Exploring Dynamic Parallelism in OpenMP

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

GPU devices are becoming a common element in current HPC platforms due to their high performance-per-Watt ratio. However, developing applications able to exploit their dazzling performance is not a trivial task, which becomes even harder when they have irregular data access patterns or control flows. Dynamic Parallelism (DP) has been introduced in the most recent GPU architecture as a mechanism to improve applicability of GPU computing in these situations, resource utilization and execution performance. DP allows to launch a kernel within a kernel without intervention of the CPU. Current experiences reveal that DP is offered to programmers at the expenses of an excessive overhead which, together with its architecture dependency, makes it difficult to see the benefits in real applications.

In this paper, we propose how to extend the current OpenMP accelerator model to make the use of DP easy and effective. The proposal is based on nesting of `teams` constructs and conditional clauses, showing how it is possible for the compiler to generate code that is then efficiently executed under dynamic runtime scheduling. The proposal has been implemented on the MACC compiler supporting the OmpSs task–based programming model and evaluated using three kernels with data access and computation patterns commonly found in real applications: sparse matrix vector multiplication, breadth-first search and divide–and–conquer Mandelbrot. Performance results show speed-ups in the 40x range relative to versions not using DP.

General Terms

Programming Models, Compilers

Keywords

CUDA, GPGPU, OpenMP, OpenACC, Dynamic Parallelism, OmpSs

WACCPD@SC '15 Austin-TX, USA

I. INTRODUCTION

The use of accelerators has been gaining popularity in the last few years due to their potential performance and higher ratios of performance to consumed power and performance to system cost when compared to homogeneous architectures based on multi-cores. The main examples of these recent hardware accelerators include General Purpose GPUs from Nvidia, AMD and ARM, the Intel Xeon Phi co-processors or FPGAs from Xylinx and Altera. The current Top500 list reflects the popularity of heterogeneous computers, as four of the Top10 machines have either GPGPUs or Xeon Phi co-processors. This paper focuses on GPGPUs, and in particular on how to effectively exploit the Dynamic Parallelism (DP)[8] feature supported in the latest devices [9].

GPU programming is architecture-dependent and applications need to make use of data parallelism in order to benefit from the massive amount of tiny processing units. GPU programming easily results in performance losses when regularity is not a characteristic of the application (e.g. divergent control flow, irregular memory access pattern or workload dependency). The above mentioned DP feature, recently introduced in the architecture of GPUs from Nvidia and AMD, has been proposed as a possible solution to solve these issues. The feature makes it possible to launch kernels from threads running on the device, so that threads can launch more threads. In this way, the CPU can launch the execution of a coarse-grained kernel on the device, which in turn launches finer-grained kernels on the device to do work where needed. This paper focuses on the use of DP in GPUs supporting CUDA.

The use of DP has some limitations, which include the overhead and serialization introduced when a parent grid launches child grids, the limit on maximum nesting depth, and the data sharing between parent and child only through global memory in a non-consistent way. Dealing with these limitations introduce difficulties in programming and may cause inefficiencies that can easily result in application slowdown. [14] recently explored the ideal performance of DP in CUDA and the actual performance when straightforwardly applied in irregular applications.

OpenACC v2.0 exposes DP to programmers, allowing the use of nested accelerator compute constructs (`parallel` and `kernels`). As long as the device supports DP, nested constructs are transformed using DP. However, most currently available OpenACC 2.0 compilers do not support it.

This paper contributes with an extension to the OpenMP accelerator model to expose DP to programmers by simply nesting `teams` constructs. In addition, an `if` clause is provided to the `teams` construct to manage DP conditionally: a child kernel is launched if and only if a given condition is met, avoiding unprofitable child kernel launches. The proposal has been implemented in the MACC [11] compiler, a research compiler for the accelerator model in the OmpSs [2] programming model. Performance evaluation is done on an Nvidia GPU using three selected kernels with data access and computation patterns commonly found in real applications: sparse matrix vector multiplication, breadth-first search and divide–and–conquer Mandelbrot.

The rest of the paper is organized as follows: Section II reviews DP in the Nvidia architecture and CUDA programming model, and provides some background on directive–based accelerator programming models, with special focus on OpenMP 4.0 and its current implementation in the MACC compiler for OmpSs; Section III discusses the potential scope and impact of DP; Section IV proposes an extension to the OpenMP 4.0 accelerator model to expose DP and its associated implementation details; Section V presents experimental results using three mentioned kernels above; Sections VI and VII describe related work, and conclusions and future work, respectively.

