Tasking in Accelerators: Performance Evaluation

1st Leonel Toledo
Barcelona Supercomputing Center (BSC),
Barcelona, Spain
leonel.toledo1@bsc.es

2nd Antonio J. Peña
Barcelona Supercomputing Center (BSC),
Barcelona, Spain
antonio.pena@bsc.es

3rd Sandra Catalán
Barcelona Supercomputing Center (BSC),
Barcelona, Spain
sandra.catalan@bsc.es

4th Pedro Valero-Lara
Barcelona Supercomputing Center (BSC),
Barcelona, Spain
pedro.valero@bsc.es

***Abstract*—In this work, we analyze the implications and results of implementing dynamic parallelism, concurrent kernels and CUDA Graphs to solve task-oriented problems. As a benchmark we propose three different methods for solving DGEMM operation on tiled-matrices; which might be the most popular benchmark for performance analysis. For the algorithms that we study, we present significant differences in terms of data dependencies, synchronization and granularity. The main contribution of this work is determining which of the previous approaches work better for having multiple task running concurrently in a single GPU, as well as stating the main limitations and benefits of every technique. Using dynamic parallelism and CUDA Streams we were able to achieve up to 30% speedups and for CUDA Graph API up to 25x acceleration outperforming state of the art results.**

***Index Terms*—CUDA, Dynamic Parallelism, GPU, CUDA Graph, CUDA Stream.**

I. Introduction

In recent years, GPU compute capabilities have been significantly increasing, however, applications and scalability of algorithms still face some important challenges. One important problem regarding scalability is hardware resource assignment; which makes it difficult to take advantage of the GPU architecture and sometimes applications are limited to execute a single kernel in the GPU without taking advantage of the whole capability of the device.

With the introduction of Dynamic Parallelism in CUDA, it is possible to launch kernels from threads running on the device. This means that the GPU can dynamically generate/execute work without returning to the host or await for new threads to be launched. CUDA Dynamic Parallelism allows implicit synchronization between the parent and the child kernels, this allows the device to execute work as needed from within the GPU and divide tasks looking to achieve the maximum degree of parallelism [1].

Tasking, on the other hand makes it possible for algorithms with run-time dependent execution flow, to be paralellized. For instance, a while-loop with independent chunks of work could have some of these chunks executed simultaneously. Tasking provides a solution for this problem by implementing a queueing system, which dynamically handles the assignment of threads to the work that needs to be performed. Threads continue to pick up work until the queue of taks is empty [2]. Task parallelism is a programming paradigm that provides more flexibility, meaning that the developer is able to deal and solve with certain problems in efficient ways, for instance, loops which length is unknown at run time [3].

Another interesting alternative is to combine CUDA Dynamic Parallelism with CUDA Streams. In that way it is possible not only to launch kernels from the GPU, but also to execute them asynchronously, creating a task-oriented programming model within the GPU. However for some applications, CUDA streams are not enough to efficiently process some HPC applications, for instance deep neural network training or scientific simulations have an iterative structure were the same workflow is executed repeatedly. CUDA stream require the work to be resubmitted at every iteration which is both time and resource consuming. To address this issue, since CUDA 10.0 it is possible to represent the workflow as a graph as an alternative for submitting tasks using CUDA. A graph consists of a series of operations such as memory copies and kernel launches, connected by dependencies which are defined separately from its execution. With this feature it is possible to determine the number of nodes and the structure that optimizes a specific problem.

This work explores different approaches to run several tasks with data dependencies on the same GPU envisioning a future integration of tasking oriented programming models and NVIDIA GPUs. We are testing different algorithm variants of a well known problem: DGEMM matrix multiplication, to test the capacity of dynamic parallelism under different circumstances. First, we test the performance of dynamic parallelism with three different kernels. Second, we extend the previous experiment by adding CUDA streams to the same three kernels to test the impact of asynchronous tasking inside the device, and finally, we test the implications of doing this operations from both the host and the device. Finally, we study the performance of CUDA Graph to determine if there is an advantage from either side.

Matrix operations are relevant within scientific and engi-

neering computing operations, there has been a significant effort for developing libraries that solve efficiently both sparse and dense matrix operations [4]. In this work, we present the results of evaluating DGEMM operations on the GPU using dynamic parallelism and CUDA graph in different configurations using the following heterogeneous system: 2 x IBM power9 8335-GTH at 2.4 Ghz, 32 GB RAM memory, and a NVIDIA V100 (Volta) GPU with 16GB HBM2 and NVLink2 for high-bandwidth communication between CPU and GPU.

