Optimization of *N*-Queens Solvers on Graphics Processors

Tao Zhang[1,2], Wei Shu[2], and Min-You Wu[1]

[1] Shanghai Jiao Tong University, Shanghai, China
{tao.zhang,mwu}@sjtu.edu.cn
[2] University of New Mexico, Albuquerque, USA
shu@ece.unm.edu

Abstract. While graphics processing units (GPUs) show high performance for problems with regular structures, they do not perform well for irregular tasks due to the mismatches between irregular problem structures and SIMD-like GPU architectures. In this paper, we explore software approaches for improving the performance of irregular parallel computation on graphics processors. We propose general approaches that can eliminate the branch divergence and allow runtime load balancing. We evaluate the optimization rules and approaches with the n-queens problem benchmark. The experimental results show that the proposed approaches can substantially improve the performance of irregular computation on GPUs. These general approaches could be easily applied to many other irregular problems to improve their performance.

Keywords: GPU, N-queens, Irregular, Divergence, Load Balancing.

1 Introduction

GPUs have been used in various computing areas like molecular dynamics, astrophysics simulation, life sciences, MRI reconstruction and so on, achieving more than 100x speedups over their CPU counterparts [1]. However, some irregular applications, like 3D-lbm and gafort, are not so suitable for executing on GPUs, showing poor speedups [2]. The programs of irregular applications generally contain complex control flow, which causes irregular memory access, branch divergence, and load imbalance. GPUs suffer branch divergence because they are composed of one or more SIMD-like streaming processors (named StreamMultiprocessors or SMs by Nvidia) which require their scalar pipelines to execute the same instructions together. Although general optimization rules and techniques are still more or less effective for solving irregular problems on GPUs, more insight and approaches developed specifically for irregular problems are imperative for further improvement of performance.

In this paper we focus on investigating and solving the performance issues found by implementing and analyzing different version of the n-queens problem benchmark on the Nvidia GTX480 GPU. The computation structures of the n-queens problem are highly irregular. Different sizes of the problem have different

O. Temam, P.-C. Yew, and B. Zang (Eds.): APPT 2011, LNCS 6965, pp. 142–156, 2011.

parallelism and thread granularity, which provide desired inputs for our study on memory access, branch divergence and load balance. Although our final n-queens kernel might be the fastest n-queens solver on GPUs, we do not aim only at optimizing this benchmark but instead at devising general rules and approaches for all irregular applications. Besides the proposed optimization approaches, we also present many useful analyses and interesting discussions, and give details on how to apply the proposed optimization approaches on other irregular applications. Overall, our major contributions are as follows:

- We propose an IRRC (Iteration-Related Reorganization of Computation) approach to reorganize the computation inside loop structures to reduce branch divergence on GPUs. The approach is simple to apply and incurs no extra overhead in execution time.
- We propose a distributed job pool approach and a monolithic job pool approach to balance the load at runtime. The two approaches are the first to address the load imbalance on GPUs at thread level. They are easy to use and need no extra memory.

2 Related Work

GPUs have already proved useful on regular problems like matrix multiplication and so on. Recently, much research has focused on solving irregular problems on GPUs [3,4,5,6,7,8] in order to exploit the power of GPUs on a wider range of applications. In all the issues found in previous work, branch divergence and load imbalance are the major issues that cause reduction of performance.

Branch divergence has long been a consideration in the design of GPU architectures [9,10] as well as GPU applications [2]. This issue can be addressed by hardware or software approaches. Fung et al. [10] proposed a hardware approach to form and schedule new warps dynamically every cycle. The basic idea is to group the threads taking the same code branches into the same warps so that the branch divergence is reduced. Zhang et al. [2] presented a sophisticated software method to eliminate thread divergence through runtime thread-data remapping. Their method is effective for those applications that expose data-branch patterns, but the remapping process incurs some overhead. Our approach is to adjust the computation inside the iterations, which is simple to use and introduces no overhead.

