NQueens on CUDA: Optimization Issues

Frank Feinbube Bernhard Rabe Martin von Löwis Andreas Polze
Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{*frank.feinbube, bernhard.rabe, martin.vonloewis, andreas.polze*}*@hpi.uni-potsdam.de*

***Abstract*—Todays commercial off-the-shelf computer systems are multicore computing systems as a combination of CPU, graphic processor (GPU) and custom devices. In comparison with CPU cores, graphic cards are capable to execute hundreds up to thousands compute units in parallel. To benefit from these GPU computing resources, applications have to be parallelized and adapted to the target architecture. In this paper we show our experience in applying the NQueens puzzle solution on GPUs using Nvidia's CUDA (Compute Unified Device Architecture) technology. Using the example of memory usage and memory access, we demonstrate that optimizations of CUDA programs may have contrary results on different CUDA architectures. Evaluation results will point out, that it is not sufficient to use new programming languages or compilers to achieve best results with emerging graphic card computing.**

***Keywords*-GPGPU; memory access trade-off;**

I. Introduction

A recent trend in microprocessor design is the increase of the number of processing cores on a single processor chip, where the resulting products are often called multicore or manycore processors. This trend originates from the desire to utilize the increased number of transistors which can be accommodated on a single chip, following the prediction of Moore's law. Other strategies for utilizing more transistors, such as pipelined and superscalar execution, have mostly been exhausted, leaving the integration of many computing cores as the major strategy to provide an ongoing increase of computing power. This trend can be seen both in the system central processors as well as in graphics processors.

For some years graphic cards were not only used to render pictures to screens, but also for numerical processing. In these applications, shader languages or vendor specific languages like *AMD Brook+* [1], *AMD Cal* [1] or *Nvidia Cg* [2] were applied. Current frameworks like *Nvidia CUDA* [3] and *AMD Stream Computing SDK* [4] are based on the *C* programming language with few extensions and have a general purpose nature. The next step will be the application of the emerging *OpenCL* [5], [6] programming framework. It allows to write programs that use either the CPU or GPU as the underlying processing device. The OpenCL implementations that were available at time of writing were not fully functional. ATI provided an implementation that was only capable to use CPUs while Nvidia only supported the use of their CUDA-enabled graphic cards. Therefore we focus on CUDA which is a well established "scalable parallel programming model and a software environment for parallel computing" [7].

The main contribution of this paper is a compilation of issues that we encountered when solving the NQueens problem using the CUDA framework. We first present the NQueens problem and the primary concepts of CUDA programming, report related work, and then report our own experiences with CUDA. The final evaluation shows that the performance impact of code changes can vary heavily for different CUDA architectures. This makes it very difficult to optimize the runtime of programs for CUDA-enabled graphic cards.

II. The NQueens Problem

In order to get a deep understanding of the CUDA programming model, we chose an active researched problem that is complex enough to demonstrate the abilities and limits of programming for GPU. The *n queens puzzle* fulfilled these requirements. Its goal is to find all ways to place a given number *n* of queens on a chessboard which has *n* times *n* fields. A configuration is only valid if no queen attacks another one. This holds if no queen is placed in a row, column or diagonal that is used by another queen.

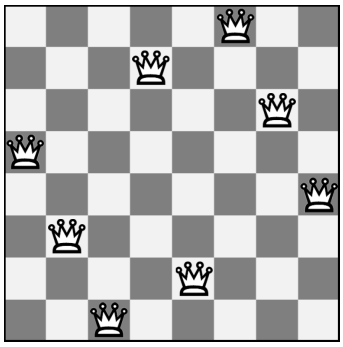

Figure 1. One of the 92 solutions for the 8 queens problem.

For a regular chess board of 8 by 8 fields, there are 92 possible configurations for 8 queens so that they do not attack each other. Because the problem state increases factorial with the number of queens, today the solutions for the puzzle are only known up to a number of 26 queens. Preußer et al. from the University of Dresden [8] calculated that there are 22,317,699,616,364,044 valid configurations and published their results on July, 11th 2009 [9].

III. Related Work

Queens@TUD (introduced in the previous section) utilized specialized FPGA (Field-programmable gate array) boards for the calculations. The overall problem was subdivided into sub-problems that were calculated in parallel. It took 270 days to calculate the solution for N=26. This is particular impressive if we take into account that only 26 Altera and Xilinx boards were used and that each of the boards was running at a very low clock frequency of 90 to 180 MHz. The great performance was achieved by minimizing the instruction set overhead. This example

also demonstrates that CPUs are not the only answer for high-performance parallel computing. Specialized hardware like FPGAs or graphic cards can be a better approach for some problem classes.

