Abstract

The Boolean domain faces us with the exponential complexity of Boolean functions and the technological progress in micro- and nano-electronics allows increasing numbers of Boolean variables. This requires very powerful Boolean computations. The progress in the performance of Graphics Processing Units (GPUs) and the possibility to utilize the GPU to solve tasks of many application domains establish a new powerful hardware basis. In this paper, we utilize the GPU for Boolean computations.

The suggested XBOOLE-CUDA allows the user to speedup required time-consuming Boolean calculations without facing the huge amount of details to implement an efficient GPU program. Based on the Bishops Problem on a chess board we generate Boolean problems in a wide range of size and explore the benefit of the GPU calculating intersection operation of XBOOLE .

1 Introduction

The technological progress in micro- and nano-electronics leads to both many helpful new applications and growing requirements for the design of digital systems [6]. Boolean functions are the main instrument for their description [4]. It is well known that the number of function values of a Boolean function exponentially growths depending on the number of Boolean variables. Hence, solving of Boolean problems which depend on many variables is an important challenge for scientists [6].

An important source for improvements to solve Boolean tasks is the utilization of many compute cores for parallel computations. Today’s processors contain a small number of cores in the Central Processing Unit (CPU). Significantly more cores are available on the Graphics Processing Unit (GPU). Therefore, the main aim of this paper is the utilization of the GPU for faster Boolean computations.

The needed resources in time and space to solve a Boolean problem are not only determined by the used hardware but also by the main data structure of the needed Boolean functions and the algorithms for their computation. One result of comprehensive researches in this field is XBOOLE [7]. Many applications of XBOOLE are shown in [9, 12]. XBOOLE was originally implemented for a single core of the CPU. Alternative methods for faster calculations of Boolean problems were explored in [14]. Werner has ported the most time consuming XBOOLE-operations to the GPU in his recent Master thesis [16]. He used Nvidia’s Compute Unified Device Architecture (CUDA) developing the new library called XBOOLE-CUDA.
The rest of the paper is organized as follows. Section 2 gives a short introduction into the concept and operations of XBOOLE. Based on that we report in Section 3 details about the implementation of XBOOLE-CUDA and the achieved benchmark results. In Section 4 we explore the practical application of XBOOLE-CUDA using several variants to solve the complex Bishop Problem on a chessboard of the size $11 \times 10$.

2 XBOOLE

XBOOLE [7] is a library of more than 100 functions which can be used within programs, written in the programming languages C or C++, to solve a wide field of Boolean problems [4, 11, 12]. In order to make this chapter self-contained, we give a very brief introduction to XBOOLE.

Two binary vectors which differ in a single position can be merged into an single ternary vector with a dash element ($-$) in this position and unchanged values in all other positions. Similarly, two ternary vectors with the same property can be also merged into a single ternary vector. A ternary vector with $d$ dash elements represents $2^d$ binary vectors. In this way XBOOLE exponentially reduces the needed memory to store sets of binary vectors.

XBOOLE uses Ternary Vector Lists (TVLs) to compress sets of binary vectors. Boolean operations are executed by XBOOLE directly on TVLs where all elements of a machine word are calculated in parallel. An orthogonal TVL does not contain any binary vector in more than one ternary vector. Most of the XBOOLE operations compute an orthogonal TVL.

The number of Boolean variables is not limited in XBOOLE. The user of XBOOLE can define as many as needed Boolean spaces. However, he must fix the number of Boolean variables for each of these spaces.

Each XBOOLE-operation is associated to one group of operations. These groups give us a quick overview about the whole XBOOLE system:

1. set operations,
2. derivative operations of the Boolean Differential Calculus,
3. conversion operations,
4. operations to manipulate ternary matrices,
5. test operations,
6. operations to manipulate sets of Boolean variables,
7. operations that provides several predicates,
8. management of objects,
9. operations to load and store data of an XBOOLE-systems,
10. management of the XBOOLE-memory, and
11. error handling.

