POSTER - Collective Dynamic Parallelism for Directive Based GPU Programming Languages and Compilers

Guray Ozen
Barcelona Supercomputing Center, Universitat Politecnica de Catalunya
Barcelona, Spain
guray.ozen@bsc.es

Eduard Ayguade
Barcelona Supercomputing Center, Universitat Politecnica de Catalunya
Barcelona, Spain
eduard.ayguade@bsc.es

Jesus Labarta
Barcelona Supercomputing Center, Universitat Politecnica de Catalunya
Barcelona, Spain
jesus.labarta@bsc.es

ABSTRACT

Early programs for GPU (Graphics Processing Units) acceleration were based on a flat, bulk parallel programming model, in which programs had to perform a sequence of kernel launches from the host CPU. In the latest releases of these devices, dynamic (or nested) parallelism is supported, making possible to launch kernels from threads running on the device, without host intervention. Unfortunately, the overhead of launching kernels from the device is higher compared to launching from the host CPU, making the exploitation of dynamic parallelism unprofitable.

This paper proposes and evaluates the basic idea behind a user-directed code transformation technique, named collective dynamic parallelism, that targets the effective exploitation of nested parallelism in modern GPUs. The technique dynamically packs dynamic parallelism kernel invocations and postpones their execution until a bunch of them are available. We show that for sparse matrix vector multiplication, CollectiveDP outperforms well optimized libraries, making GPU useful when matrices are highly irregular.

1. INTRODUCTION AND MOTIVATION

Graphics Processing Units (GPU) have become an essential component in high-performance computing) architectures to address the computational and energy-efficiency requirements of a broad range of applications. Early GPU programs were based on a flat, bulk parallel programming model, in which programs had to perform a sequence of kernel launches, each kernel trying to expose enough data parallelism to efficiently use the available resources in the GPU. In order to give support to more irregular codes (e.g. graph algorithms with irregular data access patterns and unpredictable control flows), the latest GPU released architectures include Dynamic Parallelism (DP). DP makes it possible to launch kernels from threads running on the device, without intervention of the threads running on the host processor.

Unfortunately, the overhead of launching kernels from the device is much higher compared to launching from the host. Recent studies [2, 3] show that DP introduces a noticeable overhead during kernel launch, precluding in most cases the exploitation of nested parallelism to improve application performance. In order to mitigate this overhead, this paper proposes Collective Dynamic Parallelism (CollectiveDP), a code transformation technique for directive-based programming model in order to efficiently exploit the potential performance of nested parallelism.

OpenMP and OpenACC are the two most widely used directive-based GPU programming languages. The latest version of OpenACC supports nested parallel regions to target DP in recent Nvidia GPUs; however OpenMP standard does not support yet. A recent study [1] proposed the semantics for nested teams regions in OpenMP and gave an implementation based on the use of DP. Figure 1 shows a simple example to illustrate the support for directive nesting in these two approaches.

```
1  /* Nested parallelism in OpenACC */
2  #pragma acc parallel loop
3  for (int i = 0; i < N-1; ++i) {
4     #pragma acc parallel loop
5     for (int j = arr[i]; j < arr[i+1]; ++j)
6        // BB.Process
7  }
8
9  /* Nested parallelism in extended OpenMP */
10 #pragma omp target
11 #pragma omp teams distribute parallel for
12 for (int i = 0; i < N-1; ++i) {
13    #pragma omp teams distribute parallel for
14    for (int j = arr[i]; j < arr[i+1]; ++j)
15       // BB.Process
16 }
```

Figure 1: Simple example with nested regions in OpenACC and extended OpenMP [1].

2. CODE TRANSFORMATION

In DP a parent kernel can instantiate the execution of child kernels. Instead of directly launching the execution of these child kernels, *CollectiveDP* dynamically bundles kernel instantiations made by the parent kernel and postpones their execution until a sufficiently large number of them is available, mitigating the excessive overhead introduced by DP. To do that, *CollectiveDP* saves in a buffer the necessary context (variables) of the parent kernel in order to execute the child kernel instantiation a posteriori. Finally, when all threads executing the parent kernel finish, *CollectiveDP* launches the execution of a single kernel, named *Next-Level*

PACT '16 September 11-15, 2016, Haifa, Israel

ACM ISBN 978-1-4503-4121-9/16/09.

DOI: http://dx.doi.org/10.1145/2967938.2974056

parallel Kernel (NLK), to actually execute the finer-grained invocations in parallel. Hence, *CollectiveDP* is able to thoroughly minimize kernel launching count and making nested parallelism usable, providing the right granularity for irregular applications.