The NQueens@Home project headed at the Universidad de Concepción de Chile [10] is a massive-parallel distributed system with constantly varying participants. A middleware provides and distributes work packages over the Internet. The actual calculations are done by the personal computers of registered users. In contrast to similar systems, NQueens@Home does not exploit graphic cards for the calculations.

Both projects are using a heavily optimized sequential algorithm written in C-language by J. Somers [11] which we took as the basis for our algorithm as well. It is a backtracking algorithm and works as follows: For a new queen it uses a bit-mask to select a free place. After placing the queen it updates three lists of masks. One stores all occupied columns, the second all occupied positive[1] diagonals and the last one all occupied negative diagonals. For the next row, the positive diagonal is shifted to the right and the negative one to the left. After that the placing bit-mask is recalculated and a new queen will be set. If no place is available or all queens were set, the algorithm removes the latest entries and sets the latest queen to a different position. In case that all queens were set, the solution counter is increased by one.

In [12] the general purpose graphic programming language *Nvidia Cg* is used to solve the NQueens problem. The authors evaluated their solution with a Nvidia GeForce 6800 Ultra graphic card and compared the results with a Pentium M 2.00 GHz processor. The processor dramatically outperformed the graphic card.

[13] discusses different implementations of NQueens solvers and evaluates selected approaches. For that, a Nvidia GeForce 9600 GT running *CUDA 1.0* implementations and a Intel Quad Core 2.4 GHz running C++ implementations were used. This evaluation also shows that general purpose usage of graphic cards is slow. The authors also state that CUDA is even slower than *Cg*.

Both publications study only a single architecture of graphic cards. We will show that there are significant differences in the way graphic cards with different CUDA-Architectures run the same code.

IV. The CUDA-Programming Model

Nvidia CUDA [3] is a well established "scalable parallel programming model and a software environment for parallel computing" [7]. It allows writing programs that run on general purpose graphic processors (GPGPU) by Nvidia. Due to the hardware architecture of these devices, many complex computational problems can be solved much faster compared to current CPUs. These problems include physical computations and video processing. Nowadays, development for CUDA is done using some C extensions. The code is precompiled with the `nvcc` tool provided by Nvidia and eventually compiled to binaries accessing CUDA-enabled graphic drivers. Because CUDA works with all modern graphic cards from Nvidia, even Nvidia ION [14], it is widely-used in current computers. This makes it particularly interesting for research on parallel computing.

[1]Positive means facing from the upper left corner to the bottom right corner.

The CUDA programming model aims to enable programmers to develop parallel algorithms that scale to hundreds of cores using thousands of threads. CUDA developers should not need to think about the mechanics of a parallel programming language, but should be enabled to employ the CPU and the GPU in parallel [7].

Listing 1 shows the regular way to use CUDA. At first, memory is allocated on the host as well as on the device. Then the input parameters for the CUDA program are copied to the device. This is necessary because CUDA code cannot access host memory. The parts of the application that run on the graphic card are called *kernels*. A kernel is executed by a number of thread blocks that consist of a number of lightweight CUDA threads. The size of a thread block and the number of blocks can be configured at kernel launch time. The selection of these values is a non-trivial problem that is beyond the scope of this paper. CUDA threads that reside in the same block can communicate using slow device memory or fast shared memory. In addition they can be synchronized. The amount of shared memory per block can also be configured at kernel launch time. While the kernel is running, the CPU is free to process additional workloads. In this example it simply waits, till the calculations are finished using the `cudaThreadSynchronize` statement. The final step is to load the result data from the graphic card's memory to the host memory.

```
1  // 1. allocate memory on the graphic card
   cudaMalloc((void**)&dataGpu, memSize);
3  // 2. copy to device memory of the graphic card
   cudaMemcpy(pDataGpu, pData, memSize,
       cudaMemcpyHostToDevice);
5  // 3. run the calculation kernel
   kernel<<<blockSize, threadsPerBlock,
       sharedMemorySize>>>(pDataGpu);
7  // 4. wait for the graphic card threads to
       finish calculations
   cudaThreadSynchronize();
9  // 5. copy from device memory of the graphic
       card back to host memory
   cudaMemcpy(pData, pDataGpu, memSize,
       cudaMemcpyDeviceToHost);
```

Listing 1. CUDA usage model

Each kernel fulfills the same tasks, as listed in Listing 2. At first each thread calculates its unique thread identifier using some constructs provided by the CUDA environment (blockIdx, blockDim, threadIdx). This identifier can be used to make control decisions and to compute memory addresses to access input parameters. Then the actual calculations take place. In the last step of a CUDA kernel the calculated result is written back to the device memory which can be accessed by the host program afterwards.

```
   // 1. derive thread id from block id and
       relative thread id
2  int threadId = blockIdx.x*blockDim.x+threadIdx.x
       ;
   // 2. read from array (using thread id as index)
4  int parameter = dataGpu[threadId];
   // 3. calculate the result
6  result = parameter * parameter;
   // 4. write to array (using thread id as index)
8  dataGpu[threadId] = result;
```

Listing 2. CUDA kernel model

There is a strict separation of kernels and normal program routines. Kernel methods must not be recursive and must not use static

variables. On future CUDA-enabled architectures atomic double-precision floating point operations will be supported via hardware, but at the moment only single-precision floating point operations are feasible. Using double-precision floating point anyway results in slow performance due to the emulation with single-precision operations.