Load balancing on GPUs has become a recent topic because of the advent of more elegant synchronization hardware components such as atomics, and the need to exploit the power of GPUs on more complex graphics rendering or scientific computation. Previous approaches to load balancing work on two levels: block-level [3] and warp-level [4,5]. Block-level load balancing moves the work between hardware processors while warp-level balancing adjusts the work between warps (a warp is a group of a certain number of threads, often 32, that executes on a single streaming processor). The imbalance they try to address is caused by uneven distribution of aggregate jobs (jobs and new jobs generated during execution) among hardware processors. However, the n-queens problem doesn't

generate new jobs during execution. Instead the load imbalance is introduced from the variance of thread granularity within the same warp. This imbalance cannot be resolved using existing approaches, hence we propose new approaches to address this issue.

The n-queens problem is a typical constraint satisfaction problem. The ultimate goal of the n-queens problem is to find distinct solutions to place N queens on an $N \times N$ chess board. This problem has useful applications in a wide range of areas: parallel memory storage approaches, image processing, VLSI testing, traffic control, deadlock prevention, physical & chemical studies and networks [11]. The total solution cost for the n-queens problem increases exponentially with N [12]. As a result, it is placed in the NP (Non-Deterministic Polynomial) complexity class [13]. Currently the solution count for the n-queens problem for N larger than 26 is still unknown. The "QUEENS@TUD" project carried out by the TUD university found the solution count for the 26-queens problem with 9 months' computation effort on massively-parallel FPGA-based devices. This result was confirmed later by the Russian "MC#" project which solved the 26-queens problem on two supercomputers of the Top500 list [14]. The n-queens problem was also studied as an asychronous/irregular application in computer architecture literature. Shu et al. [15] proposed a high-level runtime support system (the P Kernel) to run loose asynchronous and asynchronous applications including the n-queens problem on an SIMD machine MasPar MP-1. Similar to Shu's work, Blas et al. [16] designed a low-level program coding methodology for the same purpose.

3 Design and Implementation of the Naive N-Queens Kernel

To solve the n-queens problem on parallel GPU cores, we can partition the problem into a number of disjoint sub-problems by placing k queens in the top k rows on a $N \times N$ board. Then each sub-problem corresponds to one valid placement. The symmetry feature of the n-queens problem is considered to reduce the computation task by around half [17]. To save memory space, each sub-problem is represented by three 32-bit unsigned integers that record the aggregate effect of all the rows above on three directions: vertical, left-diagonal and right-diagonal. Usually, the workload of partitioning is much smaller than that of processing the sub-problems, provided that k is relatively small. Therefore, the partitioning of the problem is carried out on the CPU, and the generated sub-problems are then processed on the GPU. In this work, we focus on the study of the workload on the GPU. In Nvidia's convention, the part of code invoked by a CPU and executed on a GPU is called the *kernel*. Figure 1 shows the pseudocode of the naive n-queens kernel.

The register variable "rowIndex" represents the index of the current row, with initial value 0. The "job_data" points to the memory space in global memory that stores the sub-problems. Each sub-problem occupies three 32-bit unsigned integers. The "results" points to the memory space in global memory to write

```
__global__ void nqueen_kernel_0(*job_data, *results, *work_space... )
{
   __register__  rowIndex, solution;

   each thread fetches a task from job_data into its array ROW[ ] in work_space;
   while(rowIndex >= 0) {
      if (no position to place new queen in ROW[rowIndex]) { rowIndex--; }
      else{
         finds a valid position P in ROW[rowIndex];
         places a queen at P in ROW[rowIndex] and mark the position as occupied;
         if (reaches last row) { solution++; }
         else{
            generates ROW[rowIndex+1] based on Row[rowIndex] and the position  P;
            rowIndex++;
         }
      }
   }

    reduction of the solutions of the threads within each block;
}
```

Fig. 1. Pseudo Code of the Naive N-queens Kernel

Table 1. Results of the Naive N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.40	1.46	11.03	78.51	664.13

back the total solution count. The "work_space" points to the memory space in global memory for threads to perform computations. Each GPU thread has its own portion in the "work_space", which can hold forty-four 32-bit unsigned integers. At the beginning of the kernel execution, each thread acquires a job (a sub-problem of the n-queens as described above) from the "job_data" array in global memory. Then threads enter a while loop to search for all valid solutions. The performance of the naive n-queens kernel on different N is shown in Table 1. The time in all the tables thereafter is the time for executing the kernels on the GPU, which does not include the time for the CPU to partition the n-queens problem into sub-problems.