Figure 2 shows the skeleton of the transformed code implementing the basic *CollectiveDP* idea. After initializing the data structures necessary to support *CollectiveDP* in `BB.CollectiveDP.Init` (line 2), the parent kernel may iterate several times (lines 5–6) depending on the size of the problem `N`. Later, each thread executing the parent kernel decides (conditional statement in line 8, checking if enough iterations need to be executed) whether to immediately execute the loop (lines 11–14) or to save the context for postponed execution of the loop as a child kernel (line 9). Finally, in `BB.CollectiveDP.HandleCases` (line 16) all postponed kernel invocations are executed through a single *NLK* dispatch (line 19). The thread responsible for launching *NLK* can be either the master of warp or the master of CTA (two implementations named –CTA and –WARP).

```
1 __global__ macc_collectivedp(..., MT* buffer) {
2   BB.CollectiveDP.Init
3   int tid = threadIdx+blockIdx.x * blockDim.x;
4   int numThreads = grimdDim.x * blockDim.x;
5   for (int u = tid; u < N;
6             u += numThreads) {
7     BB.BeforeProcess
8     if (tc > threshold)
9       BB.CollectiveDP.Save
10    else {
11      for(int p = ptrs[u];
12          p < ptrs[u+1]; p++)
13            BB.Process
14      BB.AfterProcess
15    }
16    BB.CollectiveDP.HandleCases
17    if(macc_master() && buff_counter) {
18      int BS = get_optimal_thread();
19      macc_NLK<<<buff_counter,BS>>>(...);
20} } }
```

Figure 2: Code transformation skeleton for the exemple in Figure 1.

Figure 3 shows the skeleton for the *NLK* kernel that processes the contexts that were saved in a buffer (`buffer`) for postponed kernel executions. First, *NLK* pops a context of the parent thread according to its block identifier; after that it simply executes the code in the original kernel with the context popped from the buffer in parallel. In this example, the context only contains the value of `u` in Figure 2.

```
1 __global__ void macc_NLK(..., MT *buffer) {
2   int u = buffer[blockIdx.x].macc_u;
3   for(int p = ptrs[u] + threadIdx.x ;
4       p < ptrs[u + 1]; p += blockDim.x)
5       BB.Process
6   BB.AfterProcess
7}
```

Figure 3: NLK generated by the compiler

3. EVALUATION

Experiments are done on compute nodes equipped with a CPU Intel(R) Core(TM) i7-4820K, 64GB of main memory, executing 64-bit Debian OS and a Nvidia Tesla Titan-X GPU with 3072 CUDA cores, compute capability 5.2 and 12GB of device memory.

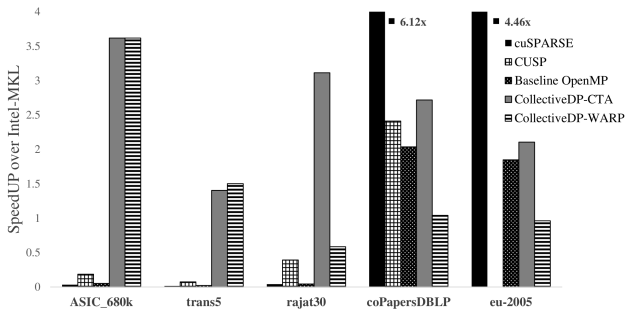

Figure 4: Speed–up for SpMV over Intel MKL library on the host.

We selected sparse matrix vector multiplication (SpMV `y = Ax`), one of the most used kernels in HPC. We compared *CollectiveDP* with state-of-the-art libraries: Nvidia cuSPARSE and CUSP. Figure 4 presents the speed–ups, relative to Intel MKL library (11.2 using Intel OpenMP) executed on the host, that are obtained with the different *CollectiveDP* approaches. For highly irregular input matrices (on the left) *CollectiveDP* speeds up up to 3.61x, 1.6x and 3.16x; for the other two less irregular matrices it achieves maximum speed-up of 2.7x and 2.1x.

4. CONCLUSION

This paper proposes and evaluates a code transformation technique, *CollectiveDP* to effectively support nested parallelism in GPUs, mainly tailored to compilers for user-directed accelerator programming models such as OpenACC or OpenMP. *CollectiveDP* dynamically packs kernel invocations and postpones their execution until a bunch of them are available. Performance is evaluated using different input matrices for a sparse matrix vector multiplication kernel.

5. ACKNOWLEDGMENTS

This work is partially supported by the IBM/BSC Deep Learning Center Initiative, by the Spanish Government (Severo Ochoa program SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P project) and by the Generalitat de Catalunya (contract 2014-SGR-1051).

References

[1] G. Ozen, E. Ayguadé, and J. Labarta. Exploring dynamic parallelism in openmp. In *Proceedings of the Second Workshop on Accelerator Programming using Directives, WACCPD 2015, Austin, Texas, USA, November 15, 2015*, pages 5:1–5:8, 2015.

[2] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 51–60, 2014.

[3] Y. Yang and H. Zhou. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 93–106, 2014.