The CUDA memory model is shown in Table I. Similar to the memory of the host, every CUDA device has its own memory. This so called device memory or global memory is the biggest memory region on the graphic device. It is off-chip, therefore relatively slow and can be accessed by host and by kernels. In addition it is persistent across kernel launches. The other extreme are the very fast registers of a thread. There are 16k registers[2] for all threads on a device. Taken a kernel that is executed with 32 blocks and 128 threads per block, only 4 registers are available for each thread. Local variables that do not fit into registers reside in so called local memory. This memory is mapped into the off-chip device memory and therefore very slow compared to registers. In addition every thread block has a small amount (16 KB) of on-chip shared memory. That memory can be either used for communication and synchronization between threads or as a storage for local variables. Because it is so restricted, for a large thread count only a small amount of shared memory can be used per thread. In some cases it pays off to decrease the thread count and to use a greater amount of the fast shared memory for each thread. There are also some special cached read-only memories called texture and constant memory.

V. Approach

After investigating the problem and different proof of concepts, we have chosen the promising algorithm of J. Somers [11] (see Section III) as basis for our CUDA solution. This decision founded on the fact that this algorithm does not use recursions, consumes little memory and is used by several other implementations, especially in the very successful ones. [8], [10]

This NQueens algorithm is backtracking algorithm. The first step is to place a queen in the first row and remember occupied fields of the next row. The next queen is then placed in the second row, but cannot sit on in the same column or diagonal as the first queen. Now we derive the free columns in the third row and place a queen on a free place. This procedure is repeated until there is no free field left. If we filled all rows we have a solution, if not we can exclude a set of configurations. In both cases we back track to the previous row, place its queen on the next free field and go on with the procedure.

Due to the fact that flipping a board setting over the y axis creates a new unique solution, it is sufficient to calculate only one-half of the solutions. For odd board sizes a special handling of the middle column is needed, because only one queen can sit in a vertical row.

A. Parallelization

The first and most important step in order to use CUDA for the calculations was to parallelize Somers algorithm. Therefore it was modified in a way that allows the precalculation of all board settings for a given number of rows. Figures 2 shows such a precalculated board setting for two queens. These boards are used as the input for the algorithm which calculates all solutions starting from the given setting. Applied to CUDA the first algorithm runs on the host and the second one as a kernel on the graphic card.

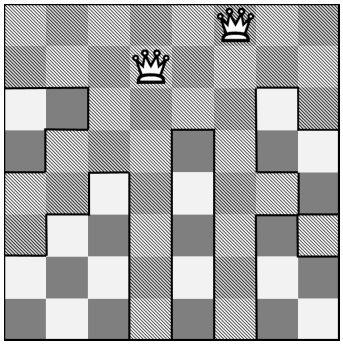

Figure 2. A precalculated board setting as a startup configuration for our parallel backtracking algorithm. The shaded area is occupied by the queens that are already placed on the board. The rest of the board is the solution space where the remaining 6 queens have to be placed by the worker thread.

The data layout of a precalculated board setting is shown in Figure 3. The data is represented as a single-dimensional array. Every thread has a chunk within this array that contains information about the precalculated rows. If we use 4 threads and have 3 precalculated rows per thread the layout would look exactly like in the figure. Every row setting is represented by a bitfield where the free places are marked by 1s and occupied ones by 0s.

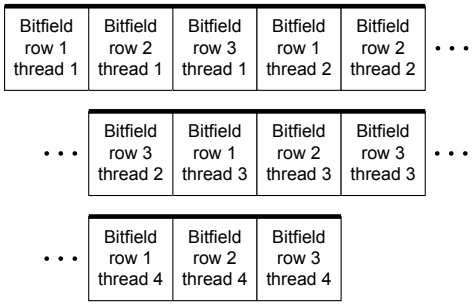

Figure 3. Data layout of precalculated board settings. In this example we have 4 threads and precalculated 3 rows.

