High Performance Implementation of Winner Selection Algorithms

 Conducted at Svenska Spel

 Johan Gille
 Jimmy Helmersson
Abstract

High Performance Implementation of Winner Selection Algorithms

Johan Gille & Jimmy Helmersson

The process to find candidates, that fit certain win-conditions, from a collection of wagers is the purpose of a winner selection algorithm. These candidates are filtered out in different groups called winning groups depending on the win-conditions. Svenska Spel AB is the largest company in the regulated gaming market in Sweden.

It is crucial for winner selection algorithms at Svenska Spel to run as efficiently as possible, since results needs to be produced in near real time for many different games. Additionally, new services and features can be developed by having efficient algorithms that are not feasible with current sequential implementations.

In this paper a variety of parallel approaches using OpenMP, Pthreads and CUDA are investigated to create efficient implementations of winner selection algorithms for the games Lotto and Bomben at Svenska Spel. Various preprocessing schemes are used on the original dataset to affect calculation times and enable different types of algorithms. Some of the algorithms are also extended, meaning they run on several, if not all, permutations of possible outcomes, something that is not possible to execute in reasonable time with the current implementations. If these extended runs are feasible then it enables the possibility for new functionality with a more detailed statistical overview that were too compute heavy or slow to determine before.

OpenMP and Pthreads run on the CPU while the CUDA algorithm uses the GPU. By using a baseline implementation they are all individually compared to determine their speed up. The effect preprocessing overhead and data allocation for CUDA is also evaluated.

It results indicate that by performing all required preprocessing for the different approaches do not yield any performance gain. The preprocessing time, and data transfer to the GPU, occupies too large of a chunk of the execution time that it is impossible to gain anything from doing the computations in parallel. However, by utilizing the preprocessed data several times significant speed up can be achieved. The extended algorithm for Lotto runs more than 200 times faster on the GPU compared to the baseline algorithm. The parallel implementations for Bomben ranges from seven to 20 times the speed up, which is not as impressive but arguable usable in different cases.
Contents

Acknowledgement ........................................................................................................... 4

1 Introduction .................................................................................................................. 5

2 Background .................................................................................................................. 7
  2.1 Parallel Approaches ............................................................................................... 7
    2.1.1 OpenMP ........................................................................................................ 7
    2.1.2 Pthreads ........................................................................................................ 8
    2.1.3 CUDA ............................................................................................................ 8
  2.2 Optimization Techniques ....................................................................................... 8
    2.2.1 Hamming Weight in GNU Compiler Collections ........................................... 9
    2.2.2 Warps and Occupancy ................................................................................... 9
  2.3 Svenska Spel Games .............................................................................................. 10
    2.3.1 Lotto ............................................................................................................. 10
    2.3.2 Bomben .......................................................................................................... 12

3 Problem and Contributions ......................................................................................... 13
  3.1 Objectives .............................................................................................................. 13
  3.2 Hardware ................................................................................................................. 14

4 Method ........................................................................................................................ 15
  4.1 Roofline Model ..................................................................................................... 15
  4.2 Data Structures and Preprocessing ...................................................................... 16

5 Implementation - Lotto ............................................................................................... 19
  5.1 Data structure ....................................................................................................... 19
  5.2 Preprocessing ....................................................................................................... 20
    5.2.1 Offset Array .................................................................................................. 20
    5.2.2 Single Rows with Systems .......................................................................... 21
    5.2.3 Single Rows .................................................................................................. 22
  5.3 Sequential Implementation .................................................................................... 22
    5.3.1 Baseline ....................................................................................................... 22
    5.3.2 Single Rows with Systems .......................................................................... 25
    5.3.3 Single Rows .................................................................................................. 26
  5.4 Parallel Implementations ....................................................................................... 26
    5.4.1 OpenMP ...................................................................................................... 27
    5.4.2 Pthreads ....................................................................................................... 28
    5.4.3 CUDA ........................................................................................................... 31
  5.5 Extended Implementation for All Rows ................................................................. 36
10.1.4 Popcount Optimizations ............................................. 78
10.1.5 Adding Bonus Numbers to the Lotto Extended Implementation ............................................. 78
10.1.6 Sorted System Preprocessing ............................................. 78
10.1.7 Evaluate GPU algorithms on different hardware .......... 78

References ............................................................................. 79

Appendix A: Work Distribution .................................................. 81
   Joint work ............................................................................. 81
   Work focused by Johan ......................................................... 81
   Work focused by Jimmy ......................................................... 81
Acknowledgements

We would like to thank our supervisors at Svenska Spel, Jan-Olof Hellstrand and Magnus Markusson, for their help and guidance during the project.