A TVL can be considered as a set of binary vectors. The set operations:

- CPL the Complement $\overline{A}$,
- ISC the Intersection $A \cap B$,
- UNI the Union $A \cup B$,
- DIF the Difference $A \setminus B$,
- SYD the Symmetrical Difference $A \Delta B$, and
- CSD the Complement of the Symmetrical Difference $\overline{A \Delta B}$

can be used to compute needed new sets of binary vectors. Using a form attribute a TVL is able to represent a Boolean function in each of the four basic forms [8]:

- D Disjunctive Form,
- K Conjunctive Form,
- A Antivalence-form, or
- E Equivalence-form.

A benefit of an orthogonal TVL is that such a TVL in ODA form can be used in both D- or A-form. Dual properties are valid for the OKE-form.
Boolean operations between functions are directly realized by the introduced set operations. In case of an ODA-form we have the following association:

- CPL negation ($f$),
- UNI disjunction ($f \lor g$),
- ISC conjunction ($f \land g$),
- DIF difference ($f \setminus g$),
- SYD antivalence ($f \oplus g$), and
- CSD equivalence ($f \circ g$).

The Boolean Differential Calculus [4, 10, 12] extends the Boolean Algebra by operations which evaluate certain changes of Boolean values. XBOOLE directly provides all $k$-fold derivative operations and all vectorial derivative operations:

- DERK the $k$-fold derivative,
- DERV the vectorial derivative,
- MINK the $k$-fold minimum,
- MINV the vectorial minimum,
- MAXK the $k$-fold maximum; MAXV the vectorial maximum.

Both the set operations and the derivative operations are the most important and most time-consuming XBOOLE operations. Hence, the utilization of the GPU for these operations has the strongest potential to speedup XBOOLE applications. Due to the restricted space of this paper we refer to [4, 11, 12] for further details about XBOOLE.

3 XBOOLE-CUDA

GPUs were basically developed to accelerate the graphical representation of data on the screen. Hence, GPUs naturally have to transform geometry- and texture-data to colored pixels. GPUs have been heavily optimized for such tasks over the years. This has been reached by throughput oriented, many-core architectures with hundreds of compute cores. The main advantage of the GPU is the much higher number of cores in comparison with the CPU. Therefore, it is an interesting challenge to port a very computation intensive task from the CPU to the GPU. Werner solved this task in [16] for main XBOOLE operations.

The porting of an efficient, sequential CPU-program like XBOOLE into an also efficient, parallel GPU-program like XBOOLE-CUDA is no simple task. Many skills are needed due to the differences in the architecture and programming paradigm between the CPU and GPU and the success strongly depends on the algorithm to port. Due to Amdahl's law [1], the theoretical maximal speedup is determined by the sequential part of a program even when an infinite number of processor cores can be utilized. Consequently, significant speedups can be reached on the GPU only for programs with a large parallel part.

A sequential program runs best on a single CPU core which is skillful and fast on executing Single Instruction, Single Data, (SISD) [2]. In opposite, the GPU requires parallel and concurrent programming. The programmer has to manage many, but slow cores such that they do not obstruct each other, even though they have to share scarce resources. These different paradigms imply much more challenges. The GPU programmer is responsible for optimal work balance, cache coherency and optimal utilization of memory hierarchies—e.g., coalesced memory access is crucial on a GPU.

A GPU consists of several, concurrently acting Multiprocessors (MPs). Each of these MPs is equipped with shared memory, very fast thread-private registers, load and store units and many compute cores, which work in parallel. On Nvidia GPUs all computations are realized by lightweight threads, where 32 threads are grouped in a warp. Instructions are executed by warps based on the Single Instruction, Multiple Data principle, (SIMD) [2]. Generally, 32 threads of a warp execute the same instruction at the same time on different data. Warp instructions are serialized by the hardware, if divergent control flows, atomic operations or memory conflicts are present. Such latencies can be hidden by executing other warp instructions, as long as there are free warps available for the warp schedulers (see latency hiding and occupancy).

