

# Chapter 2

## Alternative Approaches for Fast Boolean Calculations Using the GPU

Bernd Steinbach and Matthias Werner

**Abstract** The growing number of Boolean variables requires very efficient approaches to solve the given tasks. We explore the utilization of the GPU for fast parallel Boolean calculations in this chapter. Hundreds of processor cores of the GPU offer a significant potential for improvements. Constraints in their application may restrict the achievable speedup. This chapter gives a taxonomy about possible approaches to solve a problem using a computer. We select one problem from the Boolean domain and summarize alternative approaches for utilizing the GPU. It will be shown that the calculation time could be reduced by several orders of magnitudes for the selected Unate Covering Problem.

### 2.1 Introduction

The technological progress in micro- and nano-electronics leads to both a strong extension of applications and growing requirements for the design of digital systems. Boolean functions are the main instrument for their description. It is well known that the number of function values of a Boolean function exponentially grows depending on the number of Boolean variables.

An important source for improvements to solve Boolean tasks is the utilization of many computer cores for parallel computations. Today's processors contain a small number of cores in the *Central Processing Unit* (CPU), but significantly more cores are available on the *Graphics Processing Unit* (GPU). Hence, the utilization of the GPU to solve exponentially complex tasks of the Boolean domain is an important challenge for scientists and engineers.

There are several approaches for utilizing the GPU in the Boolean domain. In this chapter, we give a taxonomy to classify these approaches. Due to the restricted

---

B. Steinbach (✉) · M. Werner  
Institute of Computer Science, Freiberg University of Mining and Technology,  
09596 Freiberg, Germany  
e-mail: steinb@informatik.tu-freiberg.de

M. Werner  
e-mail: werner3@mailserver.tu-freiberg.de

© Springer International Publishing Switzerland 2015  
G. Borowik et al. (eds.), *Computational Intelligence and Efficiency  
in Engineering Systems*, Studies in Computational Intelligence 595,  
DOI 10.1007/978-3-319-15720-7\_2

space, we select a single Boolean problem, explore different solution methods based on the mentioned taxonomy, and compare both the necessary effort and the benefit achieved. The results of these comparisons can guide scientists and engineers who want to speed up other Boolean problems using a GPU.

## 2.2 The Explored Boolean Problem: Unate Covering

We select the *Unate Covering Problem* (UCP) of given *Petricks Functions* as object of our exploration. This problem has on the one hand an exponential complexity; on the other hand it has a high practical significance; e.g., it is needed for circuit design [3] and data mining [2]. A *Petricks Function*  $P(\mathbf{p})$  depends on non-negated variables  $p_i$  within a conjunctive form as shown in the following example:

$$\begin{aligned} P(\mathbf{p}) = & (p_4 \vee p_5 \vee p_6 \vee p_8) \wedge (p_2 \vee p_3 \vee p_4 \vee p_7 \vee p_8) \wedge \\ & (p_1 \vee p_3 \vee p_4 \vee p_7 \vee p_8) \wedge (p_1 \vee p_4 \vee p_5 \vee p_7 \vee p_8) \wedge \\ & (p_1 \vee p_2 \vee p_5 \vee p_6) \wedge (p_4 \vee p_5 \vee p_6 \vee p_7 \vee p_8) \wedge \\ & (p_1 \vee p_4 \vee p_5 \vee p_6 \vee p_7 \vee p_8) \wedge (p_4 \vee p_6 \vee p_7) = 1. \end{aligned} \quad (2.1)$$

The aim of the Unate Covering Problem consists in finding a subset of variables  $p_i$  such that values 1 of these variables determine the value 1 of the given Petrick Function  $P(\mathbf{p})$ . A minimal solution is a subset of variables  $p_i$  which cannot be reduced without losing the covering of all disjunctions (also called clauses). The 12 minimal solutions of 2 or 3 variables  $p_i$  which satisfy the equation  $P(\mathbf{p}) = 1$  given above, are:

$$\begin{aligned} & p_1 p_4 \vee p_2 p_4 \vee p_4 p_5 \vee p_4 p_6 \vee p_1 p_2 p_6 \vee p_1 p_3 p_6 \vee \\ & p_3 p_5 p_6 \vee p_6 p_7 \vee p_6 p_8 \vee p_1 p_7 p_8 \vee p_5 p_7 \vee p_2 p_7 p_8 = 1. \end{aligned} \quad (2.2)$$

*Exact minimal solutions* of the Unate Covering Problem are minimal solutions consisting of the smallest number of variables. Selected from the set of 12 minimal solutions, the 7 *exact minimal solutions* of 2 variables  $p_i$  are wanted:

$$p_1 p_4 \vee p_2 p_4 \vee p_4 p_5 \vee p_4 p_6 \vee p_6 p_7 \vee p_6 p_8 \vee p_5 p_7 = 1. \quad (2.3)$$

It is our aim to find *all* exact minimal solutions of a given Petrick Function.

## 2.3 Advantages and Drawbacks of the GPU

GPUs were basically developed to accelerate the graphical representation of wanted data on the screen. Hence, GPUs naturally have to transform geometry and texture data to colored pixels. The mathematical basis for such transformations is the multiplication of matrices. Such tasks can be efficiently solved in parallel on many processor cores, because most of the computations are independent of each other

due to the data locality. GPUs have been heavily optimized for such tasks over the years. This has been achieved by throughput-oriented, many-core architectures with hundreds of compute cores. In contrast, modern CPUs have a multi-core architecture (e.g., 8 cores) where each core has to execute single instructions as fast as possible.

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.

Drawbacks of the GPU are caused by the differences in the architecture and programming paradigm. Due to Amdahl's law [1], the theoretically 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.

The following analogy emphasizes differences between sequential and parallel programming. Sequential programming on a single CPU core is like employing a single, skillful and fast workman. Instructions are executed step by step on single data (see *Single Instruction, Single Data*, SISD, [5]). Parallel and concurrent programming on the GPU is like managing many, but slow workers. They should not obstruct each other, even though they have to share scarce resources. If some of the workers have to wait for resources, others stand in to conceal the latencies. The efficiency of such a team depends on the programmer who faces much more challenges than a sequential programmer. The GPU programmer is responsible for cache coherency, optimal work balance and utilization of memory hierarchies—e.g., coalesced memory access is crucial on a GPU.

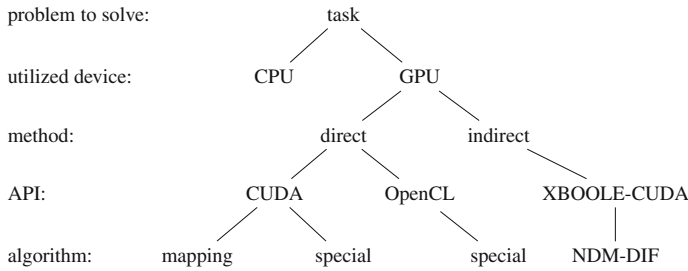
A GPU consists of several, concurrently acting *Multiprocessors* (MPs). Each of these MPs is equipped with shared memory, very fast thread-private registers, load and store units and many compute cores which work in parallel. On Nvidia GPUs the workers are lightweight threads, where each 32 threads are grouped in a warp. Instructions are executed by warps based on the *Single Instruction, Multiple Data* principle, (SIMD) [5]. 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 or memory conflicts are present.

The PCIe bus transfers the data between the main memory of the CPU and the device memory of the GPU. This data transfer is restricted by the bandwidth of PCIe bus of, e.g., 8 GB/s for PCIe v2.x. The GPU itself has internal memory layers designed by decreasing both the size and the latency.

Valuable hints regarding optimal algorithms and programs on Nvidia GPUs are given in [19].

## 2.4 Classification of Approaches for the Utilization of GPUs

Figure 2.1 shows a taxonomy which can be used to classify how a certain task can be solved on a computer. Firstly, the utilized device for computation must be selected. The device, which mainly contributes to solve the problem, can be the CPU or



**Fig. 2.1** Classification of the explored approaches

the GPU. Due to the focus of this chapter, we explore the further taxonomy only for the GPU. The CPU is only used to control the GPU and as reference for the speedup reached on the GPU. For that reason we skip the classification details for the CPU.

Secondly, we decide about the method how the device (GPU) is utilized to solve the task. The direct method faces the programmer with the technical details of the GPU and significantly increases the effort to implement the needed program. The indirect method simplifies the programming effort because general domain-specific operations wrap all details utilizing the GPU.

The third level of the suggested taxonomy deals with the *Application Programming Interface* (API). Such an API provides a set of data types and functions needed to serve a certain purpose, which is the utilization of the GPU in the studied case. The *Compute Unified Device Architecture* (CUDA) [4] is an API provided by Nvidia. Alternatively, the API *Open Computing Language* (OpenCL) [6] can be used to utilize GPUs not only from Nvidia. OpenCL is an API that allows access to heterogeneous platforms consisting of GPUs of different producers, CPU cores, and even *Digital Signal Processors* (DSPs) or *Field-Programmable Gate Arrays* (FPGAs) in the used hardware configuration.

The final question of this taxonomy asks for the algorithm to use. Typically, as for the explored problem, there are many different algorithms [16]. We select two algorithms for CUDA and refer to [14, 17] where further CUDA-algorithms are evaluated which solve the studied problem. Our decision for these two algorithms is based on the ability of the GPU. Knowing that the GPU is heavily optimized for matrix multiplication, in [8] an algorithm was developed that maps the UCP to the multiplication of matrices. Alternatively, we can utilize the properties of the UCP in a special algorithm for the GPU [13]. For comparison of CUDA and OpenCL the same special algorithm was implemented using OpenCL [7]. The domain-specific API XBOOLE-CUDA [18] provides, among others, the operations *Negation according to De Morgan* (NDM) and the set operation *Difference* (DIF) which can be utilized to solve the Unate Covering Problem with significantly less effort for the implementation of the algorithm.

## 2.5 Direct Utilization of the GPU for Solving the Unate Covering Problem

### 2.5.1 Matrix-Multiplication Using CUDA

The Petrick Function  $P(\mathbf{p})$  is mapped to the matrix  $P$  of  $n$  rows and  $k$  columns. The rows are associated top down to the variables  $p_i$  using an increasing order:  $p_1, \dots, p_n$ . Each column of the matrix  $P$  represents one clause of  $P(\mathbf{p})$  where a value 1 indicates the existence of the associated variable in the clause.

All vectors of the Boolean space are assigned top down to the matrix  $A$  of  $l = 2^n$  rows and  $n$  columns. The elements of the result matrix  $R = A \times P$  of  $l = 2^n$  rows and  $k$  columns can be calculated as usual:

$$R[r, c] = \sum_{i=1}^n A[r, i] \cdot P[i, c]. \quad (2.4)$$

The value of  $R[r, c]$  indicates how many variables of the row  $r$  of  $A$  cover the clause associated to the column  $c$  of  $P$ . Hence, if  $R[r, c] = 0$  then the row  $r$  of  $A$  is no solution of the Unate Covering Problem. The evaluation of the matrix  $R$  detects all exact minimal solutions. Figure 2.2 shows an example of this approach. The rows of bold numbers in  $R$  indicate valid covers. The set  $\{(101), (111)\}$  of associated rows of the matrix  $A$  contains the row (101) with the smallest number of values 1 and determines the only exact minimal cover  $p_1 p_3$ .

A subset of variables  $p_i$  is a valid cover when all clauses are covered. It is not needed to know how often each clause is covered, therefore the algebraic matrix multiplication (2.4) can be substituted by the Boolean matrix multiplication (2.5):

$$RB[r, c] = \bigvee_{i=1}^n A[r, i] \wedge P[i, c]. \quad (2.5)$$

**Fig. 2.2** Exact minimal cover  $p_1 p_3$  of  $P(\mathbf{p})$  found by matrix multiplication

$$P(\mathbf{p}) = (p_1) \wedge (p_1 \vee p_2) \wedge (p_2 \vee p_3) \wedge (p_3)$$

$$R = A \times P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 1 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 1 & 2 & 1 & 0 \\ \mathbf{1} & \mathbf{2} & \mathbf{2} & \mathbf{1} \end{bmatrix}$$

The advantage of the Boolean matrix multiplication (2.5) in comparison to (2.4) is that a simpler data type reduces the memory for the matrices. However, the number of rows in the matrices  $A$  and  $R$  exponentially grows depending on the number of variables  $p_i$  and restricts this approach to small instances of the UCP.

### 2.5.2 Ordered Restricted Vector Evaluation Using CUDA

The achievable speedup does not only depend on the number of used processor cores, but also on the implemented algorithm. The ordered restricted vector evaluation is a very powerful algorithm for the Unate Covering Problem. Some intermediate steps help to understand this approach.

Searching for more powerful algorithms we compare the given Petrick Function, e.g. (2.1), with the expression of the corresponding exact minimal cover (2.3). This comparison shows the transformation from the given conjunctive form into an equivalent minimal disjunctive form. This transformation is realized by the distributive law:

$$(a \vee b) \wedge (c \vee d) = a c \vee a d \vee b c \vee b d. \quad (2.6)$$

The application of the absorption law:

$$a \vee a b = a \quad (2.7)$$

reduces the calculated form to conjunctions of the minimal cover. The exact minimal cover can be found by counting the number of variables in the conjunctions and the selection of the conjunctions having a minimal number of variables. It was shown in [16] that a repeated application of (2.7) after the calculation of (2.6) for the intermediate result and a single clause reduces the runtime by a factor of more than  $10^4$ . Therefore, we use this significantly improved algorithm as basis for all further comparisons.

The theoretical basis of another approach is shown in (2.8). Two consecutive negations do not change  $P(\mathbf{p})$ . The inner negation can be executed in constant time according to *De Morgan's Law* (NDM). The outer negation must be executed as *Complement* operation (CPL).

$$\begin{aligned} P(\mathbf{p}) &= 1 \\ \overline{\overline{P(\mathbf{p})}} &= 1 \\ \overline{NDM(P(\mathbf{p}))} &= 1 \\ CPL(NDM(P(\mathbf{p}))) &= 1 \end{aligned} \quad (2.8)$$

A proof in [14] shows that an algorithm based on (2.8) solves the UCP. Using the XBOOLE operations NDM and CPL on a single CPU core, the time to solve an Unate Covering Problem could be reduced by a factor of almost  $10^5$ .

The CPL(NDM(P))-approach achieved a strong improvement. This algorithm needs almost all time for the calculation of the complement operation that must take

into account all  $2^n$  elements of the Boolean space  $\mathbb{B}^n$ . The wanted exact minimal solutions are not distributed over the whole Boolean space, but are characterized by a fixed number of values 1. Hence, the division of the Boolean space in certain subspaces is a starting point for further improvements.

**Definition 2.1** The function  $f(\mathbf{p})$  is symmetric with regard to two variables  $\{p_i, p_j\}$  if

$$f(p_i, p_j, \mathbf{p}_0) = f(p_j, p_i, \mathbf{p}_0). \quad (2.9)$$

**Definition 2.2** The function  $S_i(\mathbf{p})$  is a symmetric function that is symmetric with regard to each pair of variables. The index  $i$  indicates the number of non-negated variables in their conjunctions.

There are  $n + 1$  symmetric functions  $S_i(\mathbf{p})$  in each Boolean space  $\mathbb{B}^n$ . For  $n = 4$  we have, for instance,

$$S_0(p_1, p_2, p_3, p_4) = \bar{p}_1 \bar{p}_2 \bar{p}_3 \bar{p}_4$$

$$S_1(p_1, p_2, p_3, p_4) = p_1 \bar{p}_2 \bar{p}_3 \bar{p}_4 \vee \bar{p}_1 p_2 \bar{p}_3 \bar{p}_4 \vee \bar{p}_1 \bar{p}_2 p_3 \bar{p}_4 \vee \bar{p}_1 \bar{p}_2 \bar{p}_3 p_4$$

$$S_2(p_1, p_2, p_3, p_4) = p_1 p_2 \bar{p}_3 \bar{p}_4 \vee p_1 \bar{p}_2 p_3 \bar{p}_4 \vee p_1 \bar{p}_2 \bar{p}_3 p_4 \vee \bar{p}_1 p_2 p_3 \bar{p}_4 \vee \bar{p}_1 p_2 \bar{p}_3 p_4 \vee \bar{p}_1 \bar{p}_2 p_3 p_4$$

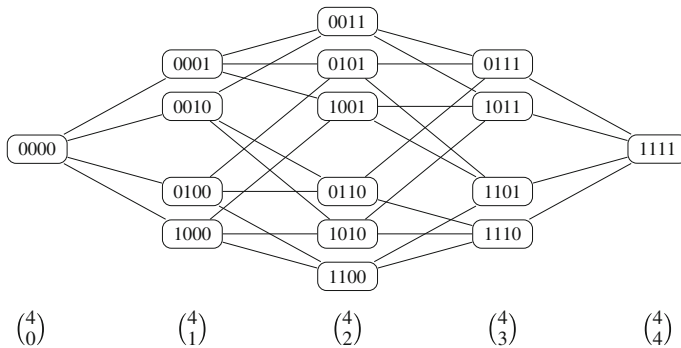
$$S_3(p_1, p_2, p_3, p_4) = p_1 p_2 p_3 \bar{p}_4 \vee p_1 p_2 \bar{p}_3 p_4 \vee p_1 \bar{p}_2 p_3 p_4 \vee \bar{p}_1 p_2 p_3 p_4$$

$$S_4(p_1, p_2, p_3, p_4) = p_1 p_2 p_3 p_4.$$

Figure 2.3 shows how the five symmetric functions  $S_i(\mathbf{p})$  structure the Boolean space  $\mathbb{B}^n$  where edges connect elements which differ in one position. The following theorems guide us to a more powerful algorithm for the UCP.

**Theorem 2.1** For each Boolean space of  $n$  variables it holds:

$$\bigvee_{i=0}^n S_i(\mathbf{p}) = 1. \quad (2.10)$$



**Fig. 2.3** The Boolean space  $\mathbb{B}^4$  structured with regard to the number of values 1

**Theorem 2.2** For each Petrick Function  $P(\mathbf{p}) \neq 0$  of  $n$  variables it holds:

$$P(\mathbf{p}) \wedge S_0(\mathbf{p}) = 0. \quad (2.11)$$

**Theorem 2.3** For each Petrick Function  $P(\mathbf{p}) \neq 0$  of  $n$  variables it holds:

$$P(\mathbf{p}) \wedge S_n(\mathbf{p}) = S_n(\mathbf{p}). \quad (2.12)$$

Theorem 2.1 directly follows from Definition 2.2. Theorem 2.2 holds because  $P(\mathbf{p}) \neq 0$  cannot be covered without any non-negated variable  $p_i$ . Theorem 2.3 holds because a  $P(\mathbf{p}) \neq 0$  does not include any negated variable  $p_i$ .

Theorem 2.1 is the key to split the calculation of the complete complement into  $n + 1$  difference operations ( $DIF(f, g) = f \wedge \bar{g}$ ) between  $S_i(\mathbf{p})$  and  $NDM(P(\mathbf{p}))$ :

$$\begin{aligned} P(\mathbf{p}) &= 1, \\ \overline{\overline{P(\mathbf{p})}} &= \overline{NDM(P(\mathbf{p}))} = 1 \wedge \overline{NDM(P(\mathbf{p}))} = 1, \\ \bigvee_{i=0}^n [S_i(\mathbf{p}) \wedge \overline{NDM(P(\mathbf{p}))}] &= \bigvee_{i=0}^n [S_i(\mathbf{p}) \wedge \overline{NDM(P(\mathbf{p}))}] = 1, \\ \bigvee_{i=0}^n DIF(S_i(\mathbf{p}), NDM(P(\mathbf{p}))) &= 1. \end{aligned} \quad (2.13)$$

The evaluation of the symmetric functions  $S_i$  can be organized in increasing order. In the visualization of a Boolean space as shown in Fig. 2.3, this ordered procedure evaluates the columns of binary vectors from the left to the right. Due to Theorem 2.2 the evaluation of  $S_0$  can be skipped. Due to the increasing order of the evaluated symmetric functions  $S_i$ , the first non-empty solution set of the difference operation of  $S_i$  contains all wanted exact minimal solutions of the UCP. Hence, all other difference operations of  $S_j$ ,  $j > i$ , can be skipped. A precise representation of this approach is given in Algorithm 1. This algorithm terminates due to Theorem 2.3.