4 Optimization

4.1 Optimal Usage of Memory

Enlarge the L1 Cache. In the naive n-queens kernel, the "job_data", "results", and "work_space" are all allocated in the global memory of the GPU. The "work_space" will be read and written many times by threads. For instance, there will be more than 4.76E+10 32-bit accesses (read or write) to the "work_space" during the computation for the 18-queens problem according to our analysis. We noticed that modern GPUs like the Nvidia GTX480 have shared memory and L1

Table 2. Results of the Naive N-queens Kernel with 48KB L1 Cache

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.37	1.26	9.54	67.94	575.54
Speedups	1.08	1.16	1.16	1.16	1.15

cache that share a configurable space. By default, the GTX480 is configured to have 48KB shared memory along with 16KB L1 cache in each SM. By configing to 48KB L1 cache with 16KB shared memory in each SM, the performance result is shown in Table 2. The speedups refer to the time ratios of the naive kernel with 16KB L1 cache (Table 1) versus the naive kernel with 48KB L1 cache. On average, the kernel runs 1.14x faster thanks to the larger L1 cache.

Utilize the Shared Memory. Registers and Shared memory are much faster than global memory because they are on-chip (around 4, 40, and 440 clock cycles per access respectively). Therefore we revised the naive n-queens kernel to move the "work_space" from global memory to shared memory. The GPU was changed back to have 48KB shared memory and 16KB L1 cache in each SM. The performance result is shown in Table 3. The speedups refer to the time ratios of the naive n-queens kernel with 48KB L1 cache (Table 2) versus this new kernel that utilizes the shared memory. On average, the kernel runs 4.49x faster because of the fast-access shared memory.

Table 3. Results of the Improved N-queens Kernel Using Shared Memory

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.08	0.27	2.13	15.51	134.23
Speedups	4.63	4.67	4.48	4.38	4.29

Solve the Memory Bank Conflict. In the improved n-queens kernel that uses shared memory, the "ROW" array allocated in the shared memory has two dimensions. The first dimension is a multiple of 32 while the second dimension is not. This memory layout causes bank conflict [18] and hence performance drops when threads access this array. This issue can be addressed by exchanging the first and the second dimension of the array. The performance result is shown in Table 4. The speedups refer to the time ratios of the improved kernel using shared memory with bank conflict (Table 3) versus the new kernel with no bank conflict. On average, eliminating the bank conflict yields a 1.24x speedup.

Table 4. Results of the Improved N-queens Kernel Using Shared Memory with no Bank Conflict

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.22	1.74	12.82	112.20
Speedups	1.33	1.23	1.22	1.21	1.20

4.2 Reducing Divergence with the IRRC Approach

The loop structures in program code fall into two categories: regular and irregular, as shown in Figure 2. A regular loop has a definite number of iterations while the number of iterations of an irregular loop depends on certain conditions. During each iteration of a loop, threads that take different code branches have to waste time on waiting for others due to the mechanism that GPUs use to handle branch divergence [9].

Regular loop:

```
For(int i=0;i<10;i++)
{
   ···;
}
```

Irregular loop:

```
while(i>0)
{
   if(con) { i--;}
   else{
        ···;
        i++;
}
```

Irregular loop:

```
while(True)
{
   ···;
   if(cond1) {break;}
   else{
        ···;
        if(cond2) {break;}
    }
```

Fig. 2. Regular and Irregular Loop Structure

As shown in the n-queens kernel in Figure 1, the main part of the kernel is an irregular while loop. At each iteration, the threads within a warp begin the computation on their own data which is a sub-problem. When the threads within a warp take different code paths, branch divergence occurs. For the 18-queens problem, we measured the ratio of divergent branches to be 22.6% with the Nvidia Compute Visual Profiler. Moreover, the numbers of iterations of threads increase exponentially with N, so the divergence could lead to a substantial negative impact on performance.