The GPU-programmer is furthermore faced with special requirements regarding the memory. Communication between CPU and GPU requires data transactions between main memory and device memory over the PCIe bus. This data transfer is limited by the bandwidth of PCIe bus of e.g., 8 GB/s for PCIe v2.x. The GPU itself has internal caches and memory layers designed by decreasing both the size and the latency.
Valuable hints regarding optimal algorithms and programs on Nvidia GPUs are given in [17]. Details of the used Nvidia Fermi-GPU architecture are described in [3] and information about memory and instruction latencies of these GPUs are given in [15].

Werner implemented the library XBOOLE-CUDA which migrates XBOOLE and also utilizes multi-GPU platforms [16], where the GPU code is optimized for Fermi-GPUs. A wrapper decides by the size of the given TVLs whether CPU is used for small TVLs or the GPU accelerates the calculation for large TVLs. Additional operations allow the programmer to customize such decisions. Implementations for the CPU and GPU are realized in XBOOLE-CUDA for all set operations, all derivative operations, all test operations, and the ORTH operation. In this way, all recent and future applications benefit from the speedup of XBOOLE-CUDA and the simple implementation of Boolean algorithms on the high domain specific level. The migration from an XBOOLE program to an XBOOLE-CUDA program simply requires the replacement of the library.

### Algorithm 1 Intersection (ISC)

<table>
<thead>
<tr>
<th>Input:</th>
<th>TVL ( T_1 ), TVL ( T_2 ), both in ODA-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>TVL ( R = T_1 \cap T_2 ), both in ODA-form</td>
</tr>
</tbody>
</table>

\[
egin{align*}
1: & \quad r \leftarrow 0 & \text{initial value of result vectors} \\
2: & \quad \text{for } i = 1, \ldots, n_1 \text{ do } & \text{executed in parallel if } \text{NTV}(T_1) \geq \text{NTV}(T_2) \\
3: & \quad \text{for } j = 1, \ldots, n_2 \text{ do } & \text{executed in parallel if } \text{NTV}(T_1) < \text{NTV}(T_2) \\
4: & \quad c \leftarrow \omega \\
5: & \quad \text{while } ([a[T_1, i, c] \oplus a[T_2, j, c]] \land b[T_1, i, c] \land b[T_2, j, c]) = 0 \text{ do } & T_1(i, c) \land T_2(j, c) ? \\
6: & \quad c \leftarrow c - 1 \\
7: & \quad \text{if } c = 0 \text{ then } & \text{the ternary vectors are not orthogonal to each other} \\
8: & \quad \text{break} \\
9: & \quad \text{end if} \\
10: & \quad \text{end while} \\
11: & \quad \text{if } c = 0 \text{ then } & \text{in case of non-orthogonal ternary vectors} \\
12: & \quad r \leftarrow r + 1 & \text{index of the new result vector} \\
13: & \quad \text{for } c = 1, \ldots, \omega \text{ do } & \text{calculate and store intersection vector in } R \\
14: & \quad a[R, r, c] \leftarrow a[T_1, i, c] \lor a[T_2, j, c] \\
15: & \quad b[R, r, c] \leftarrow b[T_1, i, c] \lor b[T_2, j, c] \\
16: & \quad \text{end for} \\
17: & \quad \text{end if} \\
18: & \quad \text{end for} \\
19: & \quad \text{end for} \\
\end{align*}
\]

The intersection is one of the easier operations to port; nevertheless, there are pitfalls that can decrease the speedup significantly. As an example, Algorithm 1 gives a better insight.

One ternary vector consists of an array of pairs of machine words, since two binary bits are needed to encode one ternary bit. If there are 128 variables, \( \omega \) can decrease the speedup significantly. As an example, Algorithm 1 gives a better insight. Implementations for the CPU and GPU are realized in XBOOLE-CUDA for all set operations, all derivative operations, all test operations, and the ORTH operation. In this way, all recent and future applications benefit from the speedup of XBOOLE-CUDA and the simple implementation of Boolean algorithms on the high domain specific level. The migration from an XBOOLE program to an XBOOLE-CUDA program simply requires the replacement of the library.

