International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 212–229, July 2016

A Performance Evaluation of Dynamic Parallelism for Fine-Grained, Irregular Workloads

Max Plauth, Frank Feinbube, Frank Schlegel and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam
PO Box 990460, 14440 Potsdam, Germany

Received: February 3, 2016
Revised: May 4, 2016
Accepted: July 6, 2016
Communicated by Michihiro Koibuchi

Abstract

GPU compute devices have become very popular for general purpose computations. However, the SIMD-like hardware of graphics processors is currently not well suited for irregular workloads, like searching unbalanced trees. In order to mitigate this drawback, NVIDIA introduced an extension to GPU programming models called *Dynamic Parallelism*. This extension enables GPU programs to spawn new units of work directly on the GPU, allowing the refinement of subsequent work items based on intermediate results without any involvement of the main CPU.

This work investigates methods for employing *Dynamic Parallelism* with the goal of improved workload distribution for tree search algorithms on modern GPU hardware. For the evaluation of the proposed approaches, a case study is conducted on the *N-Queens* problem. Extensive benchmarks indicate that the benefits of improved resource utilization fail to outweigh high management overhead and runtime limitations due to the very fine level of granularity of the investigated problem. However, novel memory management concepts for passing parameters to child grids are presented. These general concepts are applicable to other, more coarse-grained problems that benefit from the use of *Dynamic Parallelism*.

Keywords: GPU computing, Dynamic Parallelism, workload distribution, irregular workloads

1 Introduction

Leveraging graphics hardware for general purpose tasks has gained a notable prevalence rate. Nonetheless, GPU computing struggles with attaining a broader target audience outside the field of scientific computation or image processing. One reason for this struggle is the SIMD-like nature of graphics processors, which narrows down the choice of tasks which may benefit from GPU-based implementations. While current GPU programming models like CUDA [16] or OpenCL [12] provide the means for utilizing the compute resources of GPU compute devices for general purpose tasks, one of the most limiting constraint has been that only the main CPU was able to launch kernels. Hence, irregular workloads are a representative class of tasks which fails to leverage the full performance of modern GPUs. Due to the heavily varying execution times of different threads, static workload distribution cannot achieve homogeneous utilization. Dynamic redistribution of work has not been feasible thus far, since only the main CPU has been able to redistribute work.

In the foreseeable future, lifting of certain limitations of the programming models might have the biggest potential for enabling other use case domains to tap the resources of GPU hardware. Aiming towards this path, NVIDIA recently introduced an extension called *Dynamic Parallelism*. *Dynamic Parallelism* empowers GPU kernels to launch nested kernels by themselves, which enables developers to influence control flows based on intermediate results without any involvement of the main CPU.

In the presentation that introduced *Dynamic Parallelism* to the public [11], NVIDIA presented several exemplary use cases, including fluid dynamics simulations, Mandelbrot sets and n-body simulations. For these problems, even the close-meshed regions of the adaptive grid contain compute-intensive portions of work. Such characteristics are ideally suited for demonstrating the strengths of *Dynamic Parallelism*, since the overhead for spawning new thread grids is easily outweighed by improved device utilization.

To analyze the performance characteristics of *Dynamic Parallelism* for fine-grained, irregular workloads, a case study of the *N-Queens* problem is conducted in the course of this paper. The goal of *N-Queens* is to find all valid configurations for placing N chess queens on a $N \times N$ chess board, so that no two queens threaten each other. Figure 1 illustrates all valid configurations for the simple case of $N = 6$. Popular implementation strategies of *N-Queens* use a search-tree representation and perform a depth-first search as illustrated in Figure 2. Inherent to this structure is the problem of irregular workload distribution for parallel implementations, as many branches of the search-tree reach invalid configurations rather quickly. Furthermore, the smallest unit of work per thread can be broken down to applying bitwise operations (see Section 2.3), which results in very light-weight threads. These characteristics make *N-Queens* a prime candidate for examining the behavior of *Dynamic Parallelism* for less-than-ideal workloads. We provide the following contributions:

1. We analyze the performance implications of *Dynamic Parallelism* for fine-grained, irregular workloads.
2. We present generally applicable memory management concepts for efficient intercommunication with child grids.
3. We assess the behavior of the runtime environment for corner cases where limits are exceeded regarding thread count or recursion depth.

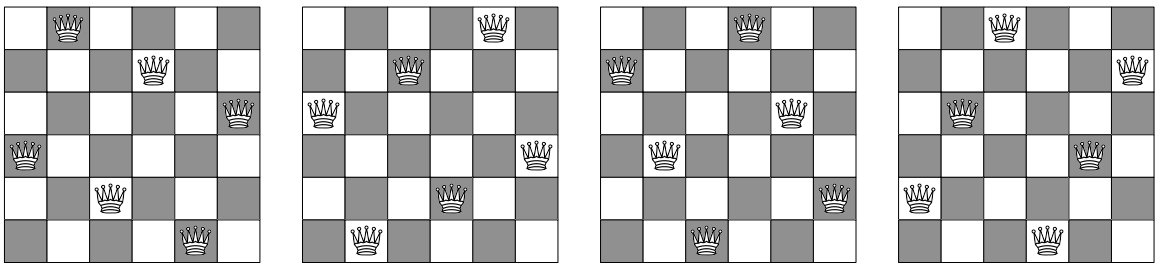

Figure 1: All four valid solutions to *N-Queens* for $N = 6$ are illustrated.

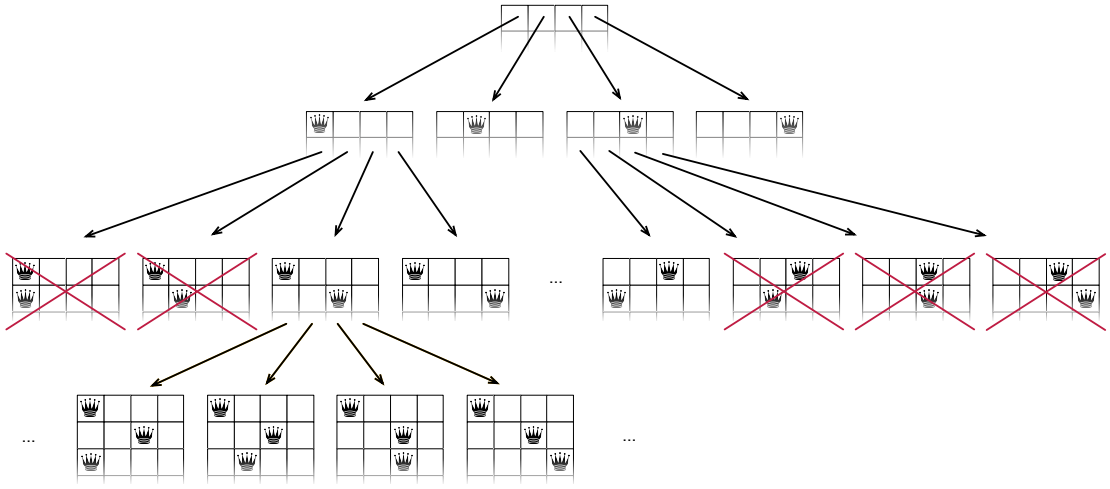

Figure 2: The *N-Queens* problem can be represented in the form of a search-tree, where each configuration of placing a queen spans a new set of subtrees.

2 Background

In this section, we review related work dealing with load balancing on GPUs, memory management on GPUs, *Dynamic Parallelism* and approaches to the *N-Queens* problem.

2.1 Load balancing on GPUs

Aiming towards optimal utilization of compute resources, several GPU-based load-balancing strategies exist. Cederman and Tsiga [4] grouped existing approaches into four classes. The general properties of each class are outlined and related approaches are presented hereafter.

2.1.1 Static task lists

Tasks are assigned to threads statically before the execution is launched. This approach fails if the amount of work per task is not known prior to the execution or when new tasks have to be created at runtime. A workaround exists that uses a read-only task list and an additional list with an atomic add function for incoming tasks [4]. Using this method, dynamic creation of tasks at runtime is possible. However it does not provide proper means of load balancing, since idle cores still have to wait for active threads to finish.

2.1.2 Blocking task-queues

One or multiple dynamically sized queues are used. Exclusive manipulation of queues is enforced using a locking mechanism. While this approach allows independent removal and creation of tasks for all threads, the locking mechanism represents a major bottleneck. The impact is especially notable for large numbers of threads [4] and thus severely affects GPU compute devices, which contain thousands of cores. A workaround has been suggested by Tzeng et al. [24], which is based on a dictionary that keeps account of dependencies between tasks. As soon as a task has finished, its dependent tasks are added to the task-queue. However, Tzeng et al. conclude that even with the workaround in place, the overhead caused by the locking mechanism cannot be outweighed for fine-grained tasks [24].

2.1.3 Non-blocking task-queues

Task-queues with lock-free manipulation techniques are usually implemented using atomic operations. As a consequence, fetching or creating new tasks causes much less overhead compared to blocking task-queues. However, the approach does not scale very well for large numbers of threads. Using mapped memory, Chen and Villa [6] have introduced a concept which uses non-blocking task-queues to implement a master-worker pattern, where the main CPU is able to generate tasks after a kernel has been launched. This approach is very well suited for scenarios where multiple GPUs have to be supplied with tasks.

2.1.4 Work stealing

For work stealing, each thread has a dedicated task-queue. New tasks are usually added by the associated thread, whereas arbitrary threads can extract work items. If its queue runs empty, the thread can fetch work from any other queue. Comparing GPU-based implementations of the previously mentioned approaches, Cederman and Tsiga found that work stealing performs best with regards to both scalability and execution time [4]. Chatterjee et al. [5] introduced an extension for the GPU computing use case, which adds a global task-queue that is used by the main CPU to enqueue new tasks. In a similar fashion, many hybrid approaches exist that combine properties of work stealing with other concepts. For example, Lauterback et al. [14] presented an approach where the contents of local task-queues are re-distributed on a regular basis to achieve balanced filling levels among queues.

2.1.5 Dynamic Parallelism

NVIDIAs implementation of *Dynamic Parallelism* is based on the *Grid Management Unit*, which was introduced with the *GK110 Kepler* GPU architecture. As illustrated in Figure 3, *Dynamic Parallelism* is a feature that enables GPU kernels to launch nested kernels by themselves, which enables developers to influence control flows based on intermediate results without any involvement of the main CPU. Strategies for leveraging this capability for implementations of the *N-Queens* problem are the extensively investigated in Section 3. Whereas the load balancing strategies discussed in the previous sections required developers to provide their own facilities to supply threads with tasks, *Dynamic Parallelism* allows the dynamic creation of threads since *CUDA 5.0* [16] and *OpenCL 2.0* [12]. However, employing *Dynamic Parallelism* is also accompanied by certain limitations which are specified in the *CUDA C Programming Guide* [16]. Most importantly, the maximum depth for nested kernel calls is limited to 24. Child thread grids launched by a kernel are not able to access registers or shared memory of their parent grids. Small data volumes can be passed directly to child grids using call parameters, whereas larger data volumes can only be handed over via global device memory. With the maximum size of the parameter buffer being limited to 4 kilobytes however, the data volume that can be passed through call parameters is quite limited.

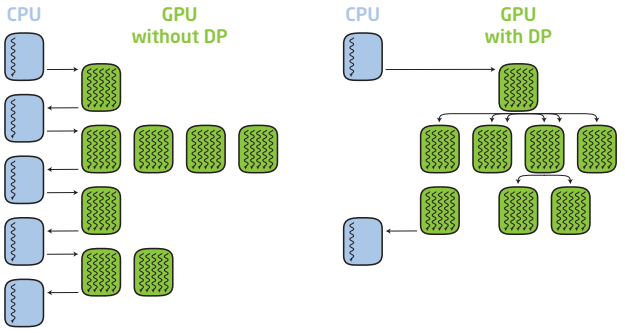

Figure 3: Without *Dynamic Parallelism* (DP), each step that influences the further control flow involves the CPU (left). Using *DP* (right), GPU kernels may launch child kernels by themselves.

2.2 Memory management on GPUs

Since *CUDA 3.2*, kernels have been able to allocate memory dynamically at runtime [16]. The default *CUDA* allocator is comparatively slow, which is why a range of custom allocators have emerged. *Xmalloc* [10] achieves factor 48 speed-up compared to the default *CUDA* allocator. This is achieved by delegating the allocation task to a single thread within a warp. Furthermore, pre-allocated buffers are used to accelerate the allocation process. *ScatterAlloc* [22] is an extension of *Xmalloc*, which achieves a speed-up of factor 10 compared to *Xmalloc* by avoiding simultaneous memory access using a hashing function. This approach has been augmented by *FDGMalloc* [25], which eliminates simultaneous access to memory entirely by using warp-local lists to serve allocation requests from pre-allocated buffers. With these improvements, *FDGMalloc* manages to outperform *ScatterAlloc*. Dynamic memory allocation is one of the major bottlenecks of the approaches documented in Section 3, which is why *FDGMalloc* is employed.

2.3 N-Queens

N-Queens is based on a puzzle, which was proposed by Max Bezzel in 1848 [3]. The goal of the puzzle is to find all valid configurations for placing 8 chess queens on a regular 8×8 chess board, so that no two queens threaten each other. The problem was later generalized to placing N queens on a $N \times N$ board for $N > 3$. For the simple case of $N = 6$, Figure 1 illustrates all valid configurations. The problem is $\mathcal{NP}$-hard and can be mapped to the clique problem [7, 9]. Furthermore, efficient approaches to the *N-Queens* problem come with a high degree of practicality, since the problem can be easily mapped to other problems [2].

Regarding implementation strategies for the *N-Queens* problem, three general algorithm classes have been identified by Erbas et al. [7]: brute-force trial and error algorithms $\mathcal{O}(N^N)$, permutation generation algorithms $\mathcal{O}(N!)$ and backtracking algorithms $\mathcal{O}(N!)$. Since brute-force approaches are not feasible, most implementations resort to the remaining approaches. However, even though permutation-based approaches and backtracking belong to the same complexity class $\mathcal{O}(N!)$, backtracking approaches have the advantage that many invalid branches can be ruled out very early. This is reflected by the fact that many of the best-performing implementations are based on backtracking techniques.

2.3.1 Serial implementations

The fastest serial implementation to date has been documented by Somers [21]. His solution is based on the recursive backtracking algorithm of Richards [18], who suggested the utilization of three bit masks to encode the position of queens on the board as illustrated in Figure 4. Using the bit mask representation, simple bitwise operations suffice to validate possible placement candidates. Somers implementation refined this approach by replacing expensive recursion with a self-managed stack of bit masks to implement a depth first search. Due to its superior performance, Somers implementation [21] is used as reference for single-threaded performance in the course of this paper.

2.3.2 Parallel implementations

In the field of parallel implementations of the *N-Queens* problem on general purpose processors, Kise et al. [13] were the first to solve the problem for $N = 24$ using an implementation based on the Message Passing Interface (MPI) standard. The implementation employed the master worker pattern for improved workload balancing and took 44 days on a 34-node PC cluster to retrieve all valid solutions. Later on, Rolfe [19] re-confirmed that MPI-based implementations are suited for efficient implementations of the *N-Queens* problem. A different approach was applied by Caromel et al. [1], who used a grid of 260 computers consisting of regular desktop computers as well as high end server hardware to retrieve all solutions for $N = 25$.

The current world record is held by Preußer et al. [17] from the Dresden University of Technology. Using a cluster of 26 field-programmable gate arrays (FPGAs) and 270 days of processing time,

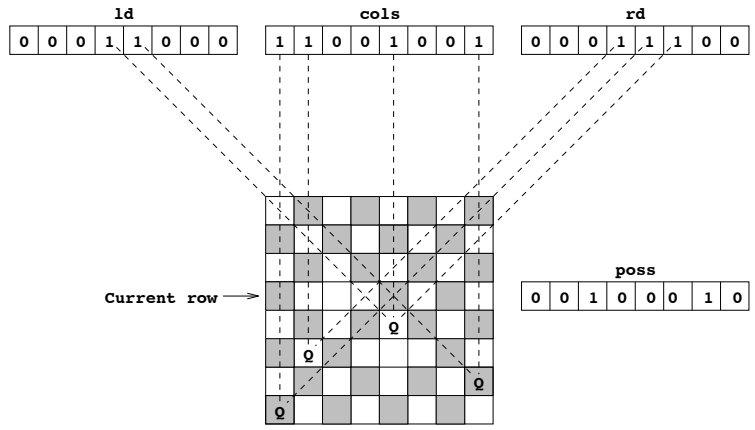

Figure 4: The chess-board can be efficiently represented using three bit masks. Bitwise operations can be applied to validate the current configuration. Image source: [18]

22,317,699,616,364,044 valid solutions were determined for $N = 26$. While the world record is held by the aforementioned FPGA-based approach, graphics hardware based implementations have also been well researched [8], [5, 23]. However, no publication is known to the authors that tries to solve the unbalanced workload distribution of *N-Queens* using *Dynamic Parallelism*. The fastest implementation for GPU compute devices known to the authors has been published by Feinbube et al. [8], which is based on Somers [21] serial implementation. In this approach, the main CPU creates initial board configurations for each thread and then hands over the tasks to the GPU. The partial results retrieved for each subtree are finally consolidated by the main CPU. The implementation [8] incorporates several GPU-specific optimizations and is also used as a reference for the implementation strategies evaluated in the course of this paper.

3 Approaches to Dynamic Parallelism

Different approaches of leveraging *Dynamic Parallelism* for GPU-based implementations of the *N-Queens* problem are presented in this section. Method *DP-1*, which is elaborated in Section 3.1, employs an excessively fine-grained approach and investigates the overhead caused by dynamic thread grid creation. In contrast to this, the strategy *DP-2* documented in Section 3.2 explores a more compacted concept where each GPU-thread is supplied with larger portions of work. Built upon the latter approach is the method *DP-3* (see Section 3.3), which aims at improving workload distribution and resource utilization. Finally, the concept *DP-SWAP* explained in Section 3.4 presents a strategy that is able to reduce the number of dynamic memory allocations in the context of *Dynamic Parallelism*. In Section 4, all implementations are evaluated and compared to the approaches of Somers [21] and Feinbube et al. [8]. Like these approaches, all strategies presented in the course of this work exploit the symmetry of *N-Queens* (see Figure 1).

3.1 DP-1: Thread swarming kernel

This fine-grained approach tries to place queens on the board iteratively row by row, using a very high thread count. For that purpose, the initial grid is supplied with a valid, pre-calculated configuration. Each thread in the grid attempts to place a queen in the next row onto a field specified by the threads index.

For the first attempt of this swarming approach, each thread that retrieves a valid configuration launches a new grid. As depicted in Figure 5, the newly created grids repeated the same actions until a thread manages to place a queen in the last row of the board. In this case, the thread atomically increments a global counter to document the valid configuration and terminates. Unlike

the backtracking approaches discussed in Section 2.3, this method does not resemble a depth first search, but rather a parallel breadth first search. The major drawback of this first attempt is that N defines the upper limit of threads in a grid. For realistic sizes for N that can be handled by modern GPUs, this strategy results in a thread count per grid well below the warp size of 32 threads. Since GPU multiprocessors execute entire warps, the difference between the warp size and the chosen N accounts for the number of unused cores and per warp.

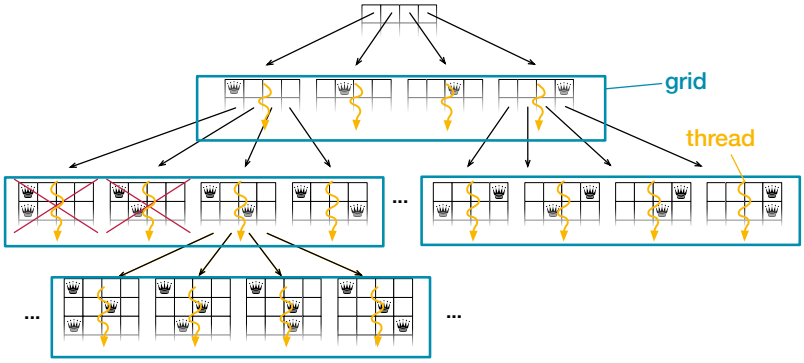

Figure 5: The first revision of *DP-1* launches one grid per row with one thread per field.

In order to clear out this drawback, a second attempt was made at implementing a thread-count intensive approach. As demonstrated in Figure 6, the main difference compared to the first approach is that one thread is responsible for evaluating multiple rows. Because one grid has to process all permutations, the grid contains N^z threads, where z is a configuration parameter that specifies the number of rows to be processed per grid. For further clarification, Listing 1 provides the CUDA source code which implements the stated strategy.

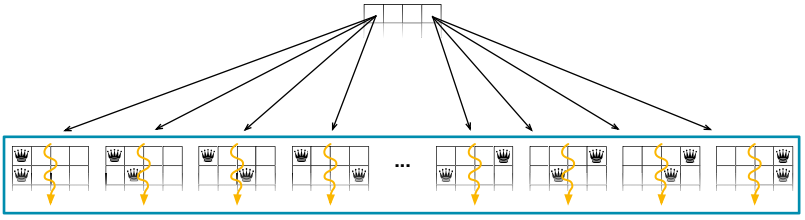

Figure 6: The improved version of *DP-1* processes multiple rows per grid and launches N^z threads, where z is the search depth.

```
1  template<int N>
2  __global__ void nQueensKernelSwarming(boardConfig_t boardConfig,
3    size_t rows, unsigned int numThreads) {
4    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
5    if (index >= numThreads) return; // does this thread participate?
6
7    unsigned int row, indexInRow;
8    for (row = 0; row < rows; ++row) {
9      indexInRow = index % N;
10     // check if queen can be placed, update boardConfig accordingly
11     if (!placeAndValidateQueenAtIndex(indexInRow, boardConfig)) {
12       return; // return when no queen can be placed
13     }
14
```

```
15      index /= N; // encode next row in remainder of index
16
17      // store valid solution
18      if (boardConfig.currentRow == N && index == 0) {
19        atomicAdd(&d_nQueensResult, 1);
20        return;
21      }
22    }
23
24    // create new stream and issue computation of next row in a new grid
25    cudaStream_t stream;
26    cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
27    unsigned int numBlocks = (numThreads + blockDim.x - 1) / blockDim.x;
28    nQueensKernelSwarming<N><<<numBlocks, blockDim.x, 0, stream>>>
29      (boardConfig, rows, numThreads);
30  }
```

Listing 1: The *DP1*-Kernel uses many light-weight threads to solve *N-Queens* recursively

3.2 DP-2: Condensed kernel with static search depth

Counteracting the overhead of the large number of light-weight threads employed in *DP-1*, this approach refers to the approach of Feinbube et al. [8] and aims at employing fewer threads and larger work packages. For that purpose, each thread applies backtracking to search a subtree up to a specified search depth. All valid configurations that are obtained by a thread at its maximum search depth are recorded. After the thread has checked all branches within its search depth, a new grid is launched upon the set of valid configurations as demonstrated in Figure 7. If a thread manages to place a queen in the last row of the board, the thread terminates after a global counter has been atomically incremented to document the valid configuration. The search depth limit was introduced to effect a periodic adjustment of the thread count in response to the encountered problem size.

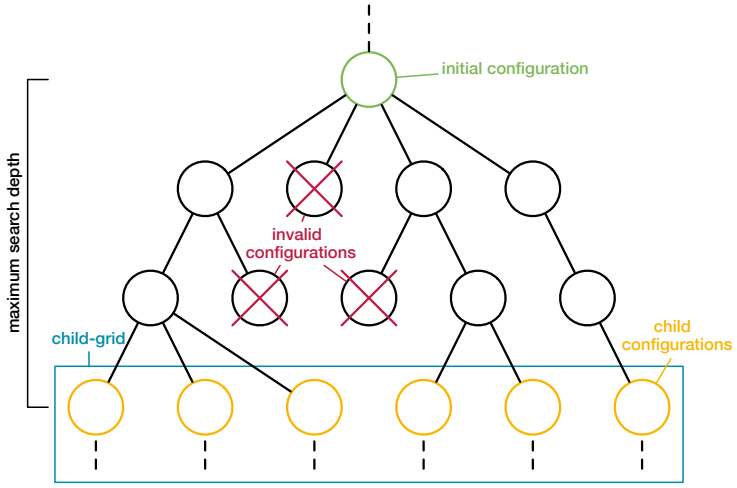

Figure 7: In *DP-2*, a thread traverses its subtree up to a maximum search depth. Valid configurations are passed to child grids.

Shared memory is used to record the state of the preceding rows for each thread. While shared memory provides much faster access times than global memory, its capacity is limited. The amount of shared memory required by the condensed approach is predominantly defined by the block size and the maximum search depth. The search depth defines the upper limit of valid configurations and corresponding bit masks that have to be accommodated in shared memory.

Unfortunately, the data volume that has to be passed exceeds the limits of the kernel parameters. As a result thereof, the configurations have to be passed via global memory, which needs to be

allocated dynamically. This approach employs a novel per-block memory management scheme that enables simple handover of large parameter data to child grids and mitigates the high costs of dynamic memory allocation. Frequent memory allocations are inhibited by allocating large memory blocks, where the size of the allocated memory region is proportional to the number of threads in a block. The latter property alleviates index-based access by child grids, since each thread in the child grid can identify its initial configuration using the thread index. Furthermore, memory is only allocated if at least one valid configuration has been found by a thread. However, this scheme does not only work for the specific problem at hands, but it is general enough to be applied to arbitrary problems that use *Dynamic Parallelism* and have to pass larger data volumes to child grids. For further exemplification of the condensed approach and its memory management scheme, Listing 2 provides the complete CUDA source code for *DP-2*.

```
1  template<int N>
2  __global__ void nQueensKernelCondensed(boardConfig_t** boardConfigs,
3    size_t depth, uint numThreads, bool freeMemory, uint* freeCounter) {
4    if (blockIdx.x * blockDim.x + threadIdx.x >= numThreads) return;
5
6    extern __shared__ char sharedMemory[]; // use SM to store configuration stacks
7    uint stackSize = (depth + 1) * blockDim.x;
8    boardConfig_t* confStacks = (boardConfig_t*) sharedMemory;
9    bitfield_t* bitfieldStacks =  (bitfield_t*) (confStacks + stackSize);
10
11   // apportion SM to threads and put initial configuration on stack
12   boardConfig_t* boardConfigBlock = boardConfigs[blockIdx.x];
13   boardConfig_t* confStack = confStacks + (threadIdx.x * (depth + 1));
14   confStack[0] = boardConfigBlock[threadIdx.x];
15
16   // wait for all threads, then free memory, as it is no longer needed
17   __syncthreads();
18   if (freeMemory && threadIdx.x == 0) {
19     free(boardConfigBlock);
20     if (atomicAdd(freeCounter, 1) == (gridDim.x - 1)) {
21       free(boardConfigs);
22       free(freeCounter);
23     }
24   }
25
26   // working variables
27   const bitfield_t mask = (1 << N) - 1; // indicates 'unused' bits
28   // indicates placement state of fields (1 = free, 0 = queen)
29   bitfield_t bitfield = mask & ~(confStack[0].col |
30     confStack[0].posDiag | confStack[0].negDiag);
31   bitfield_t* bitfieldStack = bitfieldStacks + (threadIdx.x * (depth + 1));
32   bitfieldStack[0] = bitfield;
33   bitfield_t lsb; // "least significant bit" marks the first free field
34   int stackPtr = 0; // current depth in the stack for (row of current subtree)
35
36   // parameters for potential child grid
37   boardConfig_t** childConfigs = 0; // array of references to config blocks
38   boardConfig_t*  currentConfigBlock = 0;
39   uint numChildConfigs = 0; // number of obtained child configurations
40   uint numChildBlocks = 0; // number of created memory blocks
41   uint currentBlockConfigIndex = 0; // index of current config
42
43   // traverse the subtree assigned to this grid
44   while (true) {
45     if (bitfield == 0) { // no space for a new queen
46       if (--stackPtr < 0) { // root of the partial tree has been reached
47         if (numChildConfigs > 0) { // start child grid for found configurations
48           cudaStream_t stream;
49           cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking);
50           uint* freeCounter = (uint*) malloc(sizeof(uint));
51           *freeCounter = 0; // counter for inter-block-synchronization
52           uint sharedMemSize = (sizeof(boardConfig_t) +
```

```
53              sizeof(bitfield_t)) * stackSize;
54            nQueensKernelCondensed<N><<<numChildBlocks,
55              blockDim.x, sharedMemSize, stream>>>(childConfigs,
56              depth, numChildConfigs, true, freeCounter);
57          }
58          return; // terminate traversal
59        }
60        bitfield = bitfieldStack[stackPtr]; // move back in tree
61        continue;
62      }
63
64      lsb = -((signed)bitfield) & bitfield; // seek for free queen positions
65      bitfield &= ~lsb; // mark new queen placement
66
67      // last row has not been reached yet
68      if (confStack[stackPtr].currentRow < (N - 1)) { // last row not y
69        if (stackPtr == depth) { // reached end of the current subtree
70          if (!childConfigs) { // initialize block-array if necessary
71            childConfigs = (boardConfig_t**) malloc(sizeof(boardConfig_t*) *
72              ((numThreads + blockDim.x - 1) / blockDim.x));
73          }
74          if (!(numChildConfigs % blockDim.x)) { // a new memory-block is needed
75            currentConfigBlock = (boardConfig_t*) malloc(
76              sizeof(boardConfig_t) * blockDim.x);
77            childConfigs[numChildBlocks++] = currentConfigBlock;
78            currentBlockConfigIndex = 0;
79          }
80          currentConfigBlock[currentBlockConfigIndex++] =  confStack[stackPtr--];
81          ++numChildConfigs; // store found configurations
82          bitfield = bitfieldStack[stackPtr]; // move back in tree
83          continue;
84        }
85
86      // the end of the current subtree has not been reached yet
87        int nextStackPtr = stackPtr + 1;
88        // mark new queen placement in bitfields
89        confStack[nextStackPtr].col =  confStack[stackPtr].col | lsb;
90        confStack[nextStackPtr].posDiag = (confStack[stackPtr].posDiag | lsb) << 1;
91        confStack[nextStackPtr].negDiag = (confStack[stackPtr].negDiag | lsb) >> 1;
92        confStack[nextStackPtr].currentRow = confStack[stackPtr].currentRow + 1;
93        bitfieldStack[stackPtr] = bitfield; // store current bitfeld
94        // update bitfeld for next row
95        bitfield = mask & ~(confStack[nextStackPtr].col |
96          confStack[nextStackPtr].posDiag | confStack[nextStackPtr].negDiag);
97        stackPtr = nextStackPtr;
98        continue;
99      } else {
100       // the last row has been reached and a valid configuration has been found
101       atomicAdd(&d_nQueensResult, 1);
102       bitfield = bitfieldStack[--stackPtr]; // move back one step
103       continue;
104     }
105   }
106 }
```

Listing 2: Complete source code of the condensed *DP2*-Kernel.

3.3 DP-3: Condensed kernel with dynamic search depth

One major drawback of the approach *DP-2* is that with a progression in depth, the number of branches to be analyzed by child grids decreases. As a consequence thereof, such child grids are assigned with much less work compared to their parent grids. As a countervailing measure, this method employs a dynamic search depth, which increases for every generation of child grids. That way, the decreasing width of subtrees is counterbalanced by a growing depth. Listing 3 illustrates the alterations that augment *DP-2* with dynamic search depth.

While multiple growth models exist ranging from linear growth over polynomial growth up to exponential growth, experimental results revealed that the simple model of doubling the search depth with each child grid works sufficiently well. The limiting factor for search depth is given by the amount of shared memory which is available. The upper bound is defined as follows:

$$depth_{max} = \frac{shared\ memory\ size_{max}}{size_{state} \times \#threads_{block}} - 1$$

For the evaluated target hardware *GK110*, which supports compute capability 3.5, shared memory can be configured up to a size of 48 kilobytes. For a block size of 256 threads, the search depth may not exceed 8 before the capacity is exceeded.

```
// ...
size_t stackEntrySize = sizeof(boardConfig_t) + sizeof(bitfield_t);
// double search depth for next child grid
// and make sure not to exceed shared memory capacity.
size_t childDepth = MIN( depth * 2,
  MAX_SHARED_MEM_SIZE / (blockDim.x * stackEntrySize) - 1);

nQueensKernelDynamic<N><<<numChildBlocks, blockDim.x,
  sharedMemSize, stream>>>(childConfigs, childDepth,
  numChildConfigs, true, freeCounter);
// ...
```

Listing 3: Building up on *DP-2*, adding dynamic search depth yields the *DP3*-Kernel.

3.4 DP-SWAP: Shared memory swapping kernel

Memory access and dynamic memory allocation are expensive operations. Next to the call overhead of kernels, they impose a major bottleneck. This approach builds up on top of *DP-3* and aims at reducing the frequency of memory access operations by introducing improved coalesced access patterns to leverage the full bandwidth of global memory.

This approach consists of three phases and is built on the idea of using the fast shared memory as a self-managed cache for operations on the global device memory. The first phase initializes the self-managed buffer (see Listing 4) and is issued before the actual computation. During the computation phase of the kernel, all threads in a block that find valid configurations place them in a designated memory region (see Listing 5). The last phase begins as soon as all threads of a block have finished the traversal of their corresponding subtrees. As Listing 6 indicates, all threads concertedly copy the contents of the shared memory to global memory. The first thread of a block subsequently launches a new grid.

```
// ...
extern __shared__ char sharedMemory[];

const uint stackSize = (depth + 1) *
  min(blockDim.x, numThreads - (blockIdx.x * blockDim.x));

boardConfigEx_t** swapMemory   = (boardConfigEx_t**)sharedMemory;
uint* numChildConfigs          = (uint*)(swapMemory + 1);
boardConfigEx_t* configStacks = ((boardConfigEx_t*)sharedMemory)+1;
boardConfigEx_t* childConfigs = configStacks + stackSize;

const uint bufferVolume = (MAX_SHARED_MEM_SIZE -
  sizeof(boardConfigEx_t) * (stackSize+1)) / sizeof(boardConfigEx_t);
const uint bufferThreshold = bufferVolume - stackSize;
// ...
```

Listing 4: The initialization phase of *DP-SWAP* allocates the buffer based on the memory demands of the configuration stack. Furthermore, the threshold for triggering the swapping phase is determined.

The major challenge for this approach is that the capacity of shared memory is very limited. Furthermore, a decent amount of memory is already used to record the state of each thread. As a result thereof, it may happen that a thread-block runs into a number of valid child configurations that exceeds the capacity of the shared memory buffer. In order to deal with this situation, all running threads have to be suspended and the swapping process has to be initiated once a certain filling level of the buffer has been reached. This necessitates a mechanism that is able to record the progress of running threads, so that their progress is not lost. An effective strategy for solving this issue is provided by extending the state of each thread with an additional bit-field, which indicates which configurations have already been checked by each thread.

```
// ...
if (stackPtr == depth) { // at end of subtree, store new child configs to SM
  childConfigs[atomicAdd(numChildConfigs, 1)] = configStack[stackPtr];
  bitfield = configStack[--stackPtr].visited;
  continue;
}
if (*numChildConfigs > bufferThreshold) { // buffer threshold reached
  while (stackPtr >= 0) { \\ copy all states from the stack
    childConfigs[atomicAdd(numChildConfigs, 1)] =
      configStack[stackPtr--];
  }
  break; // terminate search and proceed with swapping
}
// ...
```

Listing 5: During the computation phase, valid child configurations are placed in the buffer.

In the worst case, all threads have to store all of their stack entries. This worst case assumption defines the threshold, which when reached enforces a preemptive termination of all threads to start the swapping process and launch a child grid subsequently. Due to this preemptive termination mechanism, it is possible that child grids start at different depth levels of the search-tree. As a consequence, an adaptive search depth is not possible in this approach. In contrast to the previous approaches, only one child grid per block is launched instead of a per-thread basis. In order to fully exploit coalesced access patterns, all relevant data structures are stored using structures of arrays instead of arrays of structures. In conjunction with the swapping approach, this method extensively consolidates memory operations.

```
// ...
__syncthreads();
if (*numChildConfigs == 0) return; // no swapping necessary
uint numChildBlocks = (*numChildConfigs + blockDim.x - 1) / blockDim.x;
unsigned int* swapMemoryHeader = 0;
if (threadIdx.x == 0) { // allocate memory from thread 0
  swapMemoryHeader = malloc(sizeof(boardConfigEx_t) * (*numChildConfigs + 1));
  *swapMemoryHeader = numChildBlocks;
  *swapMemory = ((boardConfigEx_t*)swapMemoryHeader) + 1;
}
__syncthreads(); // copy configs to global memory
for (int i = 0; i < (*numChildConfigs / blockDim.x) + 1; ++i) {
  uint index = i * blockDim.x + threadIdx.x;
  // ...
  (*swapMemory)[index] = childConfigs[index];
}
__syncthreads();
if (threadIdx.x == 0) { // launch child grid
  // ...
  nQueensKernelSwapping<N>  <<<numChildBlocks, blockDim.x, SM_SIZE, stream>>>
    swapMemoryHeader, *numChildConfigs, childDepth, true);
}
```

Listing 6: In the swapping phase, all threads copy the contents of the buffer to global memory.

The major limitation of this approach is the limited size of shared memory, which results in a small buffer capacity for valid configurations. As soon as the capacity is exceeded, all running threads have to be terminated. This causes a decent waste of resources, since the results of compute tasks that did not deliver results in time are wasted. Furthermore, a larger number of configurations has to be copied. Dynamic swapping would help to alleviated this drawback. However, a proper implementation is not feasible with the currently available means of synchronization. The block-wide barrier `__synchthreads()` cannot be used, since many threads follow a different branch of execution. Another approach is the employment of atomic operations and active waiting to implement a custom barrier. The problem with this approach is that the *CUDA* runtime does not provide any guarantees that the execution of threads does not progress equally ? a fact which has been confirmed in a practical evaluation. As a consequence, it is currently not possible to implement dynamic swapping and abortively terminating threads remains as the only option.

4 Evaluation

This section provides an evaluation of the implementation approaches presented in Section 3. For that purpose, all implementations are benchmarked and compared to the approaches of Somers [21] and Feinbube et al. [8]. First, an overarching performance evaluation is provided covering all approaches. In the succeeding sections, a detailed discussion is provided for each approach. All measurements reported in this section were performed during exclusive time slots on the test system specified in Table 1. The NVIDIA *GK110* GPU was utilized, which employs the *Kepler* architecture that supports compute capability 3.5 and thus *Dynamic Parallelism*.

Table 1: Specifications of the test system.

Processor	2 × Intel Xeon E5620 (Westmere-EP)
Memory	6 × 4 GB PC3-10667 reg ECC DIMM
GPU compute device	2 × NVIDIA Tesla K20x [15]
Operating system	SuSE Linux Enterprise Server 11 SP2
CUDA version	5.5
GPU driver	331.20

4.1 Overarching performance evaluation

The performance of each approach was measured for a queen count in the range of $8 \leq N \leq 16$. Figure 8 illustrates the performance results for the approaches *DP-1*, *DP-2* and *DP-3* and compares them to both a serial CPU-based [21] and a parallel GPU-based [8] reference implementation. The central message that is conveyed by the measurements is that all implementations based on *Dynamic Parallelism* perform distinctly worse than the reference implementations. It appears as if the utilization of *Dynamic Parallelism* comes with high fixed costs, as all implementations perform especially poor for small N. While *DP-1* and *DP-2* are even outperformed by the serial CPU-based reference for the range of tested N, no implementation manages to outperform the GPU-based reference implementation. Although the implementation strategy of *DP-3* manages to provide decent performance improvements compared to *DP-1* and *DP-2*, it still lacks behind the performance of the GPU reference. Due to reasons of clearness, the measurements for *DP-SWAP* are supplemented in Figure 9. Unfortunately, *DP-SWAP* does not provide any performance improvements compared to *DP-3*. In contrary, it performs worse for all N with $N \geq 13$.

The utilization of *Dynamic Parallelism* depends a lot on using the correct parameters for tuning parameters such as search depth, block size or shared memory utilization. Many such parameters exist that often influence each other, which makes it hard to obtain an universal configuration

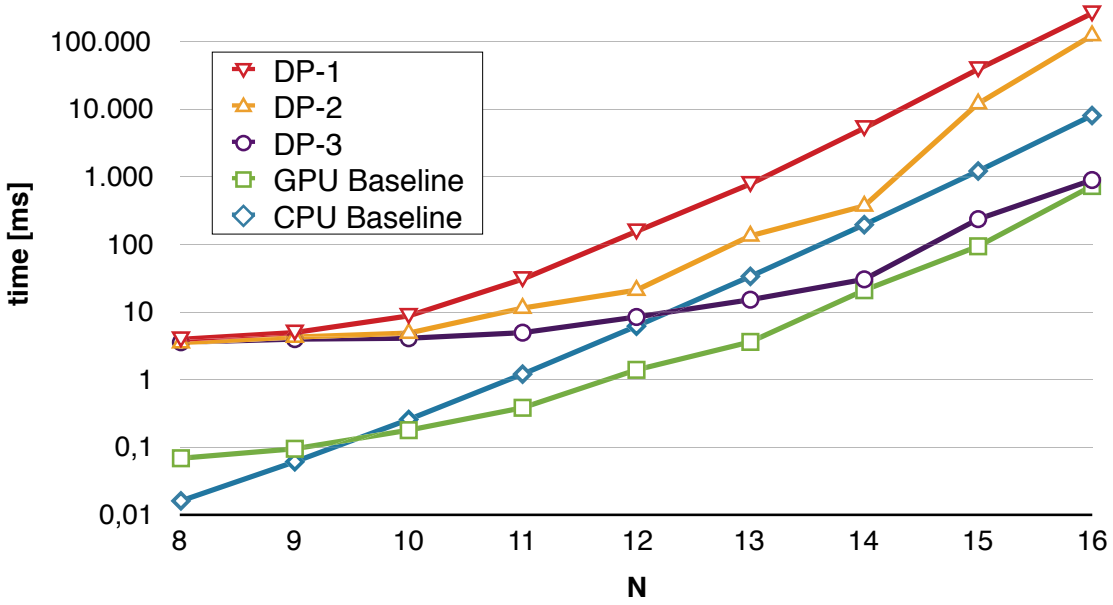

Figure 8: The approaches based on *Dynamic Parallelism* are not able to outperform the GPU reference [8], partially even the serial CPU reference [21]. Results are plotted on a log. scale.

that provides optimal performance. Many parameters could only be retrieved through experimental evaluation, which strongly limits the applicability to practical applications.

4.2 DP-1: Thread swarming kernel

A practical investigation was conducted to retrieve optimal operating conditions for the thread swarming approach with respect to stream utilization and search depth. Various tests across the tested range of values for N with $8 \leq N \leq 16$ were performed. Testing different values for the search depth parameter z revealed the best overall performance levels were obtained for $z = 3$.

Regarding the utilization of streams, the investigation attests a huge beneficial performance impact for using multiple streams and small N, whereas the advantage of using streams shrinks for large N (see Table 2). Since larger values of N result in a growing number of dynamically created threads, a possible explanation for the observed performance characteristics might be that the overhead caused by the numerous kernel launches consumes a predominant portion of the overall execution time. As a consequence, a balanced distribution of the actual computation fails to outweigh the costs of kernel launches.

Table 2: The performance impact of applying streams for multiple N is reported. The explicit use of streams in *DP-1* yields faster execution times various N. Measurements are reported in milliseconds.

N / streams	8	9	10	11	12	13	14	15
default stream	17	36	82	265	818	2.974	10.603	46.111
multiple streams	5	6	9	32	155	780	5.232	39.163

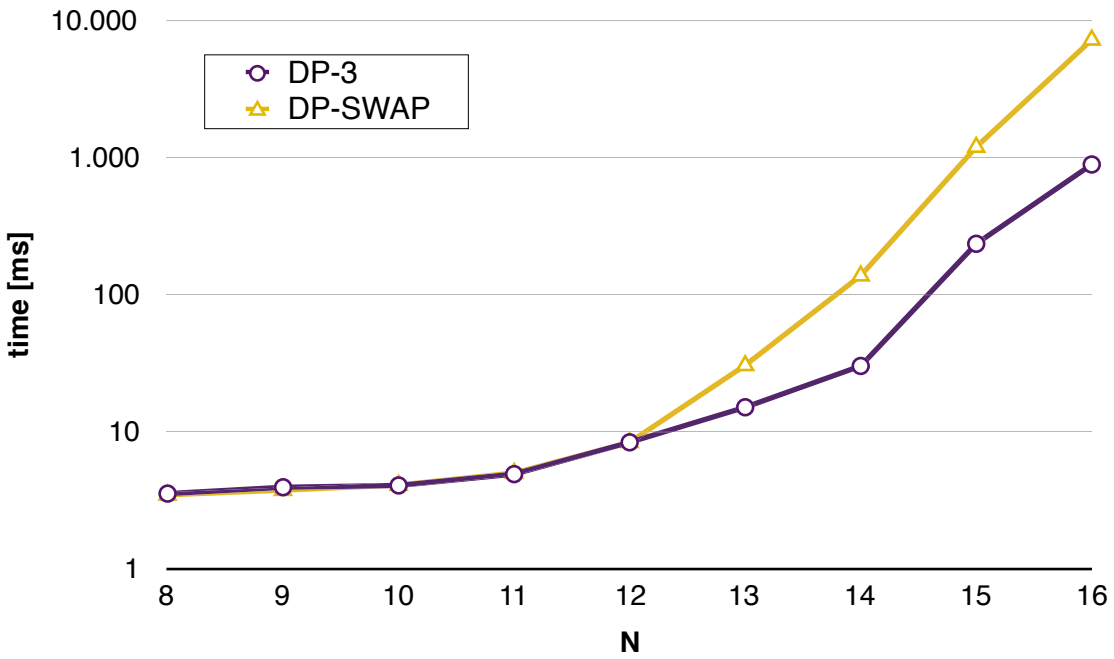

Figure 9: The swapping approach fails to gain speed-up over *DP-3* for all N. Increased shared memory consumption and static search depth are likely causes. Results are plotted on a log. scale.

4.3 DP-2: Condensed kernel with static search depth

The condensed approach with static search depth improved performance compared to *DP-1*, which is a result of increasing the amount of work that is processed per thread. However, a certain degree of unbalanced workload distribution is effected by the static search depth. As reported in Table 3, grids that are launched at progressed row numbers start upon fewer valid configurations and only few threads can progress to deeper rows. However, increasing the search depth per grid increases shared memory consumption and reduces the number of blocks that can reside on a single multiprocessor. Thus, multiprocessors have fewer opportunities to apply optimizations such as latency hiding.

4.4 DP-3: Condensed kernel with dynamic search depth

Having implemented a dynamic search depth for child grids resulted in a much more homogeneous workload distribution as illustrated in Table 3. This is reflected by the distinct performance gains over *DP-2*, which uses a static search depth for all grids. However, finding the correct growth function and the optimal value for the initial search depth required much effort.

The choice of the initial search depth and the growth function heavily depends on the structure of the search-tree, and thus is very problem dependent. For the *N-Queens* problem, the number of branches drastically decreases at increasing depths. As a consequence, applying an exponential growth function worked well for the problem. Correspondingly, other problems that come with increased branch counts at deeper levels would profit from a degressive growth function.

4.5 DP-SWAP: Shared memory swapping kernel

Contrary to the expectations, introducing massively coalesced memory access patterns did not result in improved performance. The main reason for this outcome is the increased shared memory consumption, the increased number of configurations that have to be passed to child grids and last but not least the lack of an adaptiv search depth. An undesired side-effect of this optimization approach

is that the increased shared memory consumption decreases the occupancy of multiprocessors and thus effectively prevents latency hiding.

For the approaches *DP-1*, *DP-2* and *DP-3*, exceeding the upper limit for nested kernel launches can be effectively prevented by choosing appropriate values for the search depth parameters. For this approach however, there are no guarantees that the limit is not overstepped. In the worst case it could happen that the initial configuration of a thread is carried on across several child grids, as the assigned thread is always terminated before it can finish its task. The risk for such situations increases for small search depths and large problem sizes.

Table 3: Indicated are the number of valid child configurations (min, max and average), that have to be processed by child grids for ($N = 12$). Applying dynamic search depth results in a more balanced number valid child configurations.

approach	initial row	search depth	min	max	average
static	1	4	1196	1597	1401.33
static	5	4	1	60	7.97
dynamic	1	2	59	73	63.00
dynamic	3	4	65	255	145.22

5 Conclusion

This work investigates novel approaches for improved workload distribution using *Dynamic Parallelism* for fine-grained, irregular workloads. A case study was conducted using the *N-Queens* problem as an exemplary workload. For the given problem, an evaluation of our approaches revealed that the benefits of improved workload distribution were outweighted by the overhead caused by nested kernel invocations and dynamic memory allocations. As a consequence thereof, none of the evaluated strategies managed to outperform the GPU-based reference implementation [8] and some approaches even lagged behind the performance of the serial CPU-based reference [21].

However, many aspects of the presented strategies are general enough to be applied to other search-tree or backtracking based algorithms. Approach *DP-2* investigated a novel memory management strategy for passing large data volumes to child grids. The method handles memory allocations on a per-block level to provide simplified access patterns for child grids and to reduce the overall number of allocation operations. Furthermore, a comparison between the condensed approaches using static and dynamic search depth (*DP-2* and *DP-3*, respectively) demonstrated, that evenly distributed workloads result in decent performance gains. Additional strengths and the applicability of the presented approaches were discussed in Section 4. Further implementation details have been documented in the masters thesis by Frank Schlegel [20].

For all means of optimizations, it was crucial to manually tune parameters like block size, shared memory size or search depth. This is a major hurdle for implementing portable code that performs well on different hardware as well as for varying kinds of input. Especially for fine-grained workloads, the developer has to be careful not to counteract on essential means of optimization such as latency hiding. Overall, the most notable obstacles faced with *Dynamic Parallelism* are the high overhead costs of launching nested kernels and the limited amount of shared memory for assembling parameters for child grids. However, these bottlenecks will most likely be alleviated by future hardware generations.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, (January 2007), 2007.

[2] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, January 2009.

[3] Max Bezzel. Proposal of Eight Queens Problem. *Berliner Schachzeitung*, 3:363, 1848.

[4] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64. Eurographics Association, 2008.

[5] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. Dynamic task parallelism with a gpu work-stealing runtime system. In *Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2013.

[6] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[7] Cengiz Erbas, Seyed Sarkeshik, and Murat M Tanik. Different perspectives of the N-Queens problem. In *Proceedings of the 1992 ACM annual conference on Communications*, CSC '92, pages 99–108, New York, NY, USA, 1992. ACM.

[8] Frank Feinbube, Bernhard Rabe, Martin von Löwis, and Andreas Polze. NQueens on CUDA: Optimization Issues. In *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing*, ISPDC '10, pages 63–70, Washington, DC, USA, 2010. IEEE Computer Society.

[9] L R Foulds and D G Johnston. An Application of Graph Theory and Integer Programming: Chessboard Non-Attacking Puzzles. *Mathematics Magazine*, 57(2):pp. 95–104, 1984.

[10] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. Scalable SIMD-parallel memory allocation for many-core machines. *The Journal of Supercomputing*, 64(3):1008–1020, September 2011.

[11] Stephen Jones. Introduction to dynamic parallelism. Presentation at GPU Technology Conference (GTC), 2012.

[12] Khronos OpenCL Working Group. The OpenCL Specification, Version 2.0, 2013.

[13] K Kise, T Katagiri, H Honda, and T Yuba. Solving the 24-queens Problem using MPI on a PC Cluster. Technical report, Graduate School of Information Systems, The University of Electro-Communications, 2004.

[14] Christian Lauterback, Qi Mo, Dinesh Manocha, and Chapel Hill. Work distribution methods on GPUs. 2009.

[15] NVIDIA Corporation. Nvidia Tesla K20X GPU Accelerator – Board Specification, 2012.

[16] NVIDIA Corporation. CUDA C Programming Guide, March 2015.

[17] Thomas B. Preußer, Bernd Nägel, and Rainer G. Spallek. Putting Queens in Carry Chains. Technical report, Technische Universität Dresden, 2009.

[18] Martin Richards. *Backtracking Algorithms in MCPL using Bit Patterns and Recursion.* Computer Laboratory University of Cambridge, 1997.

[19] Timothy J Rolfe. A specimen MPI application: N-Queens in parallel. *ACM SIGCSE Bulletin*, 40(4):42–45, November 2008.

[20] Frank Schlegel. *Lastbalancierung auf GPUs mittels Dynamic Parallelism.* Masters thesis (in german), Hasso Plattner Institute for Software Systems Engineering, University of Potsdam, July 2014.

[21] Jeff Somers. The N-Queens Problem - a study in optimization. http://jsomers.com/nqueen_demo/nqueens.html, 2002.

[22] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[23] Krishnahari Thouti and S. R. Sathe. Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues. In *2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, pages 806–810. IEEE, December 2012.

[24] Stanley Tzeng, Brandon Lloyd, and John D. Owens. A GPU Task-Parallel Model with Dependency Resolution. *IEEE Computer*, 45(8):34–41, 2012.

[25] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 120–126, Houston, Texas, 2013. ACM.