If we can reorganize the work of each iteration to let threads within each warp have more chance to execute together on the same code branches, performance will be better. After studying, we found that the reorganization of work can be done with three methods: adding tail checking to bypass some waiting iterations; combining multiple loop sentences into a single loop sentence; changing an invocation of a kernel in a loop on the CPU side into a loop structure inside the kernel. Since the methods are all related to iteration, we name this divergence-reducing approach the "IRRC" (Iteration-Related Reorganization of Computation) approach. Following the IRRC approach, we devised a new low-divergence n-queens kernel, as shown in Figure 3.

In this low-divergence n-queens kernel, we reordered the condition testing and added a tail check (line 14-16). In the naive n-queens kernel, we followed the pattern of "if cond then X else Y", where part X is much shorter than part Y. At an iteration, if some threads within a warp need to execute part X while others need to execute part Y, the warp has to serially execute part X followed by part Y [9]. Thus, the threads executing part X are idle for a large fraction of time without being able to run possible part Y in the next iteration. In this

```
__global__ void nqueen_kernel_2(*job_data, *results ... )
{
  __const__     tid; //The index of the thread within the block
  __register__  rowIndex, solution;
  __shared__    ROW[MAX_ROW][BLOCK_SIZE];

  each thread fetches a task from job_data into its array ROW[MAX_ROW] [tid];
   for(; rowIndex >= 0; rowIndex--) {
     if ((P = new position) is a valid position ){
       places a queen at P in ROW[rowIndex][tid] and mark the position as occupied;
       if (not reach last row){
           generates ROW[rowIndex+1][tid] based on Row[rowIndex][tid] and the position  P;
           rowIndex++;
           if (row rowIndex has empty candidate positions){
              rowIndex++;
           }
        }
        else{
          solution++;
        }
     }
  }

  reduction of the solutions of the threads within this block;
}
```

Fig. 3. Pseudo Code of the Low-divergence N-queens Kernel

low-divergence implementation, we move the condition testing into the tail of the earlier iteration. Though every thread needs to spend time for parts Y and X, threads are potentially allowed to bypass one waiting iteration, thus the branch divergence is reduced.

With this low-divergence n-queens kernel, the ratio of divergent branches to total branches dwindles to 0.000846% for the 18-queens problem. The performance of this kernel is presented in Table 5. The speedups refer to the time ratios of the improved kernel utilizing shared memory with no bank conflict (Table 4) to this new low-divergence n-queens kernel. On average, a 1.30x speedup is achieved through divergence reduction.

Table 5. Results of the Low-divergence N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.05	0.17	1.30	9.60	84.05
Speedups	1.20	1.29	1.34	1.34	1.33

How to Apply the IRRC Approach. Like the approach proposed in [2], the IRRC approach also requires an understanding of the algorithms in the application. The general rule is to reorganize the kernel code inside loops especially irregular loops, such that threads have more chances to take the same code branches. The three methods introduced for reorganization can be used separately or jointly.

4.3 Balancing the Load at Thread Level

As discussed previously, each thread of the n-queens kernel processes a subproblem that has a different number of solutions. Therefore, each threads will execute a different number of iterations. Consequently, the grain sizes of the threads vary. To verify this assumption, we measured the number of iterations and the clock cycles of the 7.41 million GPU threads from the 18-queens problem. The measurement is done inside the for loop of each thread with the low-divergence kernel. The result is as follows.

Table 6. Granularity Variance in the 18-queens Problem

Thread Granularity	Min	Max	Average	Variance	Standard Deviation
Iterations	2.00E+00	4.75E+04	6.51E+03	1.45E+07	3.81E+03
Clock Cycles	1.03E+03	2.80E+07	4.19E+06	6.12E+12	2.47E+06

As shown in Table 6, the n-queens problem exhibits heavy irregularity. GPUs are unable to balance the load at thread level. Because of the converge mechanism for branch divergence [9,10], threads with fewer iterations have to wait for the threads with more iterations within the same warp before exiting a loop structure.

We define the utilization μ_{util} of an SM as follows:

$$\mu_{util} = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} C(i,j)}{n \times \sum_{j=1}^{m} L(j)} \tag{1}$$

where n denotes the number of threads in each warp, m denotes the number of warps in the SM, $C(i,j)$ denotes the clock cycles used in processing jobs by thread i of warp j, $L(j)$ denotes the clock cycles used by the longest thread in warp j. We sampled 100 warps in the low-divergence n-queens kernel for the 18-queens problem. In these warps, the minimum, maximum and average μ_{util} were 38.39%, 73.92%, and 55.15% respectively. So, the theoretical upper-bound of the speedup with an ideal load-balancing approach is 1.81. Since the granularity of each thread cannot be known prior to the execution, the load balancing of threads can only be done at runtime.