Also, special thanks to David Black-Schaffer for providing critical feedback and discussion on both the work and report.
1. Introduction

Winner selection is a crucial aspect of large scale gambling systems. A winner selection algorithm refers to the process of iterating bets from participants in a game and checking if they fulfill a condition to be classified as a winner. The win condition is dependent on the rules of the game, it could for example be to predict the number of goals in a match. For some cases the game does not only have a win or loss outcome, instead there are different groups of winners depending on how many correct guesses they have. These groups are called winning groups. The algorithms and code which handle this must be reliable and of high performance, because many games today need quick feedback ranging from every second to every hour.

Svenska Spel AB hosts several different sports- and number-based games that requires an high performance back-end implementation of various winner selection algorithms. All of their implementations today consist of highly optimized single-threaded code and the performance only vary with the evolution of hardware. The next step that they want to extended towards is to implement algorithms which can utilize parallel execution to gain further performance.

Parallel computing is a concept that has been around for many years. Early as in 1958 John Cocke and Daniel Slotnick discussed ideas for parallelism in numerical calculations in a IDM memo [22]. Ever since new architectures, programming languages, hardware components and more have been developed to be able to utilize the benefits of parallelism more efficiently.

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem [13]. Therefore, to be able to utilize parallel computing for a problem it must be possible to break it down to discrete parts that can be solved concurrently.

October 11, 1991 marks the release of the first Graphics Processing Units (GPU) [16]. The original purpose of the GPU was to render graphics, which involves making calculations on every single pixel of an image and then displaying them on a monitor. A GPU is specifically manufactured to be able to handle thousands of these calculations in parallel. As of today, GPUs have thousands of cores and each one of them can work in parallel on different sets of pixels. For instance, the GPUs included in the NVIDIA 10-series ranges from around 700 to 3500 cores per unit [15]. Note that compared to a CPU a single core of a GPU is working on a lower clock frequency but together the cores still outperforms at parallel workloads.

Since the release of the GPU it has been realized that the characteristics of this architecture also can be utilized in areas other than just graphics rendering.
Applications that executes the same instruction on different data is well suited for a *Single Instruction, Multiple Data* (SIMD) platform, which is offered by the GPU. The concept of using a GPU for general purpose computations is called *General-Purpose computing on Graphics Processing Units* (GPGPU).

The main difficulty of utilizing the GPU for GPGPU was to write code that effectively uses the resources that the GPU has to offer. To mitigate these difficulties NVIDIA introduced a programming model and platform called *Compute Device Unified Architecture* (CUDA) for programming and manipulating a GPU to general purpose computing. This platform enables the possibility to send C-like and Fortran code straight to the GPU.

In this report we will describe different methods and APIs which can be used to utilize parallel computing on both the CPU and GPU when implementing different winning selection algorithms that Svenska Spel uses. Further explanation about what will be accomplished and investigated in this report can be found under chapter 3, "Problem".
2. Background

There are several approaches when programming a parallel program. In the upcoming sections; 2.1.1, 2.1.2, 2.1.3 three different approaches for parallel programming are presented. For the GPU an alternative to using CUDA would be OpenCL[12]. However this project will only focus on CUDA for the GPU and not OpenCL, also no message passing approach such as MPI[5] will be evaluated in this report. In section 2.2 some important optimization techniques are covered. Finally, the two games Bomben and Lotto is described in section 2.3, including a short paragraph about the challenges.

2.1 Parallel Approaches

In the three following sections the different tools, libraries and APIs are covered that are used for this report.

2.1.1 OpenMP

OpenMP (abbreviation for Open Multi-Processing) is a set of compiler directives that offers an high-level API that supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The main goals of OpenMP is to provide an API that is portable between different platforms, easy to use and a standard between several shared memory architectures [3].