Listing 3 shows the host routine that calls the CUDA kernel. In the `initData` function we calculate the initial board settings using a given board size and a row count[3]. Thereby we also determine the thread count, which is equal to the number of precalculated board settings. Next we allocate memory on the graphics card and fill it with our data. After that we start the kernel, wait for it to finish and finally sum up the solutions of the parallel kernel runs. Each thread stores its result at the first position of its input data array chunk.

```
int threadCount, mem_size = 0;
int* data_gpu;

int* data = initData(boardsize, depth, &
    threadCount, &mem_size);
int blockSize = threadCount/threadsPerBlock+1;
```

[2]Cards from the GeForce 8 and 9 series only had 8k of registers.

[3]Which is called *depth* in the listing.

memory type	scope	host access	kernel access	speed	size
host memory	per host	R/W	None	medium	big
device memory	per device	R/W	R/W	medium	big
registers	per thread	None	R/W	very fast	very small
local memory	per thread	None	R/W	medium	medium
shared memory	per block	None	R/W	fast	small
constant memory	per device	R/W	R	fast (cached)	medium
texture memory	per device	R/W	R	fast (cached)	medium

Table I
CUDA MEMORY MODEL [3]

```
6
  cudaMalloc((void**)&data_gpu, mem_size);
8 cudaMemcpy(data_gpu, data, mem_size,
      cudaMemcpyHostToDevice);

10 NqueensCUDA<<<blockSize,threadsPerBlock>>>(
      boardsize, threadCount, depth, data_gpu);
  cudaThreadSynchronize();
12
  cudaMemcpy(data, data_gpu, mem_size,
      cudaMemcpyDeviceToHost);
14
  cudaThreadExit();
16
  unsigned long long solutionCount = 0;
18 for(threadId=0;threadId<threadCount;threadId++)
    solutionCount += data[threadId * depth];
```

Listing 3. Host side of the CUDA NQueens implementation

Listing 4 shows the complete CUDA kernel for our first parallel adaption of the NQueens program. A comprehensive description of the algorithm can be found at [11]. Thus we will focus our descriptions on the code changes we made.

The algorithm consists of two main parts. The first one is the loop that initializes the fields using the input data. The second one is the critical loop that uses back tracking to calculate the solutions, beginning at the starting point from the initialization.

First we derive the thread id as described in section IV. We use this id to get the input data for the thread and fill the structures in the initialization loop: `lsb = data[threadId * depth + numrows];`

In the critical loop the main difference is the determination of the end of the stack using the row count: `if (nStack == depth)` Thus the back tracking algorithm will stop, when the sub-problem is solved.

```
1 __global__ void NqueensCUDA(int board_size, int
      threadCount, int depth, int * data)
  {
3   int threadId = blockIdx.x*blockDim.x+threadIdx.x
        ;

5   if(threadId >= threadCount || threadId < 0)
      return;
7
    ULL solutionCount = 0;
9
    /* mark columns and diagonals that are set */
11  int qBitCol[MAX_BOARDSIZE];
    int qBitPosDiag[MAX_BOARDSIZE];
13  int qBitNegDiag[MAX_BOARDSIZE];

15  /* we use a stack instead of recursion */
    int aStack[MAX_BOARDSIZE + 2];
17  register int nStack;

19  register int numrows = 0;
    register unsigned int lsb; /* least sig. bit */
21
     /* bits which are set mark free positions */
23  register unsigned int bitfield = 0;
    int board_minus = board_size - 1;
25
    /* if board size is N, mask consists of N 1's */
27  register int mask = (1 << board_size) - 1;

29  /* Initialize stack */
    aStack[0] = -1; /* signifies end of stack */
31
    int half = board_size>>1; /* divide by two */
33  /* fill in rightmost 1's in bitfield for half of
        board_size. */
    bitfield = (1 << half) - 1;
35  nStack = 1; /* stack pointer */

37  qBitCol[0] = qBitPosDiag[0] = qBitNegDiag[0] =
        0;

39  /* initialize field (precalculated setting) */
    for(;numrows < depth;)
41  {
      lsb = data[threadId * depth + numrows];
43    bitfield &= ~lsb;

45    int n = numrows++;

47    /* mark occupied places in next row */
      qBitCol[numrows] = qBitCol[n] | lsb;
49    qBitNegDiag[numrows]= (qBitNegDiag[n] | lsb)
          >>1;
      qBitPosDiag[numrows]= (qBitPosDiag[n] | lsb)
          <<1;
51
      aStack[nStack++] = bitfield;
53    /* We can't consider positions of queens that
          already on the board. */
      bitfield= mask & ~(qBitCol[numrows]|
          qBitNegDiag[numrows]|qBitPosDiag[numrows])
          ;
55  }

57  /* this is the critical loop */
    for (;;)
59  {
      /* get first (least sig) "1" bit */
61    lsb = -((signed)bitfield) & bitfield;
      if (0 == bitfield)
63    {
        /* get prev. bitfield from stack */
65      bitfield = aStack[--nStack];
```

```
      if (nStack == depth) { /* if end of stack */
67      break ;
      }
69    --numrows;
      continue;
71  }
    bitfield &= ~lsb; /* toggle off this bit */
73
    if (numrows < board_minus) /* more rows? */
75  {
      int n = numrows++;
77
      /* mark occupied places in next row */
79    qBitCol[numrows] = qBitCol[n] | lsb;
      qBitNegDiag[numrows]=(qBitNegDiag[n]|lsb)
          >>1;
81    qBitPosDiag[numrows]=(qBitPosDiag[n]|lsb)
          <<1;

83    aStack[nStack++] = bitfield;
      /* We can't consider positions of queens
          that already on the board. */
85    bitfield=mask&~(qBitCol[numrows]|qBitNegDiag
          [numrows]|qBitPosDiag[numrows]);
      continue;
87  }
    else
89  {
      /* No more rows to process; a solution. */
91    ++solutionCount;
      bitfield = aStack[--nStack];
93    --numrows;
      continue;
95  }
}
97
/* multiply solutions by two (mirror images) */
99 data[threadId * depth] =  solutionCount * 2;
}
```