Based on all these analyses, we propose a distributed job pool approach to balance the load at thread level at runtime, as illustrated in Figure 4. Each block in a SM occupies a job pool in global memory that contains a predefined number of jobs. Each thread in a block is assigned an initial job and can get a new job upon finishing. In the end, some threads consume more jobs while others consume fewer, but the overall execution times of all threads within a block/warp are close to each other, hence the load is balanced. The next job in the job pool is controlled using a semaphore. GTX480 supports atomic operations on variables in shared memory or global memory for mutexes.

The pseudocode of the n-queens kernel that utilized this approach is shown in Figure 5. A new variable named *seek* is allocated in the shared memory to point

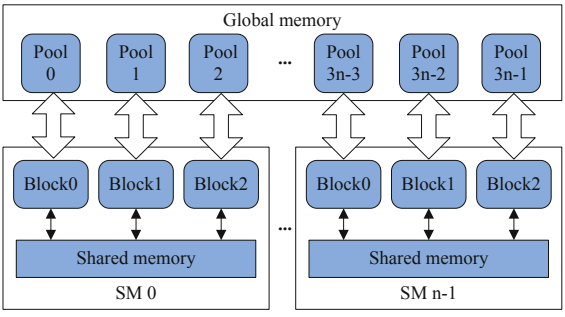

Fig. 4. Distributed job pool Approach

```
1  __global__ void nqueen_kernel_3(*job_data, *results ... )
2  {
3     __const__      tid; //The index of the thread within the block
4     __register__   rowIndex, solution, index;
5     __shared__     ROW[MAX_ROW][BLOCK_SIZE];
6     __const__       upper_bound = the upper bound of the job-pool for this block;
7     __shared__      seek;
8
9     if(tid == 0) { set seek to point to the next new job in the job-pool for this block;}
10    each thread fetches a task from job_data into its array ROW[MAX_ROW] [tid];
11    for(; rowIndex >= 0; rowIndex--) {
12
13       ····· //the same code as in the low-divergence n-queens kernel is omitted
14
15       if (rowIndex == 0) { //current job is done.
16          index = atomicAdd(&seek,1); //get index of new job
17          if (index exceeds pool upper bound)
18             break;
19          else{
20             gets this job by index from the job-pool as the new job of this thread;
21             rowIndex++;
22          }
23       }
24    }
25
26    reduction of the solutions of the threads within this block;
27 }
```

Fig. 5. Pseudo Code of the job pool N-queens Kernel

to the next job in the job pool. The upper part of the code inside the for loop is the same as in the low-divergence n-queens kernel. When a thread finishes its current job (if *rowIndex* == 0), it attempts to fetch a new job in three steps as follows:

- Get the index of the next job through an atomic operation on *seek*.
- Verify this index and break the for loop if the index is out of the range of the job pool (all jobs in the pool are processed).

– Fetch this new job from the job pool, increase *rowIndex* and continue the for loop.

The performance of the n-queens kernel with job pool is presented in Table 7. The speedups refer to the time ratios of the low-divergence kernel (Table 5) to this new n-queens kernel with job pool. All n-queens problem run faster except for the 15-queens problem. The 15-queens problem is a small problem that has a parallelism of only 7432. The job pool approach will decrease the number of threads launched on the GPU since each thread now consumes more than one job. As a result, the number of threads launched for the 15-queens problem cannot fully utilize the hardware capacity, hence the decrease of performance. However, this is not a problem for middle or large scale problems because they have adequate parallelism.

The performance of the 18-queens problem with different sizes of job pool is presented in the following Figure 6. The discussion is as follows.