OpenMP is used in schools as an education tool to train new generations of computer scientists and engineers while also used to further research and develop industrial applications. Most relevant for this report, it is also used in High Performance Computing, government research labs, and industry [6].

With OpenMP it is possible to modify a sequential program to execute in parallel with a few compiler directives which are embedded in C/C++ and Fortran source code. Additionally, the program will still execute as intended as a sequential program when compiled without the flag to enable OpenMP.

OpenMP implements thread-based parallelism and utilizes the the Fork-Join Model of parallel execution, which means that there is a master thread that handles the creation of a team that consists of parallel threads. The team executes a given task in parallel among the threads and when they complete they synchronize and terminate [3].
2.1.2 Pthreads
Parallel programs can utilize threads in a shared memory multiprocessor architecture to operate on a distributed workload.

Pthreads is the standard threading interface, short for POSIX-threads.

When compared to the cost of creating and managing a process, a thread can be handled with less overhead and fewer resources. On modern, multi-core machines, parallel programming is suited, and whatever applies to parallel programming in general, applies to parallel pthreads programs. These can be hard to do with pthreads since it is a low-level API. Having a workload that can be divided into independent parts and executed in any order makes it applicable for the parallel-model.

2.1.3 CUDA
CUDA (Introduced as Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA [8].

As mentioned under the introduction, the GPU is a powerful tool for executing code in parallel but it can be difficult to develop programs that can utilize it. However, with CUDA the programmer defines, with keywords, which segments of the application to run sequentially on the CPU and which parts to run parallel on the GPU [8].

The CUDA toolkit, which is provided by NVIDIA, delivers a comprehensive development environment for C/C++. To compile C/C++ code NVIDIA have made a LLVM-based compiler, compiled with nvcc [7] and for the Fortran code there is a PGI CUDA Fortran compiler provided by The Portland Group. However, during this project only C/C++ is used.

A CUDA GPU have a number of Streaming Multiprocessors (SM) that controls a number of cores. As seen in figure 2.1, each SM have its own L1 cache but they all share the L2 cache and global device memory. To not waste any performance, unnecessary memory transactions to the global device memory should be avoided if possible, this will be discussed in section 2.2.2.

2.2 Optimization Techniques
The following sections include information about some optimization techniques that are important to know some background about for the rest of the report. Section 2.2.1 describes an efficient way to count set bits in a binary number and why it is especially important. After that, section 2.2.2 describes more in depth techniques to optimize algorithms on a GPU.
2.2.1 Hamming Weight in GNU Compiler Collections

The winner selection algorithm for Lotto, covered in chapter 5, relies on an optimization where the Hamming Weight is calculated for 64bit numbers. This is possible since the data structure, covered in section 5.1, of Lotto is designed to make the comparison of rows efficient. The purpose of this section is to clarify what the Hamming Weight is, why it is important and how it is used in C/C++.

In coding theory, the definition of The Hamming Weight is the number of nonzero digits in a word [17]. For instance, the binary representation of number 13 is 1101, which consists of three nonzero bits. By following the definition this results in a Hamming Weight of three.

During this project the GNU Compiler Collection (GCC) is used to compile the code. GCC has several built-in functions, including a function for population count (popcount) [10]. Popcount calculates the number of 1-bits in a unsigned integer, which is by definition the Hamming Weight of the unsigned integer.

Popcount falls under the category of low level operations for built-in functions in GCC. The functions under this category map to machine-specific instructions whenever possible. By using hardware that support these machine-specific instructions performance is significantly improved compared to checking each bit separately [18].

2.2.2 Warps and Occupancy

With CUDA, kernels are executed as grids that contain blocks. These blocks contains threads that communicate within their block. A group of 32 threads is called a warp and instructions are issued per warp. A warp will stall if an
operand is not ready; this issues a context switch to another warp. To ensure that context switching do not consume unnecessary compute time all registers and shared memory are allocated for the entire lifetime of the block instead of being saved and restored for every context switch.

Occupancy is how much of the GPU that is being used, more specifically the amount active warps relative to the maximum amount of active warps. Maximum active warps depends on several factors, such as hardware, threads per block, registers per thread, and the amount of shared memory.

During optimization, one step is to ensure high occupancy so that the hardware is used efficiently. However high occupancy do not always guarantee performance; this differs depending on algorithms, for example how well latencies can be hidden by utilizing more parallelism.