II. BACKGROUND

This section briefly describes the support for DP in recent Nvidia GPUs and CUDA programming model as well as the support in directive-based approaches (OpenACC, OpenMP 4.0 and MACC/OmpSs).

A. CUDA Dynamic Parallelism

CUDA Dynamic Parallelism [8] is one of the new functionality provided in the Nvidia Kepler architecture. Threads in a kernel launched by the host (parent kernel) are able to launch new kernels (child kernels) which can also launch new ones in a nested way; the maximum nesting depth is limited by the architecture (e.g. 24 in the GK110). With DP, threads can be dynamically launched based on the amount of work that needs to be performed in a particular region of the grid domain.

Figure 1 shows a simplified version of the Mandelbrot code in CUDA from [7]. This code implements the Mariani-Silver algorithm, which relies on the fact that the Mandelbrot set is connected (i.e. there is a path between any two points belonging to the set). More generally, if the border of a region has a certain constant dwell, then every pixel in the region has the same dwell. With this assumption the algorithm recursively divides the region (kernel invocation at Line 14) if the value of dwell is not constant on the border (Lines 4 and 6); otherwise each pixel within the block is set to the computed dwell value (kernel invocation at Line 9). The algorithm also checks if the maximum architectural nesting depth is reached, branching to the iterative per-pixel computation (kernel invocation at Line 18).

When DP is used the global memory in the device is coherently accessed by the parent and the child kernel, but it is weakly consistent. To make it fully consistent the `cudaDeviceSynchronize` synchronization function needs to be

```
1 __global__ void mandelbrot_block(...,
2                      int x0, int y0, int d, int depth) {
3    x0 += d * blockIdx.x, y0 += d * blockIdx.y;
4    int common_dwell = border_dwell(..., x0, y0, d);
5    if (threadIdx.x == 0 && threadIdx.y == 0) {
6       if (/* common dwell value for border */) {
7          // uniform dwell, just fill
8          ...
9          dwell_fill<<<...>>>(...);
10         }
11      else if (depth + 1 < MAX_DEPTH) {
12         // subdivide recursively
13         ...
14         mandelbrot_block<<<...>>>(..., depth + 1);
15      } else {
16         // leaf, per-pixel kernel
17         ...
18         mandelbrot_pixel<<<...>>>(...);
19      }
20   }
21 }
```

Figure 1: Recursive Mandelbrot in CUDA

used. Local and shared memory are private to each kernel. Also, their execution order is unspecified unless parent and child are synchronized. In child kernels, device streams are available (different from host streams), so that parent and child may work concurrently in sufficient resources are available in the device.

Nevertheless, to launch kernel brings overhead. In the case of DP, since child kernels are launched in the device, latency of all child kernels affects the parent kernel execution time and might drastically slowdown the application. Re-ordering of host instructions and kernel launch, using streams to overlap data transfers, ... can be used to reduce the negative effect. However, when the device is heavily occupied, launching child kernels is stalled until sufficient compute units are available on the device.

B. Accelerator support in directive-based approaches

OpenACC. OpenACC [10] provides directives that allow programmers to specify code regions to be offloaded to accelerator devices and to control many features of these devices explicitly. The main construct is `kernels`, instructing the compiler to optimize the annotated code region and exploit the available parallelism. OpenACC also offers the `data` and `update` constructs to manage data movement, and `parallel` and `loop` constructs for detailed control of kernel offloading and parallel execution of loop.

Version 2.0 provides support for nested parallelism, allowing the programmer to nest `parallel` and `kernels` constructs within outer `parallel` and `kernels` constructs. This nesting of directives instructs the compiler to activate DP if available in the target device. The currently available version of the OpenACC 2.0 compiler from PGI (15.7) used in the performance evaluation section does not support DP. As far as we know, only the PathScale ENZO compiler suite provides support for DP[12].