Overhead Analysis. 1. Time overhead. When one thread does the atomic operation and then fetches a new job, all other threads within its warp will be halted, causing branch divergence. The overhead depends on the clock cycles to fetch a new job including the atomic operation, and the number of times this part of the code is executed. It is mandatory to condense this part of the code to reduce the time overhead. 2. Memory overhead. The job pool can be constructed by dividing the original memory space of data into the number of pools, or by

Table 7. Results of the Distributed job pool N-queens Kernel

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.16	0.98	6.47	53.61
Speedups	0.83	1.06	1.33	1.48	1.57

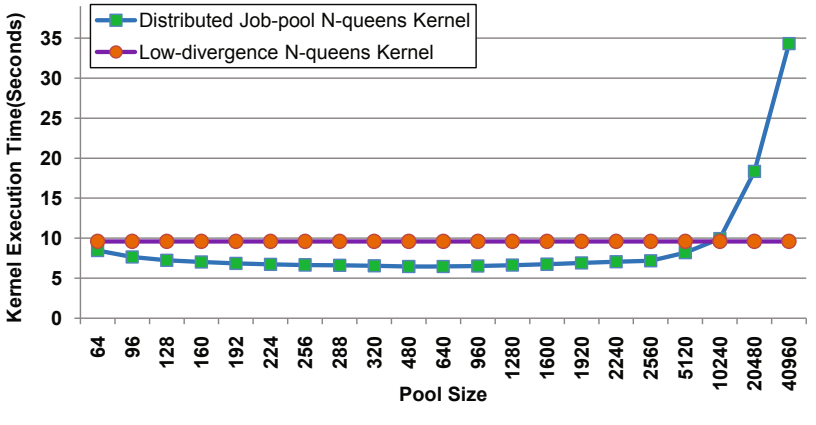

Fig. 6. Execution Time of the 18-queens Problem with Different Sizes of job pool

allocating new memory space. The former has the advantage of saving memory space and memory copying time, therefore is adopted in our solver.

Optimal Size of the Job Pool. The size of the job pool refers to the number of jobs inside. For distributed job pool approach, each block will have a job pool. The size of the job pool is a trade-off: larger pool size may bring a better balance in granularity of threads and thus less waste in waiting. However, the expense to perform load balancing also increases because threads then fetch jobs from the pool for more times, so there is more branch divergence. As shown in Figure 6, the n-queens kernel with distributed job pool runs faster than the low-divergence n-queens kernel for the 18-queens problem with any pool size from 64 to 5120. However, using a pool size like 10240 or larger would cause the overhead of load balancing to outweighs the benefit. In such a case the load balancing approach is no longer profitable.

Distributed Job Pool vs. Monolithic Job Pool. In the distributed job pool approach, there are multiple job pools, one for each block. The total jobs are divided into these job pools at runtime before threads begin taking from them. Therefore, the balance reached is a local balance within each block. We propose another approach where all blocks share a single global job pool. In this monolithic job pool approach, threads from all blocks acquire jobs from that pool, therefore a global balance is reached. However, this approach has more contention for the atomic semaphore than the distributed job pool. Besides, the atomic operation could become expensive because the atomic semaphore needs to become global and resides in the slow global memory. In general, the monolithic job pool approach has its advantages and disadvantages. The result for the n-queens kernel with a monolithic job pool is shown in Table 8. The performance is a little better than using the distributed job pool approach. The Overall Speedups refer to the time ratio of the naive kernel (Table 1) to this new kernel with monolithic job pool.

Table 8. Results of the N-queens Kernel with Monolithic job pool

Problem	15-queens	16-queens	17-queens	18-queens	19-queens
Time (sec.)	0.06	0.15	0.86	6.15	51.31
Overall Speedups	6.67	9.73	12.83	12.77	12.94

How to Apply the Job Pool Approach. GPU kernels for solving irregular applications generally exhibit more or less variance in thread granularity. Some show the variance explicitly while others may need some code modification to expose it. The code modification can be made with the three code reorganization methods proposed in section 4.2. In general, the piece of code for load balancing needs to be placed at the end of an irregular loop (refer to section 4.2 for irregular loop), to enable threads to fetch new jobs upon finishing their work. Also, several new variables are needed for the operation of the load balancing code.