**Memory load patterns**

Instructions are executed one warp at a time. When threads go through its workload, instead of having every thread go through a sequential part of the data structure, by adding a stride, each warp of threads instead iterates the data sequentially to make sure that no unnecessary memory operations are issued.

As seen in top part of figure 2.2 were no stride is used, each warp of threads will need to issue several memory loads. This is because the data needed for each wrap spans over several memory loads, illustrated as same coloured arrows that does not fit in the indicated memory load. However, at the bottom a stride is used that ensures that each warp of threads have their memory accesses aligned.

By having a stride across threads each warp will read the memory sequentially, utilizing all data in each read. This severely improves performance especially if the algorithm is bound by memory bandwidth. Reading the data from global memory is a performance bottleneck. However this type of performance loss can be hidden with CUDA by introducing more parallelism in the form of more threads and blocks that hides latencies and by utilizing the cache.

### 2.3 Svenska Spel Games

In the following two sections the games Lotto and Bomben is described. These are the two games that algorithms are developed and optimized for in this report.

#### 2.3.1 Lotto

Lotto is a game where participants individually bet on seven unique numbers between one and 35, this is called a row. At a later stage a public drawing
occurs, and the participants get one correct guess for every drawn number which is included in their row [1].

The participants can place their bets using two different approaches; either submit each row individually or create a system.

A system is a method for a participant to play multiple rows with a shortened writing method [1]. The player still chose unique numbers between one and 35 but the difference is that they can pick from eight to twelve numbers instead of just seven. The rows which they then play are all the possible unique seven-combinations of these numbers. These systems and rows are stored in a special way to greatly benefit from a popcount operation, more is discussed on this in section 5.1.

The price for betting on a row is always the same, no matter if you play on single rows or create a system. 45% of the total pot is distributed to the people who win. After the winning numbers have been drawn an additional four numbers are drawn as bonus numbers, where the original numbers can not be re-drawn.

There are five different winning groups; where the first four groups corresponds to have four, five, six or seven numbers correct from the first seven numbers. The last, and fifth group, corresponds to the case where a participant have six correct numbers from the original seven and one of the bonus numbers correct.

After all bets have been made and registered, all of these will then be processed for evaluation. Each bet is checked if it matches enough of the drawn numbers to placed in any of the winning groups. When all bets have been checked the pot can be distributed.

The challenge of implementing this is not necessarily the algorithm, it is the optimizations based on the problem at hand, including effective usage of threads, caches, data transfers and more. Additionally, one aspect is that the data need to be structured in a manner for the algorithm to run efficiently.

Figure 2.2. Illustrates how threads request memory within a warp.
2.3.2 Bomben

ODDSET is a collection name for different sports competitions at Svenska Spel and Bomben falls under this category [2]. Bomben is a game where participants typically bet on two, three or four selected matches in one type of sport, such as football or hockey, and guess how many goals each team strike for each match. The bet on how many goals one team score is called a cross, two crosses makes a match and a bet with two, three or four matches is a row.

Unlike Lotto, Bomben does not have different winning groups, either the participant guess all matches correct and wins or it is a loss.

Bomben can be played either as single rows or a system. This works like Lotto, where a system is a way to play multiple bets in a shortened writing method. A system for Bomben is to bet on a range of goals for each match and single bets simply picks an exact score.

<table>
<thead>
<tr>
<th>Permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1 1-0 : 1-1 : 3-0</td>
</tr>
<tr>
<td>Row 2 2-0 : 1-1 : 3-0</td>
</tr>
<tr>
<td>Row 3 1-0 : 1-1 : 3-1</td>
</tr>
<tr>
<td>Row 4 2-0 : 1-1 : 3-1</td>
</tr>
</tbody>
</table>

Table 2.1. Permutations of the row; [1,2]-[0] : 1-1 : [3]-[0,1].

For instance, a single bet for Bomben with three matches can be; 0-1 : 1-1 : 3-0, where those exact scores for every specific match needs to happen for it to be a correct row. Additionally, a system can be specified as a reduced bet. This allows the participant to decide to not include permutations where the home team win, away team win or results including any ties.