The contribution of this work is the analysis and evaluation of the current CUDA features for tasking in latest GPU architecture, this is a preliminary study and analysis for a future integration of NVIDIA GPUs and tasking-oriented programming models.

The rest of this document is organized as follows: Section II discusses the most relevant work that was taken into consideration for this paper. Section III describes the considerations taken for the implementation of the kernels and how the dynamic parallelism and CUDA graph are exploited. Finally section IV discusses the conclusions and future work.

II. Related Work

It is not uncommon to find libraries that implement task-based programming models [5], [6], [7], [8]. For instance, StarPU, which is designed for programming CPU/GPU hybrid architectures handling run-time concerns and task dependencies [9]. Other alternatives which follow a similar paradigm are Intel Thread Building Blocks (TBB) and High performance ParalleX (HPX). The rise of alternative for task-based programs has been so significant the last decade, that even the OpenMP model integrates task-parallelism from version 3.0 [2].

Another interesting approach is OmpSs, which is a task-based programming model that extend OpenMP directives to give support to asynchronous parallelism and devices heterogeneity.

Distributing work by tasking is something common when working with parallel systems, such as clusters, grids or cloud computing, however it is not so common in hardware accelerators [10].

In the last CUDA releases, a new feature can be found which allows to execute a set of independent kernels on the same GPU [11]. One of the feature to execute multiple kernels on the same GPU consists in using CUDA streams, for each kernel a stream is created and every command, action and data transfer is stored on independent queues. Another alternative is to create hierarchical calls within the device, exploiting the dynamic parallelism that the device is capable of [12], [13], [14].

The work of [15] presents a novel characterization for dynamic parallelism, observing the current approaches of subtask aggregation with the objective of improving the performance of irregular applications by introducing child kernels to reduce workload imbalance and improve GPU utilization, reporting a maximum of 1.8x speedup.

III. Implementation

For this study we propose four different approaches to benchmark dynamic parallelism and task oriented programming on the GPU. The first approach uses only dynamic parallelism, and tasks are generated sequentially. Second, we introduce streams within the GPU, that way we take advantage of the several execution queues that are available in the device. Next, we compare the same approach, but instead of instantiating jobs within the GPU, we create tasks from the host using multiple streams at the same time. Finally, we test task performance using the CUDA graph API. Every scenario was tested using our own DGEMM kernel based on the work of [4]

Figure 1 shows the proposed configuration for the different experiments that we perform. We define three different scenarios with different degrees of parallelism, synchronization and tasks executed every time. The objective of these configurations is to test how well the device handles each scenario and the obtained performance when the GPU executes more than one task at the same time. In the first configuration (Top) the amount of work that is performed at every iteration is very low, almost sequential so the device must wait for synchronization after every calculation is performed. The second configuration (center) has a higher degree of parallelism, it is not the optimal, but a higher number of operations are expected to be executed at the same time before the device needs to synchronize. Finally, the last proposed experiment (bottom), has the highest degree of parallelism and the number of operations executed at the same time before the device needs to wait for the next phase is expected to be the highest.

For all three kernels we made the required modifications in order to test dynamic parallelism, CUDA Streams and CUDA Graph to execute several tasks. For this experiment we are performing DGEMM matrix multiplication to represent the units of work. To determine the impact of every technique, we use matrices of size N*N which are divided into tiles of size M*M. Those values can be modified to define the amount of work, synchronization and the number of kernels that should be executed within the GPU.

A. Dynamic Parallelism

The first approach we used to test the proposed experiment was using only dynamic parallelism to evaluate the GPU tasking capabilities; the main advantage of this feature is the capacity of the GPU to constantly create work without returning to the host [16], [17], [18]. Figure 2 shows the results of different configurations and amounts of work that the device executed. As expected kernels that made the most parallel calculations had a better performance overall, the best cases had performance gains about 30%. However, there are important considerations that must be addressed, even though performance improves from test to test, it is not as significant as we expected and this is mainly due because two main reasons. First, even though it is not necessary to return to the host to generate more work, most of the memory fetching is from global memory. This approach was tested