4.4 Performance Summary

The following Figure 7 compares all the results. The n-queens kernels with distributed job pool or Monolithic job pool have similar performance, and take the least execution time among all kernels. As shown in Table 8, the average speedup of the Monolithic job pool n-queens kernel over the naive n-queens kernel is around 13 for 17-queens, 18-queens, and 19-queens problem. This shows that appropriate implementation and optimization techniques that consider the characteristics of applications and hardware architectures are crucial for performance of irregular applications. The 15-queens problem and the 16-queens problem have relatively lower speedups of around 7 and 10 respectively, partly because they have smaller parallelism, smaller granularity, and consequently less room to improve.

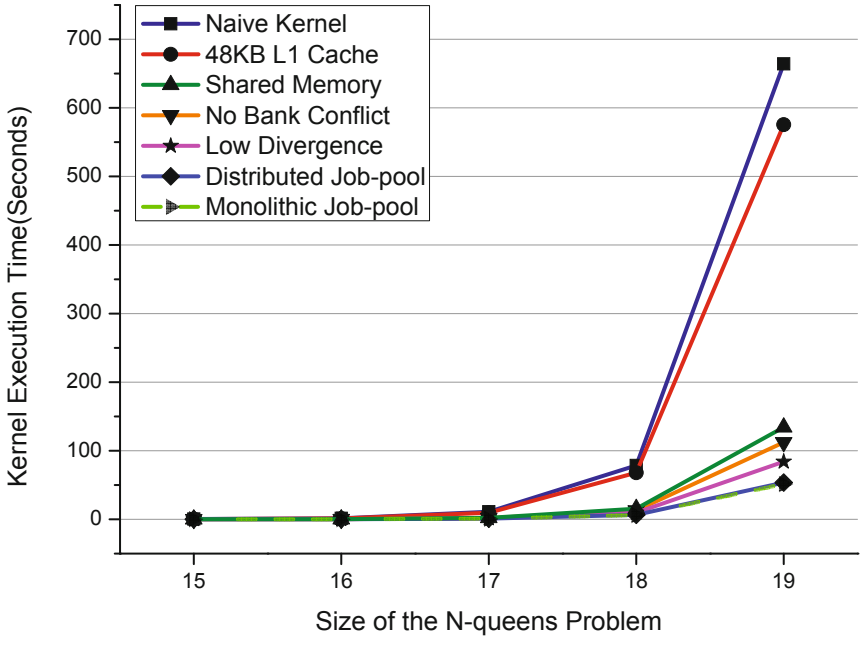

Fig. 7. Execution Time of All Kernels

5 Discussion

Besides the experiments in this work, we also applied the proposed methods on other irregular applications such as an application to calculate the potential distribution of a rectangular waveguide using the montecarlo method. We also got around 1.2x speedup by applying one of the methods to reduce divergence: combining one regular loop with one irregular loop. In general, the proposed approaches in this work can be used separately or combined to reduce divergence.

The proposed distributed or monolithic job pool approaches can balance the load of threads at some time cost and need no extra memory. These approaches are especially useful for coarse-grained threads with remarkable granularity variance. In some dynamic and/or irregular applications, the program size/space can be quite large or unpredictable. Therefore the applications often employ course-grained threads to search in the solution space. The job pool approaches would be useful for such applications.

Although Table 8 shows that the n-queens kernel with a monolithic job pool works faster than that with a distributed job pool, the distributed job pool might be a better choice in some circumstances. In GPUs other than the GTX480 that lack cache, the access to the atomic semaphore in global memory for the monolithic job pool approach might become an expensive operation. Moreover, the distributed job pool approach has better scalability than the monolithic job pool approach. The monolithic job pool approach introduces more contention for the atomic semaphore, and the contention increases with the number of simultaneous threads. Current hardware like Nvidia GPU, AMD GPU and Intel Larrabee can hold more than twenty thousand simultaneous threads on a single chip. We envision that in future that this number will be doubled or tripled often with architectural advancements.