A system is played with the structure; [1,2]-[0] : 1-1 : [3]-[0,1], which represent all permutations of the goals included and places them as single row bets. Table 2.1 display all permutations generated from this specific row.

However, for this report a modified version of Bomben is used, which have been planned to possibly be released in the future. This version consist of betting on five matches, instead of maximum four, and it introduces winning groups. This mean that participants do not have to guess the exact row, they win different amounts if they have three, four or five matches correct.

The challenges of implementing this algorithm is the same as for Lotto. It is the utilization of threads, caches, data transfers and such. Also, it is important to know how to structure the data for the algorithm to run as efficient as possible.
3. Problem and Contributions

A large scale gambling system like the one Svenska Spel AB runs with millions of users that relies on a back-end with high performance. Several of their sports- and number-based games require new odds, winners and statistics to be calculated and presented to their users.

For example, when a team scores in a game new odds need to be calculated within seconds. The updated odds are broadcasted to different media such as TV, the web and text TV. All of this so that the people who participating can get the correct information as fast as possible. While for lotto the system calculates all possible winners during each stage of the drawing process after all balls have been drawn to be presented alongside the replay during the TV announcement. This project focus on calculating the winners qualified for different categories to be able to determine the distribution of the game pot. First a normal approach is taken that have the correct outcome available, this needs to be able to run within seconds to present the results to the users. The second approach can be calculated during the time that bets are locked but before the actual game starts. This approach will calculate all possible outcomes for the game giving personalized statistics and the means for new services to for example evaluate more profitable bets. This time frame can vary in practical usage but is assumed to be at least one hour for this project.

A baseline data file with the size of a real game instance for both Lotto and Bomben will be used, these contain around 6,500,000 individual bets for Lotto and around 250,000 for Bomben. The files are also multiplied to simulate larger data sets to ensure performance for future expansion.

3.1 Objectives

The focus of the project is to evaluate and minimize the time it takes after an event occurs, e.g. a goal being scored or a ball being drawn, to when this result can be presented by the system to its users. Implementations that show potential are also implemented in an extended manner, meaning that they will run the same algorithm multiple times on the same data to produce more results.

Winner selection algorithms for different games at Svenska Spel AB are implemented. The different algorithms are:

- Baseline serial
• OpenMP parallel
• Pthreads parallel
• CUDA GPU parallel

Additionally, there are a number of implementations with each approach which operates on different levels of preprocessed input data, to see how the calculation time is affected. Further reading about OpenMP, Pthreads and CUDA can be found under section 2, "Background".

The different implementations are then compared to each other to see which has the best performance. It is also interesting to see which are the limiting factors of different implementations.

3.2 Hardware

The hardware used was provided by Svenska Spel. It is a machine with a NVIDIA GTX 1060 card, which has 1280 NVIDIA CUDA cores and a memory bandwidth of 192GB/s. Technical specifications for the relevant hardware are listed in the table below.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i5 7500</td>
<td>4 cores @ 3.4GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia GTX 1060</td>
<td>GDDR5 6GB @ 192GB/s</td>
</tr>
<tr>
<td>RAM</td>
<td>Corsair Vengeance</td>
<td>DDR4 (2x4GB) @ 42.6GB/s</td>
</tr>
</tbody>
</table>
4. Method

4.1 Roofline Model

When designing applications for multicore, manycore, or accelerator processor architectures it may be difficult to realize where optimizations can be made to improve performance. The *Roofline Model* is a visually intuitive performance model which presents its results in a graph format. The model is best used to assess the quality of attained performance by combining locality, bandwidth, and different parallelization paradigms. The figure which is produced by the model can be studied to understand both implementation and inherent performance limitations. Additionally, from the model it is possible to read what priority different optimizations should have [20].

![Roofline Model Graph](image)

*Figure 4.1. Simple illustration of a general roofline model that shows what the ceiling and slope indicates.*

Figure 4.1 illustrates the most simple example of a roofline graph. The x-axis represents the operational intensity, which is the ratio of floating-point operations to data movement in bytes, and the y-axis represents the performance, which is measured as operations per time unit. Additionally, it is
important to acknowledge that both the y-axis and x-axis is represented in a logarithmic scale. The left roofline, the orange slope, is given by the system peak memory bandwidth. To be able to draw this line one can look at the vendor specifications of the hardware. The right roofline, the blue horizontal line, is the peak floating-point performance when the optimum system peak memory bandwidth is reached [14].