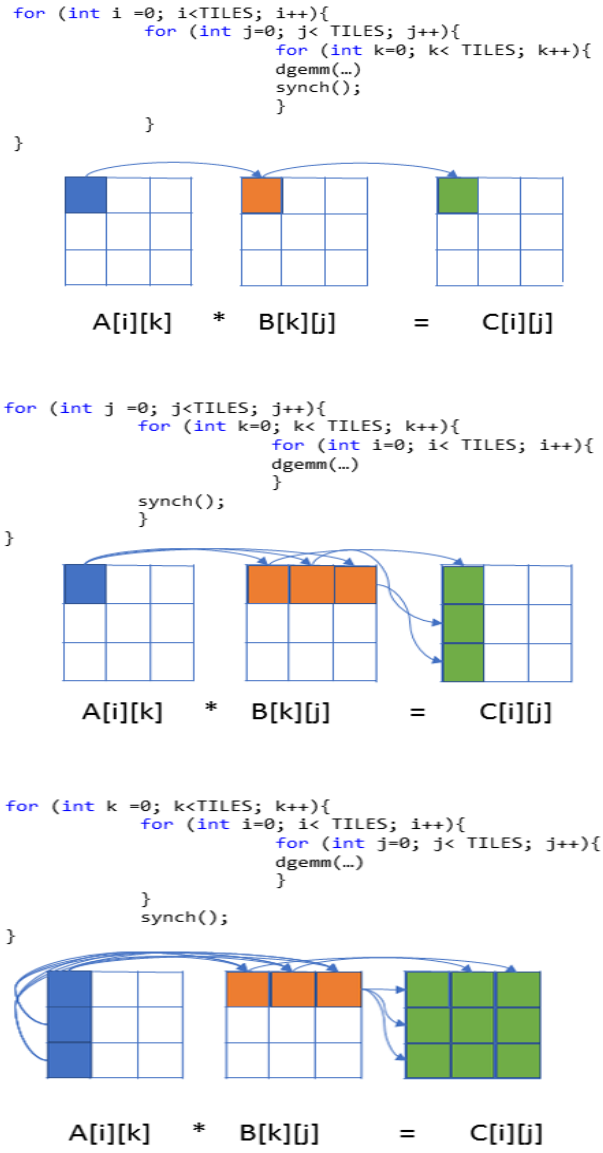

Fig. 1. Benchmark test cases used to test different degrees of parallelism in DGEMM operations. For the first test (Top) synchronization is performed after every operation, for the second test (center) every Nth operation and for the final test (bottom) every N*Nth operation.

by switching the DGEMM kernel to the one implemented in CUBLAS in order to avoid loses or gains due to the implementation of the kernel and not by the synchronization pattern. CUBLAS was faster than our implementation, but the key observation is that the pattern persisted, in other words the speedup percentage was almost the same as the obtained with our kernels. This further confirms that the bottleneck of the execution is how the hardware handles the creation and destruction of kernels in between launches from the device. Second, there is a significant overhead that is involved with creating kernels within the GPU which directly impacts the performance. To mitigate that, we added CUDA streams to maximize the amount of asynchronous work.

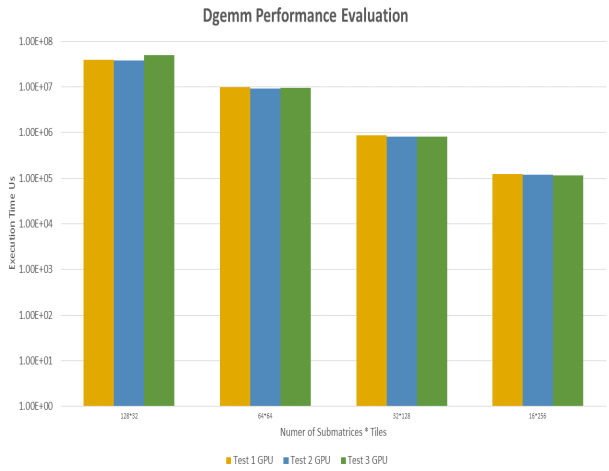

Fig. 2. Results of the different configurations using Dynamic Parallelism on a 4096*4096 Matrix. We tested several configurations which are shown in the x axis. Representing the number of tiles we used times the size of the elements contained in a particular tile.