OpenMP 4.0. The accelerator model in the OpenMP 4.0 programming interface also provides a set of directives to offload the execution of code regions onto accelerators, to map loops inside those regions onto the resources available in the device architecture, and to map and move data between address spaces. The main directives are `target data` and `target`, which create the data environment and offload

the execution of a code region on an accelerator device, respectively. The specification also contains the `teams` directive to create thread teams. In each team the threads other than the master thread do not begin execution until the master thread encounters a `parallel` region. The `distribute` directive specifies how the iterations of one or more loops are distributed across the master threads of all teams that execute the teams region.

In the current specification OpenMP does not allow the use of nested device constructs, which precludes the use of DP in current compilers. In this paper we propose that extension and show how it can be implemented. We use the MACC [9] compiler that accepts the OpenMP 4.0 accelerator directives and implements them on the OmpSs infrastructure, both briefly described in the next subsections.

OmpSs programming model. OmpSs is a task-based data-flow programming model [2] that has influenced the evolution of the OpenMP tasking model, including task dependences. Its accelerator model targets homogeneous and heterogeneous architectures with hints provided to the runtime to dynamically exploit all the available resources, making use of the information available in the dynamically build task dependence graph. That information allows the runtime to automatically move data among address spaces and perform different kinds of optimizations (including locality, overlapping, prefectching, ...). OmpSs is currently able to run applications on clusters of nodes that combine shared-memory processors (SMPs) and accelerators (GPU, Xeon Phi and FPGA).

In the initial versions, OmpSs relied on the use of CUDA and OpenCL for the specification of tasks to be executed on accelerator devices. The `target` construct was used to simply direct the compilation of the source code to the appropriate CUDA or OpenCL back-end compiler and to inject the necessary code for task offloading and data movement. The `device` clause in the target construct is used not to specify the specific device to execute the region but the device type than can execute it, delegating to the runtime system the decision of which device to use in case multiple devices of that type are available.

MACC[1] compiler. MACC [11], yet another research compiler to investigate directive-based accelerator programming, is built on top of the Mercurium [4] source-to-source compiler framework and supports OmpSs and almost all directives of the OpenMP 4.0 accelerator model; it also supports some extensions to enhance code generation and use of the device memory hierarchy.

MACC code generation assumes the OmpSs runtime dependency based parallel execution of (offloaded and non-offloaded) task instances, task scheduling and transparent management of (coherent or non-coherent) physically distributed address spaces. The `target` construct is always associated to a `task` construct and the `target data` and `target update` constructs are not considered as the actual data movement is automatically implemented by the dataflow OmpSs runtime, reducing programming complexity. The rest of device related OpenMP 4.0 constructs (`teams`, `distribute` and `parallel for`) are thoroughly implemented with the originally specified semantics.

MACC adds to OpenMP 4.0 additional data sharing clauses to the `teams` construct: `dist_private`, `dist_firstprivate` and `dist_lastprivate`; with the `chunk_size` provided in the `dist_schedule` clause, the compiler is able to make a better use of the memory hierarchy in the device (e.g. global, shared and local memory), allocating portions of arrays in different teams and performing the necessary data movement according to `firstprivate` and `lastprivate` semantics.

[1]**MACC** is an abbreviation for "**M**ercurium **AC**celerator **C**ompiler".

III. MOTIVATION FOR DYNAMIC PARALLELISM

Dynamic parallelism is generally useful for problems where nested parallelism cannot be avoided. This includes, but is not limited to, algorithms using hierarchical data structures, such as adaptive meshes, graphs and trees, and algorithms using recursion, where each level of recursion has parallelism. These unstructured applications typically introduce irregular control flows and access patterns, which preclude the exploitation of peak performance in GPU devices because of thread divergence, poor SMX utilization and inefficient memory accesses.

For example, Figure 2 shows one of the nested loops in the Breadth-First Search application that is ported and annotated with device constructs from the OpenMP version of Rodinia Benchmark suite [5]. The number of instructions executed depend on the number of edges going out from each node (loop in Lines 12–15), which may introduce thread divergence and load unbalance problems.

```
1 #define THRESHOLD 256
2
3 #pragma omp target teams distribute parallel for
4 for( tid = 0; tid < N; tid++ )
5 {
6  if (h_graph_mask[tid] == true)
7  {
8   h_graph_mask[tid]=false;
9
10  #pragma omp teams distribute parallel for nowait \
11            if(h_graph_nodes[tid].no_of_edges > THRESHOLD)
12   for(i =    h_graph_nodes[tid].starting;
13        i < ( h_graph_nodes[tid].no_of_edges +
14              h_graph_nodes[tid].starting);
15        i++ )
16   {
17    int id = h_graph_edges[i];
18    if(!h_graph_visited[id])
19    {
20      h_cost[id]=h_cost[tid]+1;
21      h_updating_graph_mask[id]=true;
22    }
23   }
24  }
25 }
```