Listing 4. CUDA kernel of our adapted N Queens program based on J. Somers' work

With this program we achieved occupancy of 100%, but utilized the slow device memory of our graphic card.

B. Optimizations

After the basic program was functional, we modified it successive to benefit from hardware features e.g. use shared memory instead of mapped device memory for the arrays. Because shared memory is fairly small, making use of it reduces the number of threads that can be deployed. But in exchange using fast shared memory brings huge performance benefits over slow device memory. The performance impact of these optimizations is discussed in the next section. This section will describe the optimizations in detail.

The first optimization step was the usage of shared memory for the stack. Therefore we used the `__shared__` statement of CUDA. Listing 5 shows the changes that were necessary to access the shared stack.

```
4  int tx = threadIdx.x * MAX_BOARDSIZE;
5  ...
16 __shared__ int aStack[(MAX_BOARDSIZE+2)*
       THREADS_PER_BLOCK];
17 register int nStack = tx;
```

Listing 5. Optimization Step 1

After that we also modified the arrays that store the columns and diagonals to use shared instead of device memory. This is shown in Listing 6.

```
11 __shared__ int qBitCol[MAX_BOARDSIZE*
       THREADS_PER_BLOCK];
12 __shared__ int qBitPosDiag[MAX_BOARDSIZE*
       THREADS_PER_BLOCK];
13 __shared__ int qBitNegDiag[MAX_BOARDSIZE*
       THREADS_PER_BLOCK];
14
15 ...
36
37 qBitCol[tx]=qBitPosDiag[tx]=qBitNegDiag[tx]=0;
```

Listing 6. Optimization Step 2

The usage of normal shared memory as shown in both optimizations requires choosing the memory size at compile time. To adjust the shared memory sizes at runtime the `extern __shared__` prefix can be used. The memory size is determined by the optional third parameter of the CUDA kernel call (see Listing 3), as shown in Listing 7.

```
9  int sharedMemorySize = MAX_BOARDSIZE * 4 *
       threadsPerBlock * sizeof(int);
10 NqueensCUDA<<<blockSize, threadsPerBlock,
       sharedMemorySize>>>(boardsize, threadCount,
       depth, data_gpu);
```

Listing 7. Optimization Step 3 - Call of the CUDA Kernel

Because the extern shared memory is one big block, we need different indices to map the four arrays per thread to it. These are shown in Listing 8.

```
2  extern __shared__ int sharedData[];
3
4  ...
10
11 int iStack = MAX_BOARDSIZE * 4 * tx + 0 *
       MAX_BOARDSIZE;
12 int iQBitCol = MAX_BOARDSIZE * 4 * tx  + 1 *
       MAX_BOARDSIZE;
13 int iQBitPosDiag = MAX_BOARDSIZE * 4 * tx + 2 *
       MAX_BOARDSIZE;
14 int iQBitNegDiag = MAX_BOARDSIZE * 4 * tx + 3 *
       MAX_BOARDSIZE;
15
16 register int nStack = iStack;
17
18 ...
36
37 sharedData[iQBitCol] = sharedData[iQBitPosDiag]
       = sharedData[iQBitNegDiag] = 0;
```

Listing 8. Optimization Step 3

The final step was to reduce the necessary amount of shared memory by using current instead of maximal board size to seize the arrays. Less usage of shared memory increases the maximum amount of threads that can be used per block. With this final optimization we hit all three limits for thread blocks per multiprocessor for graphic cards with a CUDA compute capability of 1.1 (registers, shared memory and max warps). We can run eight thread blocks per multi-processor and thus achieve a maximal occupancy of 33%.

```
10 int abs = board_size - depth;
11 int iQBitCol = abs * 4 * tx + 0 * abs;
```

```
12  int iQBitPosDiag = abs * 4 * tx + 1 * abs;
13  int iQBitNegDiag = abs * 4 * tx + 2 * abs;
14  int iStack = abs * 4 * tx + 3 * abs;
15
16  ...
39
40  for(int d=0; d<depth; d++) /* initialize field
        */
```

Listing 9. Optimization Step 4

The next section describes the impact this optimization had on the performance of different CUDA-enabled graphic cards.

VI. Evaluation

CUDA provides a comprehensive set of profiling capabilities. For a profiled program, GPU and CPU times, occupancy, grid sizes, register count and shared memory usage are captured for each kernel. In addition four out of eleven values can be chosen to be gathered. These values comprise the number of memory stores and loads, branches, instructions and launched thread blocks. For the memory accesses it also distinguishes between coherent and incoherent ones. This distinction is useful, because incoherent memory accesses are a performance bottleneck on the old graphic card architectures. These values can only be profiled for one multiprocessor (core). But even with this restriction the toolkit is powerful enough to provide a solid basis for effective optimizations.

A. Test systems

The evaluation of the different versions of our n queens program were done on the test systems listed in Table II. Test system 1 employs a graphic card from Nvidia's latest architecture (GeForce 200 Series), whereas the graphic card on test system 2 is built using the oldest CUDA-enabled architecture (GeForce 8 series). The GeForce GTX 275 has 240 multi-cores, a clock speed of 1404 MHz and a CUDA compute capability of 1.3. In contrast the GeForce 8600M GS only has 16 multi-core processors, a clock speed of 1200 MHz and a CUDA compute capability of 1.1. Test system 3 was tested with two graphic cards. First (3.1) a GeForce GTX 295 which consists of two GPUs with 240 multi-cores each, a clock speed of 1242 MHz and a CUDA compute capability of 1.3. The second configuration (3.2) used a Quadro NVS 295 that has 8 multi-cores, a clock speed of 1300 MHz and a CUDA compute capability of 1.1.

One of the main differences between CUDA compute capability 1.1 and 1.3 is the performance improvement for coalesced memory access in the latter version.

B. Performance impact of the optimizations

After our NQueens program was running on CUDA-enabled graphic devices we began to optimize it. Therefore we reduced the amount of memory needed by each thread. In addition we made use of shared memory instead of local memory to minimize the overhead of device memory accesses. Using more shared memory on the other hand reduces the number of threads that can be run in parallel on a multiprocessor, because only 16KB of shared memory are available per block.

In order to see the performance impacts of our optimizations we executed the following procedure: Each program version was executed using different configurations. These configurations varied in complexity of the sub-boards (precalculated row count, *depth*) and the thread count per block. The complexity resulted in different thread counts: the more sub-boards were precalculated, the more threads could be used and the lighter was the workload per thread. The block count can be calculated by dividing the thread count by the thread count per block. Table III shows the range of the input parameters.

Component	Description
test system 1	
CPU	Intel Core 2 Duo E8500, 2x3.17 GHz
Chipset	Intel P45
Graphics card	Gainward GeForce GTX 275
RAM	4GB (DDR2-1000)
test system 2	
CPU	Intel Core 2 Duo Mobile T9300 2x2.5 GHz
Chipset	Intel PM965
Graphics card	Nvidia GeForce 8600M GS
RAM	3 GB (DDR2-800)
test system 3.1	
CPU	Two Intel Xeon E5520 4x2.26 GHz
Chipset	Intel 5520
Graphics card	Nvidia GeForce GTX 295
RAM	6 GB (DDR3-1066)
test system 3.2	
CPU	Two Intel Xeon E5520 4x2.26 GHz
Chipset	Intel 5520
Graphics card	Nvidia Quadro NVS 295
RAM	6 GB (DDR3-1066)

Table II
Testing Environment

precalculated row count	2 3 4 5 6
thread count per block	1 2 3 4 8 16 32 64 128 180 256 448 512

Table III
Test Configuration: variing parameters

The configuration with the shortest run time was chosen as a reference. The runtime varied from some seconds to several minutes depending on the current configuration. The occupancy varied from 15.6% to 100% and the best result did often not have the highest occupancy. The runtime is calculated within the program and includes the precalculation of the board settings. The overhead for these precalculations is insignificant for huge queen counts. In addition as described in the previous chapter, at test system 3.1 and 3.2 only the GPU was exchanged. Thus their performance could be compared directly.

Figure 4 shows the results for test system 1 and test system 3.1 where current CUDA-enabled graphic cards are employed. With the first program version it took 2.9 / 3.3 seconds to calculate the number of valid configurations for a *16* queens problem. Optimizations reduced execution time, so that in the end the program only used 1.1 / 1.6 seconds to calculate the same result. As expected, using the shared instead of mapped device memory increases performance. The only exception is optimization 3, where the performance slightly declined.

The results of test system 2 and test system 3.2 are shown in Figure 5. In these systems two of the first CUDA-enabled graphic cards were used. The unoptimized NQueens program needed 14.7 / 16.4 seconds to solve the 16 queens problem in the best con-

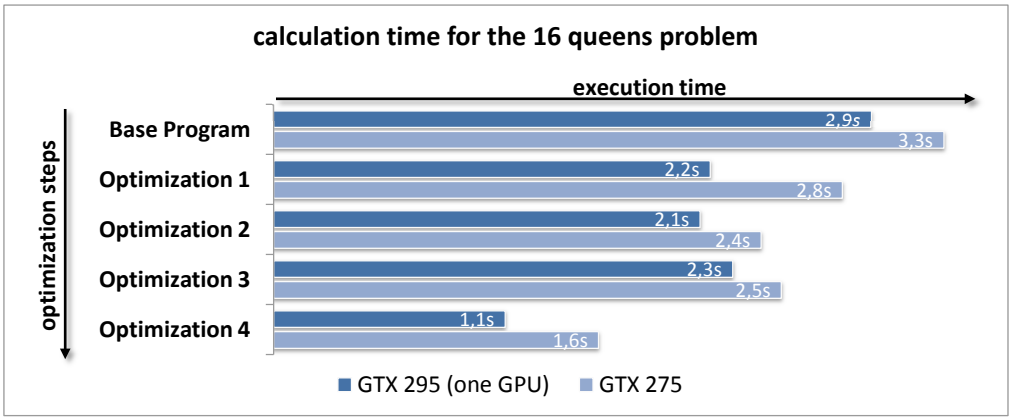

Figure 4. Runtime comparison of different versions of the NQueens program for N=16 on GeForce GTX 275 (test system 1) and GeForce GTX 295 (test system 3.1, only one GPU used). The base program and optimization details are described in section V.

figuration. In contrast to test system 1, the first three optimization steps increased the execution time. Thus NQueens Optimization 3 required 25.2 / 38.2 seconds to calculate the result. After that, with the final optimization, an increase of the performance occurs. The profit is so big that the execution time of the final version is 12.0 / 17.2 seconds and thus better than the first one.

There is a significant gap between the impacts of the optimizations for both architectures. While they provided a better performance on the later architecture, each step led to worse performance on the predecessor architecture. This holds until the last step, where a big performance gain showed up. This behavior also occurred for different queen counts (14, 15, 17). Because the problem size and runtime grow factorial, for sizes below 14 runtime measurements become inappropriate and for sizes above 17 the execution time for all configuration exceeds days.

This evaluation shows that programming for general purpose graphic cards is still problematic, because programmers can not even make assumptions on the effects that optimizations of a program will have. A program that is optimized for a chosen architecture will not necessarily run fast on another CUDA-enabled architecture. Regarding test system 2 and test system 3.2, a programmer would have stopped his optimizations towards the usage of shared memory, because this led to worse performance. So the optimal performance of 12.0 / 17.2 seconds would not be achieved.

Recommended optimizations regarding shared memory are: the application of fast shared memory instead of slow device memory and the reduction of the amount of memory that is needed per thread. Higher memory access performance is paid with reduced occupancy of graphic card resources. That means that fewer threads can be executed in parallel. The CUDA execution model assumes that most threads in a hardware execution unit (warp) are waiting for I/O (memory access) and are not ready for execution while one thread is running. Thus a reduced memory access delay and a smaller number of threads may improve the performance anyway. For CUDA graphic cards with a compute capability of 1.1 or 1.0, like the ones we used in test system 2 and test system 3.2, the order in which memory is accessed is also relevant. Successive accesses on memory addresses are very fast because all the accesses can be coalesced. If there is no such order, in the worst case each access will result in an individual I/O operation with the proper delay. These additional costs do not accumulate on newer CUDA-enabled architectures. This points out that not only memory type, but also memory access patterns have to be considered. A detailed analysis of memory access patterns and memory coalescing optimization for test system 2 and test system 3.2 are beyond the scope of the paper. Further information on optimizing for CUDA can be found in [15] and [16].

VII. Conclusion

In the literature, current programming frameworks for graphic cards are claimed to be easy to use and very fast.

We demonstrated that the performance of a GPU program is very hard to predict due to different hardware capabilities of different graphic card series. Thus optimizing the code requires a deep understanding of the target platform. Looking at the code, it is not clear whether accessing a variable will be fast, because it is kept in a register, or slow, because it resides in off-chip local memory. This is a situation that programmers acquainted to CPUs are very unfamiliar with. But since future CPU architectures will have some similarities with current GPU architectures - for example the distribution of memory, regular programmers might face similar problems in the future. Therefore studying the characteristics of modern GPUs might help us to develop better programming frameworks for future CPUs.

To build efficient programs for the Compute Unified Device Architecture (CUDA), a sophisticated knowledge about the underlying hardware architecture is needed. Thus, it is not as easy to use the CUDA framework as it seems in the first place. To make it worse CUDA comes with a lot of restrictions. Because light-weight CUDA-threads have no stack, recursion is not supported. CUDA routines are differentiated from normal program routines so reusing of code fragments becomes impossible. On Windows, long-running CUDA-threads result in a driver restart [17].

There is still a lot of work to do until general purpose programming for graphic cards will be as comfortable as programming for CPUs. Performance still has its price. The trade-off for faster programs is a higher programming effort.

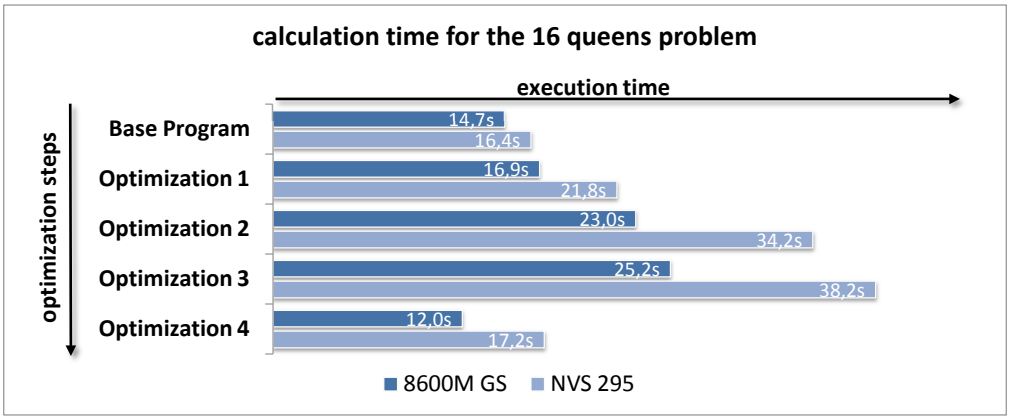

Figure 5. Runtime comparison of different versions of the NQueens program for N=16 on GeForce 8600M GS (test system 2) and Quadro NVS 295 (test system 3.2). The base program and optimization details are described in section V.

REFERENCES

[1] Advanced Micro Devices. (2009) ATI Stream Software Development Kit (SDK) v1.4beta. Advanced Micro Devices, Inc.

[2] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. ISBN 978-0321194961: Addison-Wesley Professional, March 2003.

[3] Nvidia, *Nvidia CUDA - Programming Guide - Version 2.3.1*, 2nd ed., Nvidia Corporation, August 2009.

[4] Advanced Micro Devices. (2009) ATI Stream Software Development Kit (SDK) v2.0beta. Advanced Micro Devices, Inc.

[5] Khronos Group, *The OpenCL Specification*, Khronos Group Std. 1.0, Rev. 48, October 2009.

[6] Apple. (2009) Apple - Mac OS X - New technologies in Snow Leopard. Apple Inc.

[7] G. Ruetsch and B. Oster. (2008) Getting Started with CUDA (Tutorial). NVIDIA Corporation.

[8] T. B. Preußer, B. Nägel, and R. G. Spallek, "Putting Queens in Carry Chains," Department of Computer Science, Technische Universität Dresden, Germany, Tech. Rep. TUD-FI09-03, March 2009.

[9] ——. (2009) Queens@TUD. Technische Universität Dresden. [Online]. Available: http://queens.inf.tu-dresden.de/

[10] I. Figueroa. (2009) NQueens@Home. Universidad de Concepción. [Online]. Available: http://nqueens.ing.udec.cl/

[11] J. Somers. (2002) The N Queens Problem - a study in optimization. [Online]. Available: http://jsomers.com/nqueen_demo/nqueens.html

[12] D. Amrasinghe. (2007) N-queens problem with GPGPU (Presentation). University of North Texas.

[13] V. Pamplona. (2008) n-Queens Problem: A Comparison Between CPU and GPU using C++ and Cuda (Presentation). Universidade Federal do Rio Grande do Sul.

[14] Nvidia. (2009) Nvidia CUDA-enabled products. Nvidia Corporation. [Online]. Available: http://www.nvidia.com/object/cuda_learn_products.html

[15] P. Micikevicius. (2008, November) Optimizing CUDA (Tutorial). NVIDIA Corporation.

[16] B. Oster. (2008) Advanced CUDA (Tutorial). NVIDIA Corporation.

[17] Microsoft. (2009, April) Timeout Detection and Recovery of GPUs through WDDM (Windows Hardware Developer Central). Microsoft.