B. CUDA Streams with Dynamic Parallelism

With the addition of CUDA streams, we can assign tasks to different execution queues and as long as there does not exist data dependencies between the kernels, they should run concurrently. Once created, a device stream can be used by any thread within the same thread block. However, it cannot be used after the thread block finishes executing, on other thread blocks or on the host. Similarly, streams created on the host cannot be used on the device.

By including streams into the evaluation, it was observed that there was a significant improvement in terms of performance, the experiment as a whole took less time to complete in all the configurations that were tested against using only dynamic parallelism without explicit concurrency. However, speaking of how the GPU handles several tasks at the same time, this approach shows limitations. Regardless of the configuration, the GPU did not execute efficiently the different tasks that needed to execute, even when the third configuration design had much more parallelism and a higher potential for overlapped calculations, it did not outperformed significantly the first configuration that was almost sequential. This is most likely due to the fact that the overhead of creating kernels from the device is really expensive in terms of computing time. Figure 3 shows the results of the performance of launching different configurations of DGEMM kernels and the time it takes in terms of milliseconds to complete the execution.

1) CUDA Streams from Host: Based on the previous results, it is important to address that using dynamic parallelism, whether it is alone or with CUDA streams, has an important bottleneck that is directly related to the creation of kernels within the GPU, to test the impact that it has on the performance and task execution, we performed the same experiments but this time the launches were made from the host not from the device.

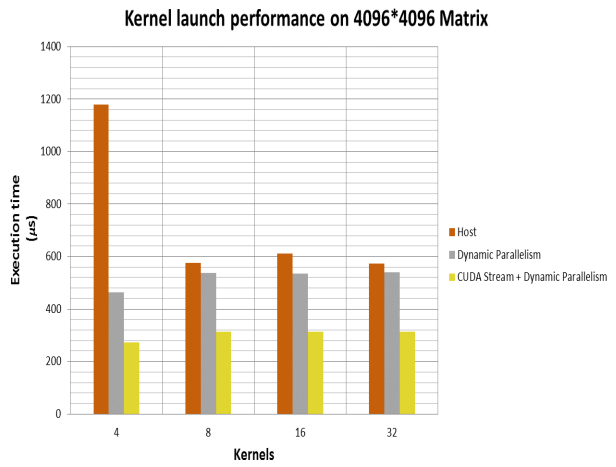

Fig. 3. On red is the result of executing N given kernels from the host, like a regular CUDA application. On gray is the execution time required to launch the same number of kernels using CUDA Dynamic Parallelism. Finally on yellow, the results of launching those kernels using CUDA streams and CUDA Dynamic Parallelism from the device.

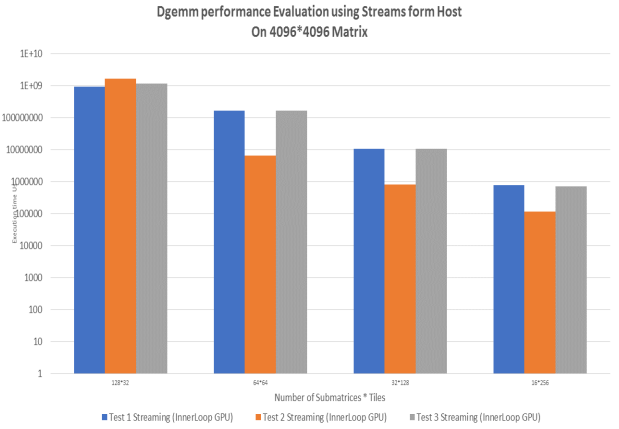

Fig. 4. Results of the evaluation of launching different kernels using different configurations for CUDA Streams from the host using different configurations. On the x-axis, the number of tiles times the elements contained in any particular tile is represented.

Figure 4 shows the results of the experiment. Depending on the configurations and the number of tiles and the sizes employed, there is a speedup that is directly related to the degree of synchronization required for each kernel. The main gains we were able to achieve was by efficiently implementing the use of CUDA streams, however they have some drawbacks as well, for instance it is possible to create as many streams as jobs to be launched, nevertheless the GPU cannot efficiently handle as many asynchronous calls. On average, the biggest acceleration was achieved when using 8 to 16 concurrent executions.

Dynamic parallelism has some advantages over launching kernels from the host, it reduces memory and data transfer between both devices, and work can be dynamically created within the GPU. Nevertheless it is important to address that there is a significant overhead involved in the creation of each new kernel, and in some cases the trade-off between latency and speedup is not worth it. Two key limitations exists, first the communication between a parent thread and its child threads has to be done through global memory variables, and second launching kernels from GPU threads has a non-trivial performance overhead, which both were key factors for the limited speedup that was achieved in this context.