The two rooflines that are illustrated in figure 4.1 is the maximum bandwidth and optimal computation. Later, to be more precise on what performance can be realistically achieved, more rooflines can be added. These new rooflines would show for example how prefetching impacts performance and will always be below the blue and orange axis. However, in this project only the simple version is used and will at a later stage help evaluate the compute and bandwidth utilization of the implementations.

4.2 Data Structures and Preprocessing

The data provided by Svenska Spel is authentic data from earlier iterations of different games but with all personal and transaction data removed. It is structured in a manner that runs well for sequential execution. However the data can not be divided unless gone through sequentially since wagers vary in size because they can contain a varying number of rows. Besides preprocessing the data to be able to divide it efficiently is has also been slightly modified to accommodate additional requirements unique to CUDA. This is applied for all implementations regardless if it is needed to reduce the amount of work.

```c
typedef struct
{
    uint32      totalen;
    uint64      serial;
    uint32      amount;
    uint32      num_boards;
    uint16      boardsize;
    XBET_FORMAT_PARAMS format;
    uint32      boarddata_size;
    uint8       boarddata[1];
} XWAGER;
```

*Figure 4.2. Structure that holds all data associated with a wager.*

When all bets are placed for a certain game the information about every individual bet is stored in a structure displayed in figure 4.2 and 4.3. Hundred of thousands of these structures are written to a file where they are stored without padding between them.
The structure in figure 4.2 is a general template for all games at Svenska Spel. Depending on the field format in 4.2, which contains the information displayed in 4.3, the data can be interpreted in various manners.

```c
typedef struct
{
    uint16  board_flags;
    uint8   f1_type;
    uint8   f1_itemsize;
    uint8   f1_numitems;
    uint8   f2_type;
    uint8   f2_itemsize;
    uint8   f2_numitems;
} XBET_FORMAT_PARAMS;
```

*Figure 4.3.* Structure that contains all format parameters that decides which game a wager originates from.

In figure 4.3 all formatting parameters are shown that decides which kind of game the wager originates from and how the boarddata should be interpreted. The value of board_flags includes different properties about the boarddata, however, this information is not relevant for this project. So, board_flags determines how many bytes at the start of the boarddata that are not in use. Additionally, the fields f1_type, f1_itemsize, f1_numitems, f2_type, f2_itemsize, and f2_numitems indicates how the data should be interpreted. For instance, at different scenarios the data can be stored in chunks of two bytes, nibbles, or single bytes.

Specific information about these flags can be found under section 5 and 6, as this is depending on which game the wager originates from.

The field boarddata stores the specific bets that the current wagers contains. The size of this field is not static, because every wager can contain a range of boards. That is why the field boardsize is needed, to know how many bytes should be read when retrieving the data.

Since the size of a wager is not always the same it is not possible find a specific wager out of order. To iterate the data the field totlen is used which stores the total length of the wager. Figure 4.4 illustrates how totlen is utilized. The first four bytes of a wager always stores totlen, which is illustrated by the green field, and the end of the wager stores the boarddata, illustrated by the blue field. By reading the value stored in the first four bytes the information where the current wager ends and where the next one starts is known. Then by simply repeating this makes it possible to iterate through all wagers, until the end-of-file is found.

Additionally, serial contains a unique serial number for each wager to make it possible to identify them. The field amount stores how much money the
participant who placed the bet went in with. \textit{Num
d_boards} stores how many boards the wager contains in the boarddata and the \textit{boardsize} is the size of each board. The size of the board depends on how many bytes it is possible to store the information about a bet, which is dependent on the game.

As mentioned earlier, this structure is not optimized for parallel execution. The main reason behind this is because it is not possible to find a specific wager, out of order, without iterating them in sequence, as they are varying in size. As a result of this it is impossible to divide the wagers in evenly distributed chunks which is needed to be able to process them in parallel efficiently.

The general idea is that the data structure needs to be modified to make it possible to divide work. This can be achieved by either creating pointers to locations in the data, or reconstructing the data to fixed size entries. However, this preprocessing is made differently depending on the algorithm. You can read more about these specific preprocessing methods under the associated algorithm in chapter 5 and 6.
5. Implementation - Lotto

This chapter contains information about implementations, data structures and preprocessing for the data originated from the game Lotto. The data consist of roughly 1 000 000 wagers which contain 6 500 000 rows. A description about lotto can be found in section 2.6.

5.1 Data structure

The original data is structured in a generic format that can handle bets of several different games, explained in section 4.1. The specific parts for Lotto are the following:

The structure shown in 4.3 contains several fields but the only one relevant for Lotto is `board_flags`. This field indicates how many bits in the beginning of the boarddata that should be ignored. The reason behind this is that there are various methods for a user to submit their rows, and the data is packed differently depending on the input method. Other than that, all wagers are stored in the same way, which means that there is no use for the other flags.

The field `num_boards` in 4.2 represents, in this case, the number of rows a wager contains. If the wager consists of a system `num_boards` is always one, as a system is stored as a single row that consists of more than seven numbers. This is referred to as a `system row`.

![Diagram of Lotto row storage](image)

*Figure 5.1.* Illustrates how a Lotto row is stored with the help of 36 bits.

Figure 5.1 illustrates on a bit level how a row of numbers are stored in the boarddata. As shown in the figure, 36 bits are utilized (with the zero not being used) to represent the numbers one to 35. The position of the bit corresponds
to the number it represents. The example row in 5.1 have the bits on positions 3, 4, 11, 15, 19, 21, and 32 set to one which represents a row with these seven numbers. With this structure it means that five bytes are required to represent a row.

5.2 Preprocessing

The largest goal behind preprocessing the data is to make it possible to divide the work amongst several executing cores and threads. However, as the interest for this project is optimization for winning algorithms there will be three different cases of preprocessing where some changes the data more drastically to make the calculations even more optimized. These methods are explained in the upcoming sections. All three different versions of preprocessed data are used for time comparisons in the result.

5.2.1 Offset Array

To make it possible to distribute the wagers, a new array is created during the preprocessing of the data. When iterating the data an offset represented as an integer is stored for every wager in the array. The offsets originates from the field `totlen` of the wagers, and every cell of the array stores the accumulated value of all sizes of the wagers before. This means that at the end, the array stores an offset for every individual wager. By using this offset array when iterating the wagers it is possible to split the work for the cores and threads to work on in parallel.

```
typedef struct{
    uint64_t* wagers;
    void* data;
    int totalWagers;
    uint64_t size;
} PREPROCESS;
```

*Figure 5.2.* The resulting structure from the preprocessing when making the work distributable.

The resulting structure is shown in figure 5.2. It contains a pointers to where the array of offsets and the data are located in the memory, which is the `wagers` and `data` field respectively. The field `totalWagers` is the number of wagers the data contains and `size` is the size of the allocated memory chunk which the data is stored within.

The offset array add eight byte of overhead for each wager in the data, and since there are roughly 1 000 000 wagers in the data it result in an array of
eight MB. The whole dataset with the offset array is 105MB, which makes the offset array to consist of 7.6% of the total size.

Additionally, the wagers are converted to an aligned representations of the original data, this structure is illustrated in figure 5.3. The different fields still contain the same information as described in section 4.2 and illustrated in figure 4.1, but converted into the size of uint64_t. This is done to ensure alignment of the data, which is especially important for the GPU. The data is aligned to 64bits to make sure that it is possible to read the boarddata in chunks of 64bits.

An issue with this approach is that it does not utilize all the bits in the new structure. It was naively applied to ensure that no alignment issue could possible occur on the GPU, and was never adjusted during the project. However, this is only relevant for the offset array implementation.

Also, in section 4.1 an offset named board_flags is explained which is needed to access the relevant information from the boarddata. While preprocessing the data, this offset is removed for all rows of every wager. This is also done to avoid unaligned accesses in the memory.

```c
typedef struct
{
    uint64  tolen;
    uint64  serial;
    uint64  amount;
    uint64  num_boards;
    uint64  boardsize;
    uint64  boarddata_size;
    uint64  boarddata[1];
} ALIGNED_XWAGER;
```

Figure 5.3. The structure which an aligned_xwager consists of.