---

**Algorithm 1:** Solve the UCP by Ordered Evaluation of Symmetric Functions

---

**Input:** Petrick Function  $P(\mathbf{p}) = d_1(\mathbf{p}) \vee \dots \vee d_k(\mathbf{p})$

**Output:** all exact minimal solutions  $AEMS$  of  $P(\mathbf{p}) = 1$

---

```

1 begin
2    $AEMS \leftarrow \emptyset$ 
3    $i \leftarrow 1$ 
4   while  $AEMS = \emptyset$  do
5     generate  $S_i(\mathbf{p})$ 
6      $AEMS \leftarrow DIF(S_i(\mathbf{p}), NDM(P(\mathbf{p})))$ 
7      $i \leftarrow i + 1$ 

```

---

The symmetric function  $S_i(\mathbf{p})$  is generated in line 4 of Algorithm 1 and contains all permutations of  $i$  values 1 within an  $n$ -bit vector. Due to this method of generation



we call it permutation vector  $pv$ . Similarly, each clause of the Petrick Function  $P(\mathbf{p})$  can be represented as an  $n$ -bit binary vector where a value 1 in the position  $j$  indicates the appearance of the variable  $p_j$  in the clause. We call the vector of all clauses of a Petrick Function  $P(\mathbf{p})$  clause vector  $cv$ .

The operations in line 6 of Algorithm 1 require the comparison of each element of  $pv$  with each element of  $cv$ . The necessary and sufficient condition that the  $n$ -bit binary vector  $pv[j]$  covers all  $cv.elements$  clauses of the Petrick Function  $P(\mathbf{p})$  is:

$$\forall cv[k] \in cv : \quad pv[j] \wedge cv[k] \neq 0. \quad (2.14)$$

The  $\forall$ -quantifier of (2.14) provides one more possibility to restrict the computation effort. The vector  $pv[j]$  is no solution if the result of  $pv[j] \wedge cv[k] = 0$  for one value of  $k$ . Hence, all further evaluations for such a vector  $pv[j]$  can be skipped.

The calculation of (2.14) can be realized in parallel on the GPU for the different permutation vectors  $pv[j]$  as shown in Algorithm 2.

Using CUDA [4, 19] we have implemented a program [13, 17] in which Algorithm 2 is used to realize the main step 6 of Algorithm 1. The number of permutations  $\binom{n}{i}$  can achieve such a large value that memory conflicts occur. We avoid this problem in our implementation by splitting  $S_i(\mathbf{p})$  into slices of a fixed maximal value. These slices are sequentially evaluated on the GPU.

---

**Algorithm 2:**  $sv = \text{BDIF\_kernel}(pv, cv)$  for the GPU

---

**Input:** the permutation vector  $pv$  of a symmetric function  $S_i(\mathbf{p})$  and the clause vector  $cv$  of the Petrick Function  $P(\mathbf{p})$

**Output:** solution vector  $sv(\mathbf{p})$  which holds  $P(\mathbf{p}) = 1$

```

1 begin
2   for  $ic \leftarrow 0, ic < cv.elements, ic \leftarrow ic + 1$  do
3     if  $pv.vector[ip] \wedge cv.vector[ic] = 0$  then
4       break
5   if  $ic = cv.elements$  then
6      $sv.vector[is] \leftarrow pv.vector[ip]$  ▷ add solution
7      $is \leftarrow is + 1$ 

```

---

### 2.5.3 Ordered Restricted Vector Evaluation Using OpenCL

The APIs CUDA [4, 19] and OpenCL [7] require different implementation details but realize the same paradigm for parallel programs. Therefore, the powerful algorithm of the previous subsection can be implemented using OpenCL. The execution of both the CUDA and the OpenCL implementation of the same algorithm on the same GPU allows a comparison of the influence of these APIs to the needed runtime.

Using Algorithm 2 to realize the main step 6 of Algorithm 1, an OpenCL program was implemented in [7]. The advantage of this OpenCL implementation is that it

can be used for different GPUs and even for multi-core CPUs. Experiments of [7] confirm the flexibility of OpenCL with regard to the utilization of different hardware resources.

A more efficient data management leads for small benchmarks to faster calculations in comparison with the CUDA implementation of [13, 17]. However, for the largest benchmark of 32 Boolean variables and 1024 clauses in the Unate Covering Problem the OpenCL implementation needs approximately twice the time of the CUDA implementation using the same GPU Tesla C2070. This result confirms the advantage of a well-developed special API for a restricted set of GPUs.

## 2.6 Indirect Utilization of the GPU for Solving the Unate Covering Problem

### 2.6.1 Implementation of XBOOLE Using CUDA

XBOOLE is a library of more than 100 functions which can be used within programs written in C or C++ to solve a wide field of Boolean problems [9, 15]. In order to make this chapter self-contained, we give a very brief introduction to XBOOLE.

XBOOLE uses the dash element (–) to express the combination of the Boolean values 0 and 1. A ternary vector with  $d$  dash elements represents  $2^d$  binary vectors. In this way the needed memory to store sets of binary vectors and the time for their computation can be reduced exponentially. The *Ternary Vector List* (TVL) is the main data structure. An orthogonal TVL does not contain any binary vector in more than one ternary vector. Most of the XBOOLE-operations compute an orthogonal TVL. Each non-orthogonal TVL can be transformed into an orthogonal TVL using the XBOOLE-operation ORTH.

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 $A \triangle B$ , and
CSD	the Complement of the Symmetrical Difference $A \overline{\triangle} B$

can be used to compute needed new sets of binary vectors.

Using the form attribute a TVL is able to represent a Boolean function by each of the four basic forms [10]:

*Disjunctive Form* (D),  
*Conjunctive Form* (K),  
*Antivalence Form* (A), or  
*Equivalence Form* (E).

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 ( $\overline{f}$ ),
UNI	disjunction ( $f \vee g$ ),
ISC	conjunction ( $f \wedge g$ ),
DIF	difference ( $f \wedge \overline{g}$ ),
SYD	antivalence ( $f \oplus g$ ), and
CSD	equivalence ( $f \odot g$ ).

The Boolean Differential Calculus [9, 11, 15] extends the Boolean Algebra by operations which evaluate certain changes of Boolean values.

XBOOLE directly provides all  $k$ -fold derivative operations:

DERK	the $k$ -fold derivative,
MINK	the $k$ -fold minimum,
MAXK	the $k$ -fold maximum;

and all vectorial derivative operations:

DERV	the vectorial derivative,
MINV	the vectorial minimum,
MAXV	the vectorial maximum.

The portable source code of XBOOLE is used to provide programming libraries for several types of CPUs and versions of operating systems. The time to solve complex Boolean problems can be reduced when time-consuming operations of XBOOLE are executed on the GPU.

Following this idea, a compatible library XBOOLE-CUDA was implemented in [18] using CUDA. 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. XBOOLE-CUDA provides the same operations as XBOOLE. Hence, the migration from an XBOOLE-program to an XBOOLE-CUDA-program simply requires the replacement of the library. XBOOLE-CUDA-operations decide by the size of the 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.

The speedup achieved by XBOOLE-CUDA strongly depends on the executed operation and the size of the data. In best case a speedup of more than  $13 * 10^3$  was realized using a special brute force algorithm. Table 2.1 summarizes the arithmetic average ( $\emptyset$ ), the standard deviation ( $\sigma$ ), as well as the measured minimal and maximal speedup of XBOOLE-CUDA-operations on a graphics board GTX 470 in comparison to the same XBOOLE-operations running on a CPU Intel i7 3.06 GHz.

**Table 2.1** Speedups of XBOOLE-CUDA using the GPU GTX 470 (448 Cores)

Operation	$\emptyset$	$\sigma$	Minimum	Maximum
CPL()	5.04×	2.21×	2.11×	8.11×
ISC()	54.90×	7.48×	43.54×	67.32×
UNI()	34.25×	15.57×	13.83×	61.44×
DIF()	19.63×	15.03×	2.22×	43.15×
SYD()	17.61×	5.40×	11.02×	24.74×
CSD()	197.58×	287.55×	3.47×	843.54×
DERK()	29.95×	10.26×	19.90×	52.63×
MINK()	68.78×	30.31×	41.06×	136.75×
MAXK()	56.24×	16.80×	42.25×	95.97×
DERV()	18.84×	2.63×	16.24×	24.01×
MINV()	77.42×	22.93×	55.63×	115.05×
MAXV()	43.19×	6.36×	32.69×	52.09×
ORTH()(xm3)	448.48×	728.29×	21.20×	2196.97×

### 2.6.2 Difference-Operations of XBOOLE-CUDA

Algorithm 3 shows the simple implementation of the unate covering problem using XBOOLE-CUDA. The theoretical basis for this algorithm is Algorithm 1.

The Petrick Function  $P(\mathbf{p})$  is given as object 1 of the memory list in the file petInput.sdt. A file of the type sdt contains all data belonging to a recent XBOOLE-system. Such files are used to exchange XBOOLE-systems between programs. The LDS-operation in line 2 of Algorithm 3 reads the file petInput.sdt. XBOOLE stores all objects in a special box-system and uses pointers for their access. The GET\_ML-operation in line 3 of Algorithm 3 delivers the pointer ctvl to the TVL of all clauses of  $P(\mathbf{p})$ .

The NDM-operation in line 6 of Algorithm 1 calculates in each swap of the while-loop the same result. Hence, in Algorithm 3 this NDM-operation is moved into line 4 outside of the while-loop. The wanted vectors of the exact minimal solution must be stored in a separate TVL. The EMPTY operation in line 5 prepares an empty list having all variables of the Petrick Function  $P(\mathbf{p})$ . The final step for preparation is the initialization of the variable *min\_cover* which indicates the index of the symmetric function which must be evaluated.

The XBOOLE-operation NTV in line 7 calculates the number of ternary vectors of the solution-TVL. The loop in lines 7 to 9 terminates when the number of ternary vectors of the solution-TVL is greater than 0. The function generate\_permutations() in line 8 is implemented in C and uses the XBOOLE-operation SDATV for storing the generated permutation vectors into the TVL with the access pointer ptvl.

The main work to solve the Unate Covering Problem is realized by the DIF-operation in line 9. XBOOLE-CUDA uses the CPU for small TVLs of permutation vectors. Time-consuming DIF-operations of larger TVLs of permutation vectors are executed on the GPU. Extremely large TVLs are split into slices so that no memory problems occur. The solution-TVL is registered in the XBOOLE-memory list as object number 2 using the XBOOLE-operation PUT\_ML in line 10. The STS operation stores all data of the extended XBOOLE-system in the file petSolution.std for later evaluation.

---

**Algorithm 3:** petSolution.std = xb\_cuda\_ndm\_dif\_p(petInput.std)

---

**Input:** file petInput.std that contains all clauses of of the Petrick Function  $P(\mathbf{p})$  as object 1

**Output:** file petSolution.std that additionally contains all exact minimal solutions of  $P(\mathbf{p}) = 1$  as object 2

```

1 begin
2   LDS(petInput.std)                                ▷ load task file
3   GET_ML(1, &ctvl)                                  ▷ select P
4   NDM(&ctvl, &ctvl)                                ▷ negate P according to De Morgan's Law
5   EMPTY(&ctvl, &stvl)                              ▷ prepare solution TVL
6   min_cover ← 1
7   while (NTV(&stvl) = 0) do
8     generate_permutations(min_cover++, &ptvl)
9     DIF(&ptvl, &ctvl, &stvl)                        ▷ main task (CPU or GPU)
10  PUT_ML(2, &stvl)
11  STS(petSolution.std)                              ▷ store solution file

```

---

## 2.7 Experimental Results

The mapping of the UCP to the matrix multiplication is explained in Sect. 2.5.1. The implementation of this mapping-approach in [8] achieves in the best case a speedup of 3.427 (time on GPU: 236.532 ms; time on CPU: 810.594 ms) using the 64 cores of the GPU GeForce 9600 GT in comparison to a single core of the CPU Intel i7 940 (2.93 GHz) solving the benchmark of 16 variables and 256 clauses. The evaluation of all  $2^n$  Boolean vectors restricts this approach to such very small UCPs.

The achieved strong improvement requires much larger benchmarks for time measurement. We used a Petrick Function of 32 variables  $p_i$  and 1024 clauses for comparison. All experiments utilized one core of the CPU Intel Xeon X5650 (2.67 GHz) or the 448 cores of a GPU Tesla C2070. Table 2.2 shows the experimental results measured on this hardware. The speedups achieved on the GPU are calculated using the reference introduced in Sect. 2.5.2.

**Table 2.2** Benchmark results: Petrick Function of 32 variables  $p_i$  and 1024 clauses

Device	Method	API	Algorithm	Time in ms	Speedup	Implementation effort
GPU	Direct	CUDA	2 within 1	20.849	$1.2 \times 10^{11}$	Expensive
GPU	Direct	OpenCL	2 within 1	43.804	$5.7 \times 10^{10}$	Expensive
GPU	Indirect	XBOOLE_CUDA	3	213.600	$1.2 \times 10^{10}$	Minor
CPU	Indirect	XBOOLE	3	786.967	$3.2 \times 10^9$	Minor

## 2.8 Conclusions

The large number of cores of a GPU is an important source to reduce the time for hard Boolean problems. There are two ways for utilizing the GPU:

1. the direct implementation using a given API;
2. the indirect implementation utilizing a domain-specific basic software, such as XBOOLE.

The first way provides unrestricted possibilities utilizing the properties of the GPU, but requires an expensive effort for the implementation. We showed that the results of this way significantly depends on the selected algorithm in a range of 3.4 for matrix multiplication to  $1.2 \times 10^{11}$  using CUDA and the algorithm *Ordered Restricted Vector Evaluation*. The implementation of the same algorithm using OpenCL reduces to speedup to one half but extends the usable devices.

The second way allows us to utilize the power of the GPU without exploring all details of GPU-programming. A much simpler XBOOLE-program requires only 37.5 times more time in comparison to the fastest CUDA implementation. The GPU ported library XBOOLE\_CUDA needs only 10 times more time in comparison to the fastest CUDA implementation, but is more than  $10^{10}$  times faster than the reference implementation.

## References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pp. 483–485. AMC, New York (1967). doi:[10.1145/1465482.1465560](https://doi.org/10.1145/1465482.1465560)
2. Borowik, G.: Data mining approach for decision and classification systems using logic synthesis algorithms. In: Klempous, R., Nikodem, J., Jacak, W., Chaczko, Z. (eds.) Proceedings of the Advanced Methods and Applications in Computational Intelligence, Springer International Publishing, pp. 3–23, ISBN: 9783319014357. (2014). doi:[10.1007/978-3-319-01436-4\\_1](https://doi.org/10.1007/978-3-319-01436-4_1)
3. Cordone, R., Ferrandi, F., Sciuto, D., Wolfler Calvo, R.: An efficient heuristic approach to solve theunate covering problem. In: Proceedings of the Conference on Design, Automation and Test in Europe, Paris, France, pp. 364–371 (2000)
4. Farber, R.: CUDA Application Design and Development. pp. 1–336, Elsevier LTD, Oxford (2011). ISBN 0123884268

5. Flynn, M.J.: Some computer organizations and their effectiveness. In: IEEE Transactions Computer, vol. 21(9), IEEE Computer Society, Washington, DC, USA, ISSN 0018-9340, pp. 948–960 (1972). doi:[10.1109/TC.1972.5009071](https://doi.org/10.1109/TC.1972.5009071)
6. Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Elsevier Science and Technology, pp. 1–296, ISBN 978-0123877666. (2011)
7. Grehl, S.: Vergleich von Implementierungen des Unate Covering Problems mit OpenCL und CUDA. Freiberg University of Mining and Technology, Project-Thesis (2013)
8. Paul, E., Steinbach, B., and Perkowski, M.: Application of CUDA in the Boolean domain for the unate covering problem. In: Steinbach, B. (ed.) Boolean Problems, Proceedings of the 9th International Workshops on Boolean Problems, Freiberg University of Mining and Technology, Freiberg, 16–17 September 2010, pp. 133–142 (2010). ISBN 978-3-86012-404-8
9. Posthoff, Ch., Steinbach, B.: Logic Functions and Equations—Binary Models for Computer Science. Springer, Dordrecht, The Netherlands (2004)
10. Steinbach, B., Posthoff, Ch.: An Extended Theory of Boolean Normal Forms. In: Proceedings of the 6th Annual Hawaii International Conference on Statistics, Mathematics and Related Fields, Honolulu, Hawaii, pp. 1124–1139 (2007)
11. Steinbach, B., Posthoff, Ch.: Boolean differential calculus-theory and applications. In: Journal of Computational and Theoretical Nanoscience, American Scientific Publishers, Valencia, California, USA, ISSN 1546-1955, vol. 7, no. 6, pp. 933–981 (2010)
12. Steinbach, B., Werner, M.: Fast boolean calculations using the GPU. In: Chaczko, Z., Gaol, F.L., Chiu C. (eds.) Proceedings of the 2nd Asia-Pacific Conference on Computer Aided System Engineering APCASE 2014, Book of Extended Abstracts, Bali Dynasty Resort, Bali, Indonesia, 10–12, pp. 86–89, ISBN 978-0-9924518-0-6 February 2014
13. Steinbach, B. and Posthoff, Ch.: Fast calculation of exact minimal unate coverings on both the CPU and the GPU. In: Roberto Moreno-Díaz, Franz Pichler and Alexis Quesada-Arencibia: Proceedings of the Computer Aided Systems Theory—EUROCAST 2013, 14th International Conference, Las Palmas de Gran Canaria, Spain, February 2013, Revised Selected Papers, Part II, Lecture Notes in Computer Science vol. 8112, Springer, pp. 234–241, ISBN: 978-1-612-08292-9, (2013). doi:[10.1007/978-3-642-53862-9\\_30](https://doi.org/10.1007/978-3-642-53862-9_30)
14. Steinbach, B. and Posthoff, Ch.: Improvements of the construction of exact minimal covers of boolean functions. In: Roberto Moreno-Díaz, Franz Pichler and Alexis Quesada-Arencibia: Proceedings of the Computer Aided Systems Theory—EUROCAST 2011, 13th International Conference, Las Palmas de Gran Canaria, Spain, February 6–11, 2011, Revised Selected Papers, Part II, Lecture Notes in Computer Science Volume 6928, Springer, pp. 272–279, ISBN: 978-3-642-27578-4, (2012). doi:[10.1007/978-3-642-27579-1\\_35](https://doi.org/10.1007/978-3-642-27579-1_35)
15. Steinbach, B., Posthoff, Ch.: Logic Functions and Equations-Examples and Exercises. Springer Science + Business Media B.V. (2009)
16. Steinbach, B., Posthoff, Ch.: Parallel Solution of Covering Problems— Super-Linear Speedup on a Small Set of Cores. GSTF International Journal on Computing, Global Science and Technology Forum (GSTF), Singapore, ISSN: 2010-2283, vol. 1, No. 2, pp. 113–122 (2011)
17. Steinbach, B., Posthoff, Ch.: Sources and obstacles for parallelization —a comprehensive exploration of the unate covering problem using both CPU and GPU. In: Astola, J., Kameyama, M., Lukac M., and Stankovi R. S. (eds.): GPU Computing with Applications in Digital Logic. Tampere International Center for Signal Processing. TICSP series # 62, Tampere 2012, pp. 63–96, ISBN 978-952-15-2920-7, ISSN 1456-2774
18. Werner, M.: Parallelisierung von XBOOLE-Operationen mit CUDA. Freiberg University of Mining and Technology, Master-Thesis (2014)
19. Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming. ISBN: 9780133261509, Pearson Education (2013)

Computational Intelligence and Efficiency in  
Engineering Systems

Borowik, G.; Chaczko, Z.; Jacak, W.; Luba, T. (Eds.)

2015, XIV, 442 p. 170 illus., 41 illus. in color., Hardcover

ISBN: 978-3-319-15719-1