For the sake of clarity and completeness, the results illustrated correspond to the execution time for computing a matrix 4096*4096 with 256 tiles and 65536 elements.However for this study we used different configurations of almost every variable that we had control of, for example the size of the matrices ranged from 128x128 to 8192x8192 elements. The number of tiles in the matrices determined the number of kernels that were launched in the experiments and ranged from 2x2 to 1024x1024 with the corresponding tiles to properly fit the original matrix. All tests were performed using square matrices. We tested the same synchronization scenarios using our DGEMM implementation from the device. The computing time drastically varies depending on the scenarios, and the degrees of parallelism that were achieved. It is important to highlight that for every experiment synchronization played a determinant role performance wise, in other words, task that had the most barriers performed significantly slower than the kernels whose tasks were launched asynchronously. Regardless the configuration of the experiment in terms of number of tiles or elements the conclusions are consistent with the presented results.

C. CUDA Graph

Finally, in the last test we performed, we adapted the code to test the previous configurations but using the latest CUDA Graph API. Figure 5 shows the graphs used to represent the work flow structure. On the left we have the first case in which all the tasks are running on the same stream, meaning that there is little to none task overlapping and kernels must wait for each other to finish and then start executing. On the second configuration (middle) we execute N kernels at the same time. This N is defined by the variable which determines the size of the Matrix. Finally the third configuration (right) shows the scenario were we can calculate a whole layer of the resultant matrix at a time. As the previous experiments we were able to define the size of the matrices, how many tiles we used to divide it and the size of each tile. Listing 1 shows how the implementation is performed, depending on the benchmark test, nodes must wait for the previous level to finish executing before starting a new task.

Listing 1. Cuda Graph Implementation

```
cudaGraph_t graph;
cudaStream_t streams[jobs];
cudaEvent_t dgemmEvent;
cudaStreamCreate(&streamForGraph));
cudaGraphCreate(&graph);
//case 1
```

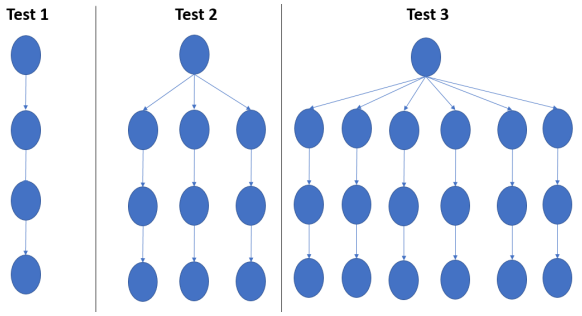

Fig. 5. Graph design to test the different configurations for the experiments, each node represents a DGEMM operation which is treated as an independent task. Tasks at different levels cannot execute until the previous one has finished.

```
cudaGraphAddNode(graph,
    kernel_a, {}, ...);
cudaGraphAddNode(graph,
    kernel_b, { kernel_a }, ...);
cudaStreamWaitEvent(dgemmEvent);
cudaGraphAddNode(graph,
    kernel_c, { kernel_b }, ...);
cudaStreamWaitEvent(dgemmEvent);
//case 2
cudaGraphAddNode(graph,
    kernel_a, {}, ...);
cudaGraphAddNode(graph,
    kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph,
    kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph,
    kernel_d, { kernel_a }, ...);
cudaStreamWaitEvent(dgemmEvent);
cudaStreamSynchronize(streamForGraph);
```

Figure 6 shows the results of the previously described evaluation. Using the CUDA Graph API, we were able to achieve a better degree of parallelism and task overlapping. For instance, between benchmark case 1 - 2 we achieved on average a 4x acceleration, and between benchmark case 1 - 3 a 12x acceleration. On the best cases we were able to achieve a 25x acceleration between cases 1 - 2 and 80x acceleration between cases 1 - 3. This is due to several factors. First the approach we are proposing works better when the graph design is breadth oriented instead of depth oriented. This reduces significantly the amount of computations that require wait events and overlapping is easily achieved. Next, this approach works better when we maintain a balance between the number of tiles and the size of each one. It is better to work with less number of tiles if it means we can have bigger sub matrices to compute. However when the number of tiles is small (between 2 - 16) we are not able to achieve significant gains in terms of tasking because there is not enough work to distribute and process concurrently.

Figure 7 shows the expected speedup for the different configurations. Prior to executing the benchmark cases we

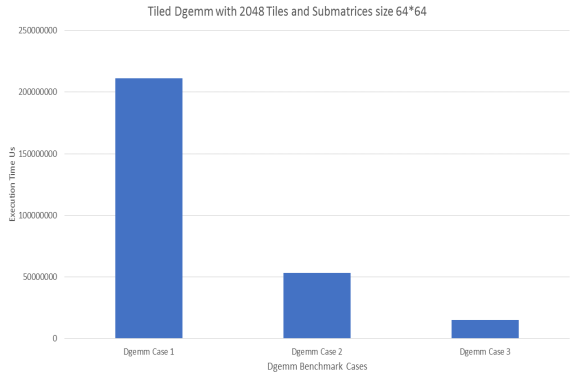

Fig. 6. Results of executing tiled Dgemm matrix operations using the different graph configurations

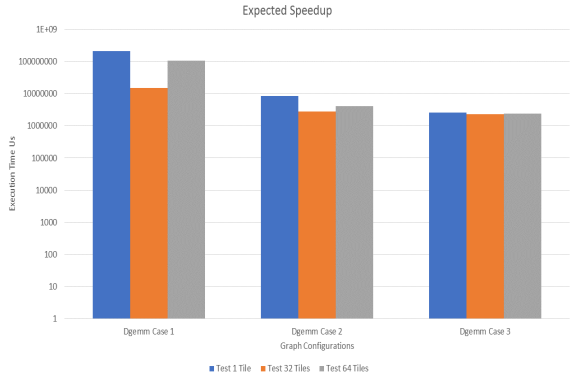

Fig. 7. Expected speedup for the different configurations of CUDA Graph.

computed a projected maximum speed up, which was calculated by executing the exact amount of nodes that each individual graph needed to complete all tasks without making any DGEMM operations at all, just launching all the nodes and measuring the time that they needed to complete using CUDA Graph API. The results of the theoretical maximum is represented in Blue, on Yellow is the performance of a determined kernel using 32 tiles and on gray are the results of the execution using 64 tiles. It is important to note that the configuration of the work flow is determinant on the performance gains that can be achieved using CUDA Graph. When comparing the results of CUDA Graph against dynamic parallelism and CUDA Streams it is evident that the graph feature outperforms both in terms of efficiency and work overlap in the device.

IV. Conclusions and Future Work

In this work we have tested several kernel benchmark configurations, different sizes for both matrices and tiles with dynamic parallelism and asynchronous streaming as well as CUDA Graph API. There is a speedup between the proposed scenarios, on average a 30% acceleration is perceived in each execution using dynamic parallelism and concurrent streams. However, while there is a performance gain, it is not as significant as expected, this is due to the nature of the dynamic

parallelism, creating tasks from within the GPU tends to be slower to the host when a small number of threads are required. There are some limitations that need to be addressed in more efficient ways, for instance, when dealing with matrices larger than 8192*8192 elements, the device is not capable of handling the volume of data, even when using libraries such as CUBLAS to make the computations.

The presented results show different degrees of speedups, however in the scope of this work, dynamic parallelism and concurrent streams within the device do not work as expected. There is much overhead involved and work do not overlap properly. On the other hand, CUDA Graph API has a better performance overall, in both terms of performance and tasking capabilities. CUDA Graph API allowed kernels to run concurrently while minimizing waiting and synchronization. With the addition of CUDA Graph API, we are able to outperform the current results such as the ones mentioned in the work of Zhang, et al [15].

After analysing the presented results, this work shows how the GPU can be used for a tasked-oriented programming model in a way that applications are no longer limited to execute a single kernel in the GPU and taking the most advantage of the device capabilities, and avoiding issues such as the bottlenecks presented in the creation of kernels with dynamic parallelism while having independent chunks of work executing at the same time.

As future work it would be interesting to make a comparison between other models such as OmpSs and evaluate which alternative proves better in terms of tasking in the GPU as well as the integration of both features for future releases. As a matter of fact CUDA Graph API proved to be the superior option for task-oriented programming models such as OmpSs, which could definitely benefit from the integration to its architecture.

V. Acknowledgments

This project has received funding from the EPEEC project from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801051, from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya , under project MPEXPAR: Models de Pro-gramació i Entorns d'Execució Paral · lels (2014-SGR-1051). Finally, this project also received funding from the Spanish Ministry of Economy and Competitiveness under the Juan de la Cierva Grant Agreement No IJCI-2017-33511 , and from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska Curie grant agreement No. 749516 .

References

[1] F. L. P. Pedro Valero-Lara, "Full-overlapped concurrent kernels." *ARCS 2015- The 28th International Conference on Architecture of Computing Systems.*, vol. 1, no. 1, pp. 1–8, 2015.

[2] R. V. D. P. E. S. C. Terboven, *Using Openmp–The Next Step: Affinity, Accelerators, Tasking, and Simd*, paperback ed. Mit Press, 2017.

[3] P. Valero-Lara, I. Martínez-Pérez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta, "Variable batched DGEMM," in *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, 2018, pp. 363–367. [Online]. Available: https://doi.org/10.1109/PDP2018.2018.00065

[4] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of dgemm on fermi gpu," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 35:1–35:11. [Online]. Available: http://0-doi.acm.org.millenium.itesm.mx/10.1145/2063384.2063431

[5] P. Valero-Lara, S. Catalán, X. Martorell, and J. Labarta, "BLAS-3 optimized by ompss regions (lass library)," in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019*, 2019, pp. 25–32. [Online]. Available: https://doi.org/10.1109/EMPDP.2019.8671545

[6] P. Valero-Lara, R. Sirvent, A. J. Peña, and J. Labarta, "Mpi+openmp tasking scalability for multi-morphology simulations of the human brain," *Parallel Computing*, vol. 84, pp. 50–61, 2019. [Online]. Available: https://doi.org/10.1016/j.parco.2019.03.006

[7] P. Valero-Lara, R. Sirvent, A. J. Peña, X. Martorell, and J. Labarta, "Mpi+openmp tasking scalability for the simulation of the human brain: Human brain project," in *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*, 2018, pp. 5:1–5:8. [Online]. Available: https://doi.org/10.1145/3236367.3236373

[8] J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan, "Parallel programming models for dense linear algebra on heterogeneous systems," *Supercomput. Front. Innov.: Int. J.*, vol. 2, no. 4, pp. 67–86, Mar. 2015. [Online]. Available: https://doi.org/10.14529/jsfi150405

[9] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," INRIA, Research Report RR-7240, Mar. 2010. [Online]. Available: https://hal.inria.fr/inria-00467677

[10] F. L. P. J. J. S. D. I. R. Pedro Valero-Lara, Poornima Nookala, "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

[11] F. L. P. Pedro Valero-Lara, "Analysis in performance and new model for multiple kernels executions on many-core architectures." *ICCI*CC.*, vol. 1, no. 1, pp. 189–194, 2013.

[12] D. Merrill and A. S. Grimshaw, "High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing." *Parallel Processing Letters*, vol. 21, no. 2, pp. 245–272, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/ppl/ppl21.html/MerrillG11

[13] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications." in *IISWC*. IEEE Computer Society, 2014, pp. 51–60. [Online]. Available: http://dblp.uni-trier.de/db/conf/iiswc/iiswc2014.html/WangY14

[14] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing." in *HPCA*. IEEE Computer Society, 2016, pp. 358–369. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpca/hpca2016.html/WangYMCZG16

[15] J. Zhang, A. M. Aji, M. L. Chu, H. Wang, and W.-c. Feng, "Taming irregular applications via advanced dynamic parallelism on gpus," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: ACM, 2018, pp. 146–154. [Online]. Available: http://doi.acm.org/10.1145/3203217.3203243

[16] A. Andy, "Adaptive parallel computation with cuda dynamic parallelism," 2014. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/

[17] J. Dong, F. Wang, and B. Yuan, "Accelerating birch for clustering large scale streaming data using cuda dynamic parallelism," in *Intelligent Data Engineering and Automated Learning – IDEAL 2013*, H. Yin, K. Tang, Y. Gao, F. Klawonn, M. Lee, T. Weise, B. Li, and X. Yao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 409–416.

[18] M. T. Jeffrey DiMarco, "Performance impact of dynamic parallelism on different clustering algorithms," vol. 8752, 2013. [Online]. Available: https://doi.org/10.1117/12.2018069