The intersection is one of the easier operations to port; nevertheless, there are pitfalls that can decrease the speedup significantly. As an example, Algorithm 1 gives a better insight.

One ternary vector consists of an array of pairs of machine words, since two binary bits are needed to encode one ternary bit. If there are 128 variables, \( \omega = \lceil \frac{128}{32} \rceil = 4 \) (a, b) words or \( 2 \ast \omega = 8 \) machine words are needed for a single ternary vector on a 32 bit system.

For the parallelization the “bigger” for-loop in lines 2 and 3 is chosen, otherwise there would be fewer threads having more sequential work, which yields longer runtime. At first glance, the test for non-orthogonality in line 5 seems to be independent from other threads. However, this while-loop can be skipped early by some threads of a warp. This violates the SIMD pattern, i.e., threads of a warp have to wait, until the last one has finished the test.

Making things even worse, further instructions in lines 12 to 16 are executed only for these threads, which passed the test. Those, who failed the test, have to wait again. If the latency is very high, e.g., in case of the DIF-operation, sorting of TVLs can enable better speedups. This will be a task for our future work. Algorithm 1 requires in line 12 a single atomic operation on the GPU. Hence, if there are many output vectors to be computed then the GPU badly performs due to the serialization by atomic operations.

Another big pitfall is the memory alignment of the (a, b) words on the GPU. In XBOOLE (CPU) (a, b) words are stored one after another, e.g., TV\(_x\) = \{a\(_x\)\(_1\), b\(_x\)\(_1\), a\(_x\)\(_2\), b\(_x\)\(_2\)\}. While this is good for sequential work, it fails at SIMD on the GPU. One warp instruction can be executed in a single run if the requested 32 bit words are coalesced. Otherwise, several transactions are
from the field $(1, 2)$ to the field $(3, 4)$ we get the requirement rule: A maximal solution requires one Bishop on each diagonal. For the marked diagonal in Figure 1 Bishop Problem requires large problems we selected the GPU to solve a given Boolean problem. Knowing that the beneficial utilization of the GPU the aim of the experiment in this section is the comparison of the time needed by the CPU or CPU Intel i7 3.06 GHz. Gray values indicate the weakest and maximal speedup of XBOOLE-CUDA-operations on a GTX 470 in comparison to the same marizes the arithmetic average (\(\overline{\sigma}\)) and the standard deviation (\(\sigma\)), as well as the measured minimal and maximal speedup of XBOOLE-CUDA-operations on a GTX 470 in comparison to the same XBOOLE-operations running on a CPU Intel i7 3.06 GHz. Gray values indicate the weakest value in each column.

### 4 Experimental Results

The aim of the experiment in this section is the comparison of the time needed by the CPU or the GPU to solve a given Boolean problem. Knowing that the beneficial utilization of the GPU requires large problems we selected the Bishop Problem [5] on a chessboard of the size 11 \(\times\) 10 which depends on 110 Boolean variables. The Boolean space \(\mathbb{B}^{110}\) contains \(2^{110} = 1,298 \times 10^{33}\) binary vectors.

The Bishop Problem is defined as follows: Let be given a chessboard with \(m\) rows and \(n\) columns. Arrange a number of Bishops on the board not attacking each other, but attacking all empty fields on the board. Each of the 110 Boolean variables carries the information:

\[
    x_{i,j} = \begin{cases} 
    1 & \text{if a Bishop is on the field } (i, j), \\
    0 & \text{otherwise} . 
\end{cases}
\]

We explain the condition of the Bishop Problem using the simple example of Figure 1. A bishop on the field (2, 3) attacks the fields (1, 2) and (3, 4) on one diagonal as well as the fields (4, 1), (3, 2) and (1, 4) on a second diagonal which can be expressed by the following equation:

\[
    C_{2,3} = x_{2,3} \land \overline{x}_{1,2} \land \overline{x}_{3,4} \land \overline{x}_{4,1} \land x_{4,2} \land \overline{x}_{1,4} = 1 .
\]

A maximal solution requires one Bishop on each diagonal. For the marked diagonal in Figure 1 from the field (1, 2) to the field (3,4) we get the requirement rule:

\[
    R_i = C_{1,2} \lor C_{2,3} \lor C_{3,4} = 1 .
\]
Such requirement rules $R_i$ must be hold for the selected chessboard of the size $11 \times 10$ for all 36 diagonals. Hence, all maximal solutions of the Bishop Problem belong to the solution set of the Boolean equation:

$$\bigwedge_{i=1}^{2\times(m+n-3)} R_i = 1 .$$

(4)

In the special case of the chessboard of the size $11 \times 10$ there are 36 functions $R_i$. These functions have a disjunctive form. Using the XBOOLE operation ORTH we get 36 TVLs in ODA-form. All solutions of the Bishop Problem result from a sequence of the 35 XBOOLE operations intersection (ISC) according to (4) for $m = 11$ and $n = 10$. There are 66,049 maximal solutions of the $11 \times 10$ Bishop Problem; 65,536 solutions contain 18 Bishops on the board, 512 solutions contain 19 Bishops on the board, and there is a single solution of 20 Bishops on the board. Figure 2 shows one solution for each of the number of Bishops. More details about Bishop Problems for other sizes of the chessboard are given in [13].

Depending on the associated diagonal line, the TVLs of the functions $R_i$ in (4) contain only two to ten ternary vectors. The intersection operations of (4) yield larger TVLs. We calculated these TVLs using the XBOOLE operations intersection (ISC) for the TVLs $R_i$ in the given order on the CPU until a fixed number of ternary vectors occurs in the intermediate solution. We used the values $10^k$, $k = 2, \ldots, 7$, as limits. A larger value of this limit increases the number of ternary vectors in the TVLs for the remaining ISC operation, but decreases the number of required ISC operations, which are executed for comparison either on the CPU or on the GPU.

Table 2 summarizes our experimental results. Regardless of the chosen value of limit the same number of 66,049 solutions is calculated. This number is very small in comparison to all possible $2^{110} = 1,298 \times 10^{33}$ assignments of Bishop pieces to the $11 \times 10$ chessboard. The column $n_{res} = NTV(res)$ contains the numbers of ternary vectors calculated by the ISC operation between the TVL $f_1$ and the TVL $f_2$. The columns $n_{f_1} = NTV(f_1)$ and $n_{f_2} = NTV(f_2)$ contain the associated numbers of ternary vectors. It can be seen that result of an ISC operation is used as the TVL $f_2$ in the subsequent ISC operation.

The ISC operation can make opposite impacts. The main impact is the intersection of two
The operation ISC appears when the used TVLs depend on disjoint sets of variables; in this case, the cross product of the given TVLs is calculated in which each vector of the first TVL is combined with each vector of the second TVL.

Due to these both effects the number of consecutive intersections increase up to a certain size and decreases for the later ISC operations. The operation ISC($f_1$, $f_2$) must compare each ternary vector of the TVL $f_1$ with each ternary vector of the TVL $f_2$. It should be mentioned that all given numbers of ternary vectors in Table 2 are exactly the same for the CPU and the GPU. Column $t_c$ of Table 2 reports the measured time to calculate the ISC operation on the CPU and column $t_g$ contains the measured time for the same operation on the GPU. These values were captured on the Intel® Core™ i7-3930K CPU @3.20 GHz and the GPU NVIDIA GeForce GTX 690 @0.915 GHz (2 × 1536 cores). The GTX 690 is a dual-GPU based on the Kepler architecture (successor of Fermi-GPU). Depending on the size of the input TVLs one or two GPUs were used (see last column of Table 2).

The column ratio of Table 2 expresses the achieved speedup using the GPU. The reached
Table 3: Comparison between GPU computations and data transfer

<table>
<thead>
<tr>
<th>Limit level</th>
<th>100</th>
<th>1000</th>
<th>10 000</th>
<th>100 000</th>
<th>1 000 000</th>
<th>10 000 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>13 900 ms</td>
<td>13 335 ms</td>
<td>17 020 ms</td>
<td>52 848 ms</td>
<td>81 362 ms</td>
<td>900 228 ms</td>
</tr>
<tr>
<td>gpuISC()</td>
<td>12 022 ms</td>
<td>11 656 ms</td>
<td>15 543 ms</td>
<td>51 291 ms</td>
<td>79 539 ms</td>
<td>894 671 ms</td>
</tr>
<tr>
<td>... invocations</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

GTX 690 – GPU device #1

| Memcpy HtoD | 3404 ms | 1494 ms | 1103 ms | 896 ms | 1429 ms | 8643 ms |
| ... invocations | 452  | 324  | 203    | 177    | 144    | 342    |
| ... size     | 1127 MB | 839 MB | 551 MB | 587 MB | 512 MB | 2446 MB |
| ... bandwidth | 339 MB/s | 561 MB/s | 500 MB/s | 655 MB/s | 358 MB/s | 290 MB/s |
| MemcpyDtoH   | 257 ms  | 227 ms  | 144 ms  | 147 ms  | 99 ms   | 1 ms    |
| ... invocations | 219  | 191   | 110    | 90     | 75      | 26      |
| ... size     | 1177 MB | 1005 MB | 587 MB | 519 MB | 320 MB | 1 MB    |
| ... bandwidth | 4580 MB/s | 4320 MB/s | 3990 MB/s | 3460 MB/s | 3170 MB/s | 521 MB/s |
| Ratio compute memcpy | 0.2 | 1.7 | 8.4 | 44.5 | 43.0 | 97.0 |

GTX 690 – GPU device #2

| Memcpy HtoD | 1108 ms | 1299 ms | 659 ms | 630 ms | 256 ms | 964 ms |
| ... invocations | 444  | 318  | 203    | 177    | 144    | 342    |
| ... size     | 1125 MB | 840 MB | 551 MB | 587 MB | 512 MB | 2446 MB |
| ... bandwidth | 1020 MB/s | 676 MB/s | 836 MB/s | 932 MB/s | 1690 MB/s | 2540 MB/s |
| MemcpyDtoH   | 230 ms  | 147 ms  | 110 ms  | 114 ms  | 138 ms  | 1 ms    |
| ... invocations | 208  | 147   | 95     | 83     | 77      | 33      |
| ... size     | 1057 MB | 653 MB | 462 MB | 397 MB | 323 MB | 1 MB    |
| ... bandwidth | 4650 MB/s | 4350 MB/s | 4080 MB/s | 3390 MB/s | 2290 MB/s | 1030 MB/s |
| Ratio compute memcpy | 0.5 | 2.0 | 13.6 | 62.3 | 165.4 | 871.0 |

Speedup is influenced by both the product of the numbers of input vectors $n_{f_1} \cdot n_{f_2}$ and the number of output vectors $n_{res}$. We take row 4 of Table 2 as an example. An intersection between 1,182,923 and 198 ternary vectors yields 9,535,372 ternary vectors. This large number of output vectors causes 8 atomic operations per thread on average, which serialize the whole GPU execution a lot, so that only a small speedup of 2.1 is reached. In opposite, the last row shows an intersection between 19,605,640 and 1,902,160 ternary vectors, which yields only 66,049 ternary vectors. Here, the GPU performs very well with a speedup of 166.8, because nearly no serialization due to output instructions takes place. The comparison of the columns $n_{f_1} \cdot n_{f_2}$ and ratio of Table 2 indicates as rule of thumb that the GPU should be used if $n_{f_1} \cdot n_{f_2} > 10^8$. 

Figure 3: Time-lines for GTX 690: (a) limit=100, (b) limit=10,000,000.
We evaluated all periods of time for the intersections on the GPU to solve the Bishop problem on the board $11 \times 10$ in detail. Table 3 shows the summarized results. The shorter total time for the limit level 100 in comparison to the limit level 10,000,000 indicates that shorter TVL should be preferred. However, in some cases TVLs with extreme large numbers of vectors cannot be avoided. In such cases the larger speedup strongly contributes to the solution, e.g., for the explored problem the single ISC operation between TVLs of 1,902,160 and 19,605,640 ternary vectors needs more than 1 day and 17 hours on the CPU but is finished after 15 minutes using the GPU (see Table 2).

Table 3 shows furthermore that the utilization of the GPU kernels grows for larger TVLs. This can also be seen in the time-lines of Figure 3 which are captured from the profiler. Figure 3 (a) shows for the smallest TVLs with the value limit=100 in the upper part the activities for the first GPU and thereunder the same information for the second GPU. Each of these three time-lines show top-down the time to copy data from the host to the device, the time to copy data from the device to the host, and the compute time. It can be seen that time intervals between working kernels are not completely filled with data transfers between the host and the device. The preparation of the data on the CPU to speedup the next running kernel on the GPU needs some of this time.

Obviously, there is a side effect that causes the excessive periods of time which are depicted in the upper time-line of Figure 3 (a) and have a total value of 3.404 seconds (see Table 3, limit level 100, GTX 690 GPU device #1, Memcpy HtoD). The aim of this measurement is the time needed to copy the data from the host to the device. The results of the profiler show that the PCIe bus is not exclusively available for this data transfer; e.g., the first copy of 8 MB after 5 seconds of this task to the device number 1 needs 37.291 milliseconds (throughput 214 MB/sec) and the next copy of also 8 MB needs only 1.985 milliseconds (throughput 3.93 GB/sec). On Tesla C2070 GPUs we could not reproduce these profiler results.

Figure 3 (b) uses a much larger scale to show the same information for the largest TVLs with the value limit=10,000,000. The nearly continuous bars in Figure 3 (b) depict that both GPUs are working the most time to calculate the solution.

5 Conclusions

The exponential complexity of many Boolean problems and the growing number of Boolean variables restrict real tasks, which can be solved in an acceptable period of time. The new library XBOOLE-CUDA [16] utilizes one or more GPUs to shorten this time. The result of a benchmark analysis is that XBOOLE-CUDA shorten the time for large Boolean problems depending on the used operation and size of the TVLs by one to two orders of magnitudes. It is an important benefit for the programmer that he can use utilize the power of the GPU without exploring all details of GPU-programming simply by the exchange of the XBOOLE-library with the XBOOLE-CUDA-library.

Experimental results based on the Bishop Problem confirm these results. We found that the more input vectors and the less output vectors appear in an ISC operation the larger is the speedup on the GPU. Before an ISC operation is executed only the numbers of vectors of the input TVLs are known. The product of the number of ternary vectors of both input TVLs can be used as raw indicator to decide whether the ISC operation should be executed on the CPU or on the GPU. We found that the GPU calculates the intersection operation faster than the CPU if this value is larger than $10^8$. Another important parameter is the size of slices of the TVLs, which are calculated by one kernel on the device. This size depends on the memory on the device. We identified the value of $2^{21}$ bytes of one slice for the GPU GTX-690. This value can be associated to XBOOLE-CUDA using the function call:

\[ \text{XB::SetMaxAllocCount}(1<<21); \]

XBOOLE-CUDA was developed and optimized for the Fermi architecture of GPU. Our successful experiments using both GPUs on a graphic board GTX-690 confirm that XBOOLE-CUDA also runs on GPUs with the newer Kepler architecture. The utilization of special properties of this architecture can be a source of further improvements in the future.

The most time consuming task needs altogether 41.457 hours on the CPU. This time could be reduced to 15 minutes using the both GPUs of the graphic board GTX-690 and XBOOLE-
CUDA. This is a speedup of 165.8.

Our experimental results show furthermore that the same result could be calculated using TVLs of different sizes. The overall time was shorter in the case of many XBOOLE-operations with small TVLs in comparison with few XBOOLE-operations or even one XBOOLE-operation with large TVLs. This gives a general hint for modeling of the problem to solve and further research for approaches that already achieve significant speedups on the GPU for smaller TVLs.

References