6 Conclusion

We optimized the memory access of the n-queens problem benchmark, and found that the performance was on average 6.34x better with shared memory than using global memory across different sizes of the n-queens problem. Also, we proposed the IRRC (Iteration-Related Reorganization of Computation) approach to reduce branch divergence based on our categorization of the loop structures. By employing this approach, the divergence ratio was reduced remarkably in our experiment. The irregular computation achieved an average 1.30x further speedup over the n-queens kernel that had optimized memory access. Finally, we proposed two approaches to balance the thread load at run time: distributed job pool approach and monolithic job pool approach. These two approaches had similar performance in our experimental evaluation. With the load balancing approaches, the irregular computation ran 1.26x faster on average than the low-divergence n-queens problem kernel. For all the proposed approaches, we introduced their principles and analyzed the factors that affect their effectiveness. Especially, for the two job pool approaches, we made extensive analyses on the time and memory expenses, the optimal size for the job pool, and the performance differences.

Our work is the first to reduce the branch divergence based on our classification of loop structures, and the first to identify and address the load imbalance at thread level at runtime. Together the IRRC approach and the job pool approaches can indeed effectively leverage the impressive computation power of graphics processors. All other irregular applications on GPUs can benefit from the proposed general approaches. As part of our future work, we will apply these

approaches on other dynamic and/or irregular problems, such as sparse matrix-vector multiplication with an unrestricted nonzeros pattern, and the Euler solver sweeping over unstructured meshes. We will keep improving our approaches and devise more general approaches.

Acknowledgment. The authors would like to thank Linghe Kong, Xiaoyang Liu, Sandy Harris and anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This research was partially supported by NSF of China under grant No. 61073158.

References

1. Hussein, M., Abd-Almageed, W.: Efficnent Band Approximation of Gram Matrices for Large Scale Kernel Methods on GPUs. In: Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–10. ACM Press, New York (2009)
2. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: 24th ACM International Conference on Supercomputing (ICS), pp. 115–126. ACM Press, New York (2010)
3. Cederman, D., Tsigas, P.: On Dynamic Load Balancing on Graphics Processors. In: 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 57–64. ACM Press, New York (2008)
4. Tzeng, S., Patney, A., Owens, J.D.: Task Management for Irregular-ParallelWorkloads on the GPU. In: High Performance Graphics 2010, pp. 29–37. ACM Press, New York (2010)
5. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: Proceedings of High Performance Graphics 2009, pp. 145–149. ACM Press, New York (2009)
6. Solomon, S., Thulasiraman, P.: Performance Study of Mapping Irregular Computations on GPUs. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1–8. IEEE Press, New York (2010)
7. Deng, Y., Wang, B.D., Mu, S.: Taming Irregular EDA Applications on GPUs. In: Proceedings of the 2009 International Conference on Computer-Aided Design, pp. 539–546. ACM Press, New York (2009)
8. Vuduc, R., Chandramowlishwaran, A., Choi, J.W., Guney, M.E., Shringarpure, A.: On the Limits of GPU Acceleration. In: Hot Topics in Parallelism (HotPar). USENIX Association, Berkeley (2010)
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. J. IEEE Micro. 28, 39–55 (2008)
10. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 407–420. IEEE Press, New York (2007)
11. Bell, J., Stevens, B.: A survey of known results and research areas for n-queens. J. Discrete Math. 309, 1–31 (2009)

12. Bozinovski, A., Bozinovski, S.: n-queenss pattern generation: an insight into space complexity of a backtracking algorithm. In: 2004 International Symposium on Information and Communication Technologies, pp. 281–286. Trinity College Dublin, Dublin (2004)
13. Khan, S., Bilal, M., Sharif, M., Sajid, M., Baig, R.: Solution of n-Queen Problem Using ACO. In: IEEE 13th International Multitopic Conference (INMIC), pp. 1–5. IEEE Press, New York (2009)
14. QUEESNTUD project, http://queens.inf.tu-dresden.de/
15. Shu, W., Wu, M.Y.: Asynchronous problems on SIMD parallel computers. J. IEEE Trans. on Parallel and Distributed Systems 6, 704–713 (1995)
16. Blas, A.D., Hughey, R.: Explicit SIMD Programming for Asynchronous Applications. In: IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 258–267. IEEE Press, New York (2000)
17. Cull, P., Pandey, R.: Isomorphism and the n-queenss problem. J. ACM SIGCSE Bulletin 26, 29–36 (1994)
18. NVIDIA CUDA C Programming Guide, http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf