM processors.

One example that summarizes the advantages of using OpenACC is the the work of [6], which evaluates the use of OpenACC, OpenCL and CUDA in terms of performance, productivity, and portability. This work concludes that OpenACC is a robust programming model for accelerators while improving programmer productivity.

B. CUDA Graph API

Performance of GPU architectures continues increasing each generation. However, it is important to address that each kernel launch has an associated overhead regarding the submission of each operation to the GPU. These overheads are becoming more and more significant and can have a negative impact on performance [7]. Many current applications need to perform a large number of different operations to solve a given problem. Most the times these operations are involved in patterns that require many iterations, so this kind of overhead can produce significant performance degradation.

To address this issue, since CUDA 10.0, it is possible to represent the workflow as a graph. A graph consists of a series of operations such as memory copies and kernel launches, which are connected by dependencies. This feature allows developers to represent the work as a graph of nodes and create a static structure that may be launched at any time and be executed as many times as needed. The CUDA Graph API has two main advantages: First, the overhead of launching GPU operations, such as memory transfers or kernel executions, has no impact on performance, since the static structure which defines the graph, is submitted only once to the GPU. Second, we have the freedom to create the workflow to be submitted to the GPU. There may be operations which are completely independent from each other, so depending on the hardware capabilities, it is possible to overlap the execution of different nodes of the graph.

III. Particle Swarm Optimization

A. Algorithm

Particle Swarm Optimization (PSO) is an evolutionary computational technique originally developed by Kennedy and Eberhart [8]. The algorithm was developed as a simulation of a simplified social system, with the objective to simulate the behavior of bird flocks. This algorithm is also considered as an optimizer. This technique shares some similarities with genetic algorithms. For instance, the system is initialized with a random population that evaluates different sets of solutions. Every potential solution is considered as a particle within the search space of the problem. This means that each particle has its own set of parameters such as velocity, speed, acceleration, position and learning factors. Each particle keeps track of the values which are associated with the best solution, also known as fitness. This is an iterative algorithm, where in every iteration, the values (speed, position, etc.) of all the particles are evaluated and updated, with the target of moving each particle to locations that potentially have a better solution.

PSO is used by many applications; for instance, those problems that involve maximization or minimization [9], [10].

Listing 1. OpenACC PSO code

```
void main (){
  //Initialization
  initParticle();
  calculateFitness();
  updatePopulationBest();
  //Computation
  while(i<ITERATIONS){
    //findBestParticle kernel
    #pragma acc kernels deviceptr(inputData)
    for(int i=0; i<POPULATION; i++){
      findBestParticle(inputData...,);
    }
    //updateParticlePosition kernel
    #pragma acc kernels deviceptr(inputData)
    for(int i=0; i<POPULATION; i++){
      updateParticlePosition(inputData...,);
    }
    //calculateFitness kernel
    #pragma acc kernels deviceptr(inputData)
    for(int i=0; i<POPULATION; i++){
      calculateFitness(inputData...,);
    }
    //updateBestPopulation kernel
    #pragma acc kernels deviceptr(inputData)
    for(int i=0; i<POPULATION; i++){
      updateBestPopulation(inputData...,);
    }
  }
}
```

PSO is robust enough to work with functions in a continuous, discrete or mixed search space, as well as multi-objective problems [11].

In this work we use PSO as case of study to test the impact of integrating static graphs on directive-based programming models. Due to the iterative design of the algorithm and its potential to parallelize several areas of the code, it is an extraordinary test bed to study and analyze the impact of the use of GPU static graphs.

It is important to mention that the target of this work is not to improve the PSO algorithm itself, but to study the impact of combining the different target programming models (OpenACC and CUDA Graph) in an application.

To measure the impact of the different approaches, we developed several versions of PSO. First, we studied a sequential version of the algorithm, which we use as a baseline. The code that we tested is shown in Listing 1.

We implemented the original version of the code based on the work of Kennedy et al. [9]. In the first step we initialize all particles by defining their position and velocities within the boundaries of the search space. Then, each particle evaluates the solution determined by its position to calculate its fitness. Next, we find the best fitness of the population and store its value. The following steps are computed in each iteration: i) calculate the position of the particle that has the best fitness, ii) compute the next position and iii) update each particle speed in the x, y and z axes. Afterwards, we calculate the fitness of the particles in their new position and finally we update the value

of the best particle. We repeat these steps as many iterations as needed, and finally the particle with the best fitness is selected as the best possible solution of the problem.

Given the stochastic nature of the problem, it is not guaranteed to find the optimum value all the times. However, sometimes an approximation is sufficient and it has the advantage of being much faster than a brute force search.

B. OpenACC and CUDA Graph Implementations

To attain additional acceleration, we involved the GPU for computing the most computationally intensive tasks in the algorithm, as well as avoiding having many small tasks that are constantly switching context and awaiting for some of them to finish. We use OpenACC for GPU parallelization (see Listing 1).

The modifications on the code from the CPU to the GPU architecture are minimum. However, there are important considerations that significantly impact the behavior of the code. The main difference is in the distribution of the work. Other significant factor is the use of Unified Memory. Managing memory between CPU and GPU is an important challenge. There are significant limitations, particularly concerning memory bandwidth, latency and GPU utilization [12]. To mitigate this issue, since CUDA 6.0 it is possible to use Unified Memory access. This provides a mechanism to simplify the GPU memory communication with the host while providing high bandwidth for data transfers at run-time. We also use Unified Memory for more readable code and coding productivity [13]. Doing this, the GPU and CPU memory communications can be kept hidden from the developer, so the programmer does not have to deal with the issues that arise from moving data, further enhancing coding productivity.

The OpenACC version can be efficiently integrated with CUDA Graph to minimize the overhead of creating and launching multiple kernels in every iteration. Although this overhead is measured at the scale of microseconds, this can degrade the performance considerably on long runs. Using CUDA Graph we can create a high-level representation of the workflow; in other words, we determine the topology of the graph by determining the order of the tasks that need to be executed in every iteration. We still use the OpenACC kernels as a node of the graph. The code presented in Listing 2 shows the changes that were made in order to combine both models, OpenACC and CUDA Graph.

CUDA Graph allows us to store the set of kernels to be computed (workflow) before being launched. In that way, it is possible to know in advance the amount of work that needs to be submitted to the GPU.

To attain that, we use acc_get_cuda_stream(acc_async_sync) and acc_set_cuda_stream(0, stream1). In that way, we ensure that CUDA Graph recognizes the streams used by OpenACC. Finally, this stream must be also known by the OpenACC async clause.

Using OpenACC and CUDA Graph, the stream creations are more efficient and execution is faster. This is due to the way that CUDA Graph launches the kernels to the GPU. All the kernels are treated as a whole instead of processing each of them individually. This reduces considerably the overhead when submitting multiple kernels to the GPU.

Finally, we also exploit the potential overlapping of those parts of the application that are independent and can be executed in parallel. These are the functions updateParticlePosition and calculateFitness. To do that, we need to use cudaEventRecord and cudaStreamWaitEvent.

Listing 2. OpenACC and CUDA Graph PSO code

```
int main (int argc, char *argv[]){
  cudaStream_t stream1, stream2, streamForGraph;
  cudaEvent_t event1, event2;
  cudaGraph_t graph;
  //Initialization
  initParticle();
  calculateFitness();
  updatePopulationBest();
  //Graph definition
  cudaStreamCreate(&stream1);
  cudaStreamCreate(&stream2);
  cudaStreamCreate(&streamForGraph);
  void* stream = acc_get_
       cuda_stream(acc_async_sync);
  acc_set_cuda_stream(0,stream1);
  cudaStreamBeginCapture(stream1 ,
       cudaStreamCaptureModeGlobal);
  //OpenACC Kernels
  findBestParticle(..., stream1);
  //Fork
  cudaEventRecord(event1, stream1);
  updateParticlePosition(..., stream1);
  calculateFitness(..., stream2);
  // Join
  cudaEventRecord(event2, stream2);
  cudaStreamWaitEvent(stream1, event2);
  updateBestPopulation(..., stream1);
  cudaStreamEndCapture(stream1 , &graph);
  cudaGraphExec_t graphExec;
  cudaGraphInstantiate(&graphExec, graph,
       NULL, NULL, 0);
  //Computation
  for (int i = 0; i < ITERATIONS; i++) {
    cudaGraphLaunch(graphExec, streamForGraph);
  }
  cudaStreamSynchronize(streamForGraph);
}
```

IV. Performance Analysis

We conduct the performance evaluation by using the following heterogeneous system: 2 x IBM POWER9 8335-GTH at 2.4GHz, 32GB RAM memory, and an NVIDIA V100 (Volta) GPU with 16GB HBM2 and NVLink2 for high-bandwidth communication between CPU and GPU. This architecture is similar to that used in the current top-2 (Summit at ORNL) and top-3 (Sierra at LLNL) fastest supercomputers in the TOP500 list today.

Table I shows the details of the seven functions used in our analysis. These are considered as standard benchmarks

TABLE I
DESCRIPTION OF THE EMPLOYED FUNCTIONS FOR TESTING PSO

Function	Formula
Sphere	$f(x) = \sum_{i=0}^{n} x_i^2$
De Jong	$f(x) = \sum_{i=0}^{n} i * x_i^4$
Griewank	$f(x) = \frac{1}{4000} \sum_{i=0}^{n} x^2 - \prod_{i=0}^{n} cos(\frac{x}{\sqrt{i}}) + 1$
Rastrigin	$f(x) = \sum_{i=0}^{n} [x^2 - 10cos(2\pi * x_i) + 10]$
Rosenbrock	$f(x) = \sum_{i=0}^{n} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$
Schaffer	$f(x) = \sum_{i=0}^{n-1} [0.5 + \frac{(sin\sqrt{x_i^2+x_{i+1}^2})-0.5}{(1+.001(x_i^2+x_{i+1}^2))^2}]$
Schaffer 2	$f(x) = (\sum_{i=0}^{n} x_i^2)^{.25} [1 + (50(\sum_{i=0}^{n} x^2)^{0.1})^2]$

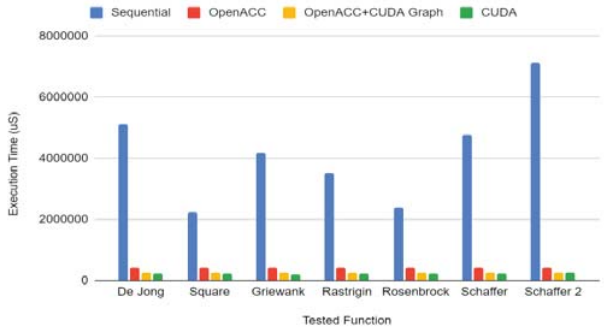

Fig. 1. Execution time (μs in logarithmic scale) comparison between sequential, OpenACC, OpenACC and CUDA Graph and CUDA implementations of the PSO algorithm.

for PSO. For the sake of simplicity, these formulas are simplified for 1D space; however, all the functions used in our experiments were implemented for a 3D space.

For the experiments, we use all the functions described in Table I on a simulated population of 1,000 particles. We execute all the simulations during 10,000 iterations.

Figure 1 illustrates the wall time (μs) of our test bed. As expected, the sequential version is the slowest. We see a much better performance by using the native OpenACC implementation. However, the hardware is used in a more efficient way with the combination of both programming models, OpenACC and CUDA Graph. This approach is able to attain an important reduction in the execution time (even more than one order of magnitude in some cases). In average, the speedup is ranged from $2\times$ to $4\times$.

It is important to highlight that our approach based on GPU static graphs (OpenACC and CUDA Graph) is very close to the optimized CUDA performance. The maximum performance difference between the optimized CUDA implementation and the OpenACC and the CUDA Graph counterpart is about 10%.

From this point on, we focus on the comparison between the OpenACC implementation and the use of GPU static graphs (CUDA Graph) as part of the OpenACC specification. The behavior illustrated in Figure 1 remains true in these new results, i.e., the sequential version continues being the slowest approach and the use of OpenACC and CUDA Graph is not farther away than 10% from the performance attained by the optimized CUDA code. We use the Schaffer 2 function (Table I) to test how the use of different settings may affect the behavior of our proposed model (GPU static graph). We decided to use this function because it is the most computationally expensive.

First, we analyze the impact of increasing the size of the population (number of particles). In the PSO algorithm, the larger the population, the larger the kernels (more threads are necessary). Figure 2 illustrates the impact of increasing the number of particles on execution time. In these tests, the number of iterations is 100. The use of OpenACC and CUDA Graph is able to achieve an acceleration of up to $3.5\times$. However, the larger the population, the lower the acceleration. This is expected because the chance to execute more than one kernel in parallel in the GPU is reduced by increasing the number of particles. When running larger kernels, the acceleration reached is about $1.2 - 1.3\times$.

Next, we analyze the impact on performance of increasing the number of iterations. In PSO, the larger the number of iterations, the more kernels need to be executed to finish the simulation. In the experiments, we use a population equal to 1,024 particles. The speedup reached in the previous experiments when using this size of population is equal to $1.7\times$. Figure 3 illustrates the execution time for different test cases using different number of iterations while keeping constant the size of the population. The use of CUDA Graph and OpenACC is able to keep the speedup (about $1.4-1.7\times$) for a population size equal to 1,024. A similar trend is reached when using different population sizes.

As we have seen along this section, it is possible to obtain an important acceleration by combining OpenACC with CUDA Graph, at the expense of a small losing of coding productivity, which is against the motivation behind the OpenACC Standard. To mitigate this limitation, in the next section we present a proposal to integrate the concept of static graph into the OpenACC syntax.

V. DIRECTIVE-BASED GPU STATIC GRAPH API PROPOSAL

In the previous sections, we were able to prove the efficiency of using GPU Static Graph (CUDA Graph) with CUDA and OpenACC. Although the performance was satisfactory, the integration of both programming models is not easy, harnessing OpenACC programming productivity.

It is important to highlight that the time to develop parallel solutions is a valuable factor to be considered along with portability. That is why it is so important to provide an efficient way to easily implement GPU codes while hiding

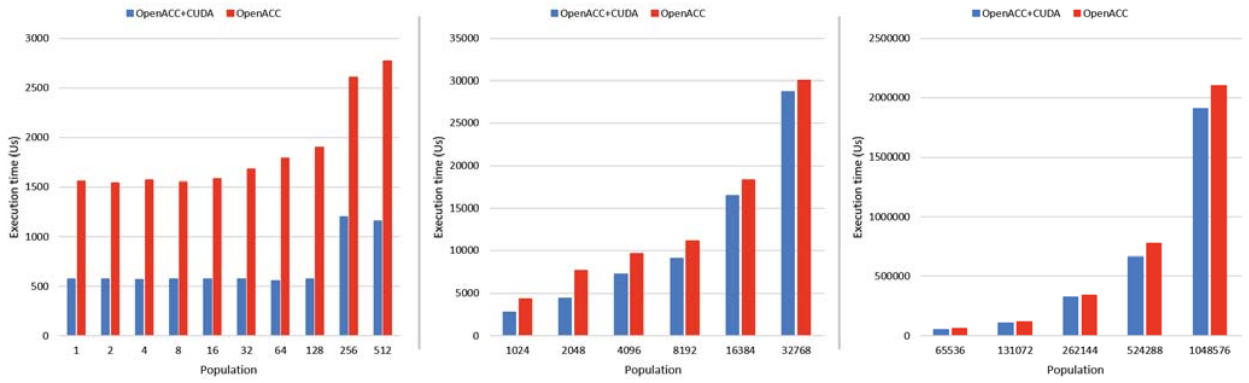

Fig. 2. Execution time (μs) increasing the size of the population and keeping constant the number of iterations (100).

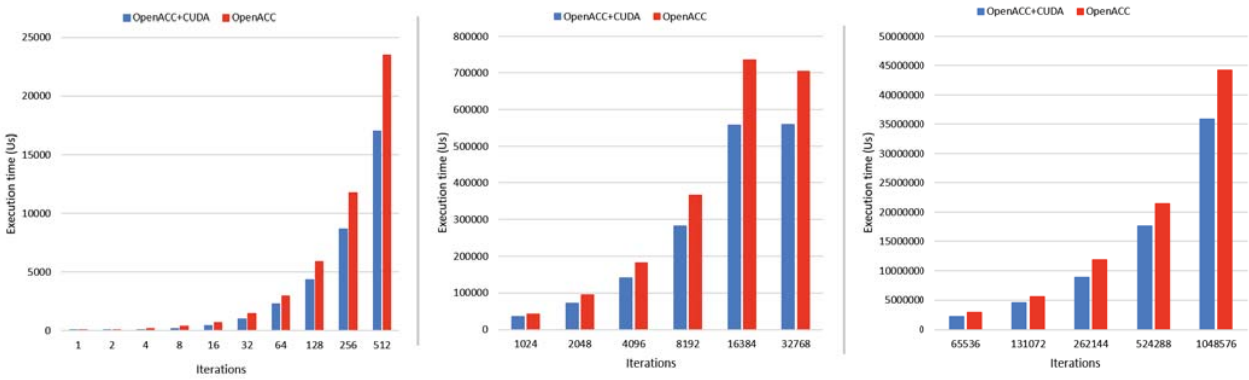

Fig. 3. Execution time (μs) increasing the number of iterations and keeping constant the size of the population (1,024 particles).

low-level hardware/software details, which are usually highly time-consuming.

In this section, we propose an new approach for developers to use static graphs within OpenACC. Our motivation is to provide a simple and easy to use directive which can annotate and define a workflow as a static graph. Listing 3 shows an example of the code we propose using *staticgraph pragma* for the PSO implementation.

The `staticgraph` clause is interpreted by the compiler to create a static graph which treats every OpenACC kernel as a CUDA Graph node. The topology of the workflow is recorded using CUDA Graph and an instance of this can be run for as many iterations as necessary (`accGraph_t`). This poses an important reduction in the number of lines of code and a substantial increase in terms of programming productivity, in comparison with the original CUDA Graph and OpenACC code (see Listing 2). We use Unified Memory to further simplify the code, and avoid complex transfers of data between device and host.

To exploit the potential overlapping among those parts of the application that can be executed in parallel, we need to use OpenACC Queues and the async clause, hence preventing the explicit handling of any CUDA constructs such as CUDA Streams.

These modifications proposed for the OpenACC specification provide developers with a robust and strong mechanism to easily translate iterative algorithms into static graphs. These graphs can be recorded prior to execution, which allows the runtime to be aware of the dependencies and the order of the execution. Once the topology is defined, then all the GPU work is handled as one single GPU launch by the driver, avoiding the overhead associated to deal with each of the kernels separately. As shown in previous sections, this yields significant benefits both in terms of performance and coding productivity.

VI. Related Work

Although GPU capacity has increased significantly, the scalability of algorithms and applications still faces important challenges [1]. One important problem regarding scalability is the hardware resource assignment. Some applications are limited to execute a single kernel in the GPU without benefiting from the whole capability of the device [14]–[16].

Other interesting examples are the task-based programming models using GPUs, such as StarPU [17] and OmpSs [18]. Both programming models propose a task-based API which allows tasks to be executed on GPUs and tune scheduling

Listing 3. OpenACC staticgraph model with kernels overlapping

```
int main (int argc, char *argv[]){
  accGraph_t graph;
  ...
  #pragma acc staticgraph(graph) \
              deviceptr(inputData) {
    //Enqueue
    #pragma acc kernels deviceptr(inputData) \
                async(1)
    findBestParticle(inputData...);
    //Fork & Enqueue
    #pragma acc kernels deviceptr(inputData) \
                async(1)
    updateParticlePosition(inputData...);
    #pragma acc kernels deviceptr(inputData) \
                async(2)
    fitnessBestParticle(inputData...);
    //Join & Enqueue
    #pragma acc kernels deviceptr(inputData) \
                async(1)
    updateBestPopulation(inputData...);
  } //End pragma
  for (int i = 0; i < ITERATIONS; i++) {
    #pragma acc launchstaticgraph(graph) \
              deviceptr(inputData)
  }
}
```

algorithms. The work of Kato et al. [19] proposes a GPU scheduler to provide prioritization and isolation capabilities for GPU applications in real-time and multi-tasking environments. CUDA graphs were explored under OpenMP as a compile-time implementation strategy [20], abstracting the concept of static graphs from users.

In contrast, the focus of our work is twofold: i) analyze the potential of using CUDA graph for a better programming productivity, performance and scalability of applications; and ii) propose a new specification for a better integration of OpenACC with CUDA Graph.

VII. Conclusions

In this work, we present how CUDA Graph can be efficiently and successfully used on GPUs in a way that applications are no longer limited to execute a single kernel. We evaluated the use of static graphs and the OpenACC programming model, using the PSO algorithm as a test case. Several advantages arise from the use of OpenACC and CUDA Graph: i) we provide a mechanism to easily benefit from using static graphs on GPUs, without compromising the performance; ii) in most cases we are close to peak performance while comparing the results with a pure and optimized CUDA code; and iii) by using OpenACC we enable programmers to write and offload parallel code into the GPU in an easy and transparent way. Finally, we propose a new pragma-based clause to integrate the use of static graphs as part of the OpenACC specification, which provides a simpler and more transparent way to implement static graphs from this programming model.

Acknowledgment

This project has received funding from the EPEEC project from the European Union's Horizon 2020 Research and Innovation program under grant agreement No. 801051.

References

[1] L. Toledo, A. J. Peña, S. Catalán, and P. Valero-Lara, "Tasking in accelerators: Performance evaluation," in *20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2019, pp. 127–132.

[2] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*, 1st ed. Addison-Wesley Professional, 2017.

[3] C. Bonati, E. Calore, S. Coscetti, M. D'elia, M. Mesiti, F. Negro, S. F. Schifano, and R. Tripiccione, "Development of scientific software for HPC architectures using OpenACC: The case of LQCD," in *IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*, 2015, pp. 9–15.

[4] R. Dietrich, G. Juckeland, and M. Wolfe, "OpenACC programs examined: A performance analysis approach," in *44th International Conference on Parallel Processing*, 2015, pp. 310–319.

[5] C. Chen, C. Yang, T. Tang, Q. Wu, and P. Zhang, "OpenACC to Intel Offload: Automatic translation and optimization," in *Computer Engineering and Technology*, 2013, pp. 111–120.

[6] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with OpenACC, OpenCL and CUDA," in *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[7] G. Alan, "Getting started with CUDA Graphs," 2019. [Online]. Available: https://devblogs.nvidia.com/cuda-graphs/

[8] Eberhart and Yuhui Shi, "Particle swarm optimization: Developments, applications and resources," in *Congress on Evolutionary Computation*, 2001, pp. 81–86.

[9] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Internat. Conference on Neural Networks (ICNN)*, vol. 4, 1995, pp. 1942–1948.

[10] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, pp. 33–57, Jun 2007.

[11] "Benchmark set," in *Particle Swarm Optimization*. John Wiley and Sons, Ltd, 2010, ch. 4, pp. 51–58.

[12] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory access performance in CUDA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[13] U. Jarzabek and P. Czarnul, "Performance evaluation of Unified Memory and Dynamic Parallelism for selected parallel CUDA applications," *J. Supercomput.*, vol. 73, no. 12, p. 5378–5401, Dec. 2017.

[14] P. Valero-Lara, P. Nookala, F. L. Pelayo, J. Jansson, S. Dimitropoulos, and I. Raicu, "Many-task computing on many-core architectures." *Scalable Computing: Practice and Experience*, vol. 17, no. 1, pp. 32–46, 2016. [Online]. Available: https://doi.org/10.12694/scpe.v17i1.1148

[15] P. Valero-Lara and F. L. Pelayo, "Full-overlapped concurrent kernels," in *The 28th International Conference on Architecture of Computing Systems (ARCS)*, 2015, pp. 1–8.

[16] P. Valero-Lara and F. L. Pelayo, "Analysis in performance and new model for multiple kernels executions on many-core architectures," in *IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing (ICCI*CC)*, 2013, pp. 189–194.

[17] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.: Pract. Exper.*, vol. 23, no. 2, p. 187–198, Feb. 2011.

[18] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures." *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011.

[19] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIX Annual Technical Conference (ATC)*, 2011.

[20] C. Yu, S. Royuela, and E. Quiñones, "OpenMP to CUDA Graphs: A compiler-based transformation to enhance the programmability of NVIDIA devices," in *23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2020, pp. 42—47.