Figure 2: Breadth-First Search Code using the proposed nested teams

Other algorithms include recursion in their control flow, with functions that invoke themselves a number of times until a base case is executed. If that function is implemented as a device kernel, DP would allow its recursive invocation. Figure 1 illustrates this kind of algorithms with the Mariani-Silver version of Mandelbrot.

IV. DYNAMIC PARALLELISM IN OPENMP

In this section we propose an extension to support DP in the OpenMP accelerator model and describe its implementation in the MACC compiler.

Semantics of nested teams

We define a semantic for nested `teams` in OpenMP 4.0 [1] in order to explicitly specify DP (current semantic is not specified in the current release of the specification). In our proposal, each nested `teams` construct might be able to create a league of thread teams, with each team's master thread executing the code block in its scope. When the compiler encounters a nested `teams` construct, it extracts the inner region and creates a child kernel for it. Inner device constructs (e.g. `distribute, parallel for`) bind to the closest `teams` construct. Thus, the programmer can easily express kernels with different granularities and leave the compiler/runtime system to appropriately exploit that parallelism making use of DP.

As commented before, the use of DP incurs overheads that can easily degrade the performance of the application. For this reason we propose an `if` clause for the `teams` construct with a threshold condition that can be dynamically tested to select between kernel launch or inline execution. With the expression in the `if` clause a programmer can trade overhead, potential concurrency and load balance benefits of the DP mechanism. In the evaluation section we show the impact of such trade-off in the overall performance of the application and observe that typically there is a wide range of thresholds that result in close to optimal performances.The compiler generates and `if-then-else` code block to implement non-nested (executed by parent) and nested (using DP) versions of the invocation.

Figure 3 shows the code for sparse matrix vector multiplication using our proposed extensions. In this code, the sparse matrix is stored using the usual compressed row (CSR) format. In each iteration of the loop at Line 9 one row of the sparse matrix and vector are multiplied. The directive in Line 8 spreads the iterations of this loop across the teams/threads hierarchy created by the compiler. The inner loop in Line 18 performs the dot product of row by vector; since the number of elements in each row may change, its execution on a single thread would lead to thread divergence. In addition, the access to those elements can not be coalesced in order to efficiently access memory. The use of DP in this case, as instructed to the compiler in Lines 16-17 by using nested `teams`, might result in better resource utilization if the number of elements is long enough. The `if` clause is used with a condition to prevent launching inner kernels if the number of elements in the current row is bigger than a defined `THRESHOLD` value. Besides, Figure 4 shows OpenACC way of same multiplication. We have used its `parallel` construct. Also we indicate with `loop` and `reduction` to inner iteration. We couldn't activate OpenACC nested `parallel` feature on inner iteration as our OpenACC compiler (PGI) doesn't support DP yet.

Kernel launch

In order to implement kernel launch nesting, the MACC compiler needs to decide on a block size to use for each

```
1 void SpMV_CSR(uint R, uint N, TYPE *A, uint *cols,
2               uint *rows, TYPE *X, TYPE *y, int THRESHOLD)
3 {
4  int row, elem;
5
6  #pragma omp target map(to:A[R], cols[R], rows[R], X[R])
7                     map(from:y[R])
8  #pragma omp teams distribute parallel for
9  for(row = 0 ; row < R; ++row)
10 {
11   int row_start = rows[row];
12   int row_end = rows[row + 1];
13   int IIT = row_end - row_start;
14   TYPE dot = 0;
15
16   #pragma omp teams distribute parallel for
17               reduction(+:dot) if(IIT > THRESHOLD)
18   for (elem = row_start; elem < row_end; elem++)
19     dot += A[elem] * X[ cols[elem] ];
20
21   y[row] = dot;
22 }
23
24 }
```

Figure 3: Sparse Matrix Vector Multiplication using the proposed nested teams in OpenMP

```
1 void SpMV_CSR(uint R, uint N, TYPE *A, uint *cols,
2               uint *rows, TYPE *X, TYPE *y)
3 {
4  int row, elem;
5
6  #pragma acc parallel loop copyout(y[R]) \
7                copyin(A[R], cols[R], rows[R], X[R])
8  for(row = 0 ; row < R; ++row)
9  {
10   int row_start = rows[row];
11   int row_end = rows[row + 1];
12   int IIT = row_end - row_start;
13   TYPE dot = 0;
14
15   #pragma acc loop reduction(+:dot)
16   for (elem = row_start; elem < row_end; elem++)
17     dot += A[elem] * X[ cols[elem] ];
18
19   y[row] = dot;
20 }
21
22 }
```

Figure 4: Sparse Matrix Vector Multiplication OpenACC implementation

kernel launch. To help on this decision, the MACC compiler makes use of the `cudaOccupancyMaxPotentialBlockSize` API available from CUDA 6.5, which heuristically calculates the kernel's block size to achieve the best occupancy level. In the implementation presented in this paper, MACC uses this to configure the launch for the outermost kernel. For the invocation of kernels in the inner levels, MACC simply performs one-iteration per thread mapping.

Another issue related with the invocation of nested kernels is that the launching of child kernels by threads in the same block gets serialized. This serialization can be avoided if separate CUDA streams are used for each thread. The MACC compiler implements this hiding this limitation to the programmer and achieving kernel launch overlaps.

Data sharing

As mentioned before, parent and child kernels share the access to global memory in the device, but with weak consistency. In order to ensure consistent access to global memory, `cudaDeviceSynchronize()` needs to be used. To deal with this issue, our model implicitly injects `barrier` construct only at the end of inner `teams` construct, which could be disabled by adding a `nowait` clause to the `teams` construct.

When data sharing is necessary between the parent and child

kernels, the compiler uses an early memory allocation mechanism in order to reduce the overheads of global memory allocation. In essence, the compiler allocates an area for the child kernel while allocating an area for the parent kernel. Thus, the allocation latency during the parent kernel execution time is avoided. MACC compiler calculates memory size statically during the compile time, it thus allocates memory as it considers every single parent thread will invoke child kernel.

Reduction support

The `teams` directive allows `reduction(reduction-identifier : list)` clause. Therefore we have implemented reduction as well in the case of nested `teams` construct. To support current clauses of `teams`, our extension to OpenMP 4.0 also allows `reduction`. When the MACC compiler encounters this clause, it generates reduction code at the same time as child kernel generation. However, the reduced variables must be returned to the parent kernel, using global memory to achieve this communication. Memory allocation for variables in reduction clauses is then performed using the previously mentioned early allocation mechanism.

Recursion support

As mentioned before, there is a hardware limit on maximum nesting depth; as of Compute Capability 3.5, the hardware limit on depth is 24 levels. In order to hide this limit to the programmer, the compiler needs to generate multiple versions of the code.

Figure 5 shows a basic example for recursion transformation as currently done by MACC. The upper code block is an input code that involves recursion at Line 7. After transformation of this, the compiler generates three functions; `foo`, `gpu_foo` and `device_in_foo`. The function `foo` corresponds to the original function invocation and includes the entire code replacing the recursive call with a call to `gpu_foo`. This `gpu_foo` kernel will recursively call itself until the maximum nesting depth is reached; when this happens, the compiler will force the invocation of `dev_in_foo` which is sequential.

Input OpenMP Code Proposal for Recursion

```
1 #pragma omp declare target
2 void foo() {
3   /* code 1 */
4
5   #pragma omp target teams distribute parallel for
6   for (int i = 0; i < count; ++i) {
7     foo();
8   }
9
10  /* code 2 */
11 }
12 #pragma omp end declare target
```

Transformed Code of Recursion

```
1 void foo() {
2   /* code 1 */
3
4   gpu_foo<<< ... >>> (... , 0);
5
6   /* code 2 */
7 }
8
9 __global__ gpu_foo(..., int depth) {
10  ...
11  if (depth > MAX_DEPTH)
12    dev_in_foo();
13  else {
14    /* code 1 */
15    gpu_foo<<< ... >>> (... , depth + 1);
16    /* code 2 */
17  }
18 }
19
20 __device__ dev_in_foo() {
21
22  /* code 1 */
23  for (int i = 0; i < count; ++i) {
24    dev_in_foo();
25  }
26  /* code 2 */
27 }
```

Figure 5: Code transformation to handle recursion

V. EVALUATION

In this section we present the performance evaluation of the dynamic parallelism proposal for OpenMP and its implementation in the MACC compiler. To that end we use three kernels: sparse matrix vector multiplication (SpMV), breadth-first search (BFS) and recursive Mandelbrot. Although the kernels may seem simple codes, they are representative of issues that do appear in many applications (like load imbalance or recursion represented by Mandelbrot) and important kernels in engineering (SpMV) or graph processing (BFS) applications.

For the experimental evaluation we have used a node with two IBM Power8 [13] S824L sockets (10 cores each, 8-way SMT, running at 3.52 GHz) and 1 TB of main memory, and two Nvidia Tesla K40c GPU devices (2888 CUDA cores, compute capability 3.5) with 12GB of memory. GCC 4.9 has been used as a back-end compiler for CPU code generation and the CUDA 7.0 toolkit for device code generation. OpenMP codes are compiled with the MACC compiler. We also used the currently available OpenACC PGI 15.7 compiler for comparison purposes, which does not support DP. Evaluation results are reported in terms of execution time for the generated kernels and performance.

A. Sparse Matrix Vector Multiplication SpMV

Sparse matrix vector multiplication, `y = Ax`, is one of the most used kernels in scientific computing e.g., iterative solvers for linear systems of equations and eigensolvers. To evaluate SpMV we used seven matrices from the University of Florida sparse matrix collection [6], whose main characteristics are shown in the Table 1. The last two columns show how many times the number of elements in a row (`IIT`, computed in Line 13 in Figure 3) is larger than 256 or 128 (values used as `THRESHOLD` in the `if` condition in Line 17 in the same Figure.

Table 1: Sparse Matrices

Matrix	Size	Nnz	IIT>256	IIT>128
rajat30	*643994*	*6175377*	*121*	*277*
ASIC_680k	*682862*	*3871773*	*2*	*76*
rajat29	*643994*	*4866270*	*22*	*27*
transient	*178866*	*961790*	*13*	*31*
c-big	345241	2341011	2	*160*
Raj-1	263743	1302464	65	*93*
trans5	116835	766396	8	*32*

Three versions are evaluated: OpenMP/MACC implemented using OpenMP 4.0 and the MACC compiler; it does not make use of nested teams (just defined in Line 8 in Figure 3); OpenACC/PGI is equivalent to the previous one but implemented in OpenACC and compiled with the PGI compiler; finally OpenMP/MACC-DP uses the proposed teams nesting for OpenMP, as shown in Figure 3, and compiled with MACC in order to make use of DP.

Figure 6: Sparse Matrix Vector Multiplication Performance Results

Figure 6 shows the performance results obtained for the SpMV kernel. In order to clearly show performance difference, we took into account only kernel computation time, without considering data movement overheads. To calculate flop performance, we used this formula $(kernel_time/(2 * numRows * numOfNonZeroElem)$. On average, the OpenACC/PGI version is 10x times better than the equivalent OpenMP/MACC version. However, OpenMP/MACC-DP is 4x times faster than OpenACC/PGI, showing the performance benefit of using DP. When the threshold value is reduced to 128, the performance is lower; this is because, waiting child kernels takes more time even though child kernels are executed faster.

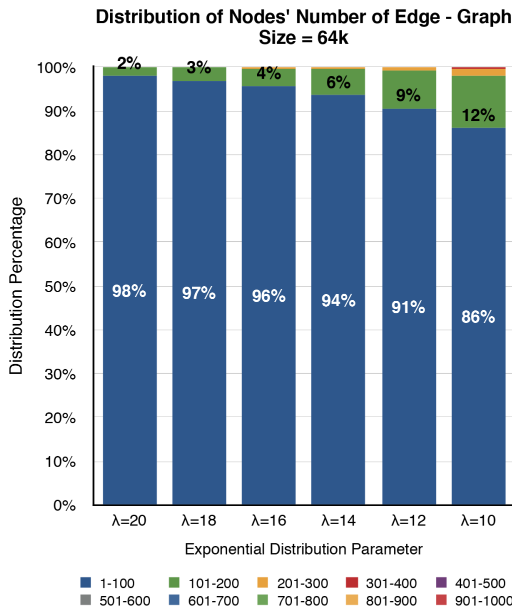

Figure 7: Breadth First Search 64K Graph Distribution

B. Breadth-First Search BFS

The structure of the BFS code used to perform this evaluation is shown in Figure 2, which comes from the Rodinia benchmark [5]. The code is annotated with `target teams distribute parallel for` in Line 3 and `teams distribute parallel for` in Line 10 conditionally guarded by the `if` clause dependent on the number of edges in Line 11, instructing the compiler to generate code making use of DP. We also used the CUDA implementation of this algorithm from the Rodinia benchmark suite without any modification. OpenACC and OpenMP 4.0 versions are ported from CPU OpenMP version of the benchmark suite.

In order to evaluate BFS application we used six different graphs of the same size (64K nodes), generated using the `graphgen` application that comes with Rodinia. The original `graphgen` randomly generates the number of edges for each node; the application has been modified to follow an exponential distribution. The edge distribution for the 6 graphs is shown in Figure 7.

Figure 8 shows the execution time results. Kernel execution time for the CUDA, MACC and PGI are very similar. However, the DP-enabled MACC version yields up to 2.7x speed-up.

C. Mandelbrot

Two different versions of Mandelbrot are used in this paper, both annotated using OpenMP 4.0 and compiled with MACC: *escape* and *Mariani-Silver*. In the *escape* algorithm each pixel of image is handled by a CUDA thread, which calculates the number of iterations to converge and decide whether it belongs to the set or not. This version clearly suffers from thread divergence. The *Mariani-Silver* algorithm exploits a hierarchical approach, as we explained before in Section II. The code is shown in Figure 10 that is ported from CUDA version at Figure 1. It incorporates three child kernel invocations. The first inner `teams` provides fine grain parallelism to fill uniform regions. For non-uniform regions it subdivides recursively as is seen at Line 17. When the maximum nested is reached or size of sub region is small the last `teams` is used to compute to rest of image pixel by pixel.

Figure 9 compares the performance results of these two algorithms for 4 different sizes of the set. The code of *escape* algorithm was compiled with MACC without DP. The DP-enabled MACC compiler was used for the *Mariani-Silver* algorithm. On the average a 3.62x speed-up is obtained; only for the smaller set size the *escape* algorithm performs better.

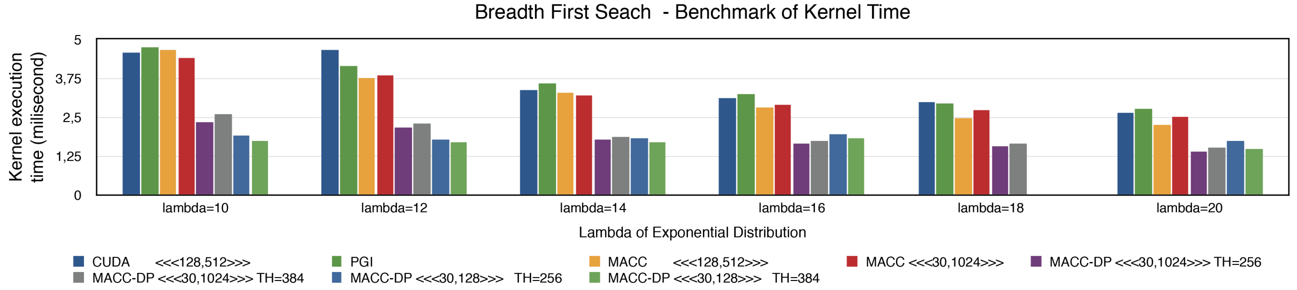

Figure 8: Breadth First Search CUDA Kernel Execution Time

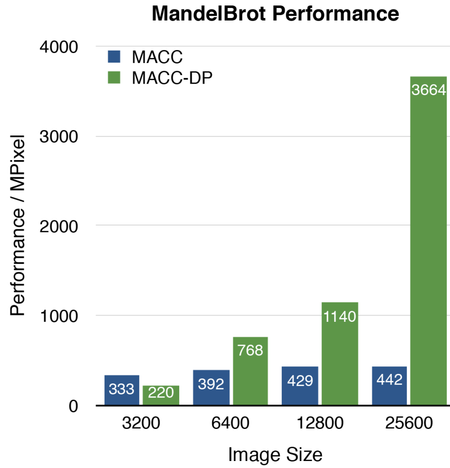

Figure 9: Mandelbrot Application Performance Results

```
1 #pragma omp declare target
2 void mandelbrot(... , int depth)
3 {
4    #pragma omp target teams distribute
5    for (...)
6    {
7       /* Computations */
8     if (/* common dwell value for border of rectangle */)
9     {
10       #pragma omp teams distribute parallel for
11       for (...)
12          /* Computations */
13    }
14    else if (depth + 1 < MAX_DEPTH
15       && /* condition to check size */ )
16    {
17       mandelbrot(..., depth + 1);
18    }
19    else {
20      #pragma omp teams distribute parallel for
21      for (...)
22         /* Computations */
23    }
24  }
25 }
26 #pragma omp end declare target
```

Figure 10: Mandelbrot code using the proposed nested teams in OpenMP 4.0.

VI. RELATED WORK

OpenACC [10] is a directive-based programming model designed to allow easy access to emerging advanced architecture systems for existing production codes. It also provides an approach to coding the newest technologies without the need to learn native GPU programming languages. Version 2.0 introduced nested-parallelism. Our proposal for OpenMP 4.0 has been clearly inspired by the approach followed by OpenACC.

In [15] we see an alternative to dynamic parallelism. They introduce a couple of directives in order to mimic nested parallelism without using DP. The main idea is to create more than the requested number of threads in advance, and to activate them to compute the iterations annotated by their directives. We believe that the solution based on dynamic parallelism is more suitable for large numbers of threads.

In order to obtain the best thread hierarchy, dynamic parallelism may be the solution. The research paper [3] compares two methods for achieving the best hierarchy. One of the methods does a statically configuration based on code structure at compile time. The other method uses dynamic parallelism at runtime to decide the hierarchy. Results show that with dynamic parallelism the best thread hierarchy is obtained; however, they reported a lot of overhead when shaping an entire hierarchy with dynamic parallelism.

[14] analyzes characteristics of irregular applications in which DP can be used. Performance results are shown in terms of ideal performance (excluding overhead of DP) versus real performance. These applications, though suitable for DP, do not benefit from its use, and in fact suffered from application slowdown.

VII. CONCLUSION AND FUTURE WORK

In this paper we have proposed an extension to the current OpenMP 4.0 accelerator model to allow `teams` nesting. The proposal is implemented using dynamic parallelism in the MACC compiler. The proposal also offers a way to conditionally control dynamic parallelism, avoiding in some cases the excessive overhead caused by this feature. Although this condition is currently inserted by the programmer, profile guided techniques or compilers which has just in time capability could discover these conditional expressions automatically.

VIII. ACKNOWLEDGMENTS

This work is partially supported by the BSC-IBM Technology Center for Supercomputing agreement, the Spanish Ministry of Innovation ”Computacion de Altas Prestaciones VI” (TIN2012-34557) and the ”Grup de Recerca de Qualitat” 2014-SGR-1051 from Generalitat de Catalunya.

IX. REFERENCES

[1] OpenMP ARB. Openmp application program interface, v. 4.0. 2013.

[2] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[3] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating gpu threads for openmp 4.0 in llvm. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press.

[4] Barcelona Supercomputing Center. The mercurium compiler http://pm.bsc.es/mcxx.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54, 2009.

[6] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1, 2011.

[7] NVIDIA. Adaptive parallel computation with cuda dynamic parallelism, http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/.

[8] NVIDIA. Cuda dynamic parallelism programming guide, 2013.

[9] NVIDIA. Next generation cuda compute architecture: Kepler tm gk110 http://www.nvidia.es/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf.

[10] OpenACC. The openacc standard, http://www.openacc-standard.org.

[11] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. On the roles of the programmer, the compiler and the runtime system when programming accelerators in openmp. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, pages 215–229, 2014.

[12] PathScale. Enzo , http://www.pathscale.com/enzo.

[13] Balaram Sinharoy, James Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, D. Levitan, S. Tung, David Hrusecky, James W. Bishop, Michael Gschwind, Maarten Boersma, Michael Kroener, Markus Kaltenbach, Tejas Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1), 2015.

[14] Jin Wang and Sudhakar 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.

[15] Yi Yang and Huiyang 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.