This approach minimally alter the original data with the benefit of making it possible to find a specific wager without the need of iterating them in sequence.

5.2.2 Single Rows with Systems

While iterating through the wagers only the relevant information is extracted, which is the the rows of each wager. The rows are individually stored in a new array of the type uint64_t, which results in a cell size that are large enough to store the bit-representation explained in section 5.1. Since rows and system rows are stored on the same format, which is illustrated in 5.1, they can be handled equally.

The resulting data have a completely different structure compared the original. The benefit is that the rows are now stored uniformly in an array, which makes it possible to use indexing to split the work.
However, the data still contain both system rows and regular rows, which cause the algorithms to be affected by from load-imbalance, and also divergence for CUDA. This brings the next preprocessing approach which is covered in the upcoming section.

The final structure returned by the preprocessing is an array of type uint64_t.

### 5.2.3 Single Rows

With this approach all system rows are removed and stored as its single rows representation.

Regular rows already have the correct representation in the original data, they are simply stored in a new array. However, system rows needs to be transformed to all regular 7-number rows which they represent. All of the rows it represent are generated and stored individually in the array as regular rows. By preprocessing the data to this state, issues with load-imbalance are removed since all the rows require equal amount of work from the algorithm. Also, since the algorithm does not have handle both system rows and regular rows the divergence for CUDA is significantly reduced.

However, one drawback with this approach of preprocessing is that when generating the rows which a system represents, the resulting data contain more rows. For instance, a system of twelve numbers represents 792 single rows that now are stored individually instead. However, the data consists of 5 500 000 entries and 80 000 of these are systems, which represents approximately 1 000 000 additional rows. By storing these systems as their single rows increases the input file size from 45MB to 52MB.

### 5.3 Sequential Implementation

The sequential implementation is made to understand the algorithm and to have an implementation to compare the parallel implementations against. Additionally, the actual logic for the winner selection algorithm is reusable for later parts. Different implementations which utilizes data with various structures are represented in the upcoming sections.

### 5.3.1 Baseline

This implementation iterates through all 1 000 000 wagers which contains a total of 5 500 000 rows, one at a time, from the original unprocessed data described in section 5.1. The data is traversed by interpreting the meta data and finding the actual boarddata, as illustrated in figure 4.3. For every wager the program retrieves the boarddata and compares that to the drawn row.

To determine the winning groups the algorithm starts by calculating the system size of the wager. This is done by calculating the Hamming Weight
with popcount, explained in section 2.4, of the row. If the Hamming Weight is seven it treats it as a regular wager.

To decide which winning group a regular row belongs to the program make use of logic bitwise &. The bit-representation of the current row and the drawn row is compared to each other with the &-operator, which returns a new bit-representation of the numbers that are represented in both rows. Then, by calculating the Hamming Weight of the resulting bit-representation the amount of correct numbers are known. However, if the Hamming Weight is six an additional check needs to occur for the 6+1 possibility, which is six correct guesses with one correct bonus number. Determination of the amount of correct bonus numbers are achieved in the same manner as for regular numbers. Bitwise & between the bit-representation of drawn bonus numbers and row is calculated and the the Hamming Weight of this result is the correct guesses. This procedure is repeated for the number of boards which the wager contains.

However, if the system size is larger than seven it needs to be treated as a system. A system represents several rows, which are all seven-number permutations of the system numbers. In this case, the calculation of the additions to the winning groups is more complicated. For instance, imagine the scenario where a wager contains a row with a system of size ten. The number of permutations from this system is calculated by:

\[
\binom{10}{7} = 120
\]  

(5.1)

Expression 5.1 is the mathematical way to determine how many unique combinations it possible to take seven (without re-picking an element) elements from a quantity of ten, where the order disregarded. Where ten and seven is the system and row size respectively. So, a wager with a system size of ten represents 120 rows. Now, if this system contains six correct guesses it will not only add one to the corresponding winning group but instead it also adds to every group with less correct guesses. The reason behind is that there are also permutations of the system which contains five and four of the drawn numbers. To determine how many permutations which contains five and four correct the following equation is used:

\[
\binom{\text{systemSize} - \text{totalCorrect} \text{wrongNumbers}}{\text{totalCorrect} \text{correctNumbers}} \times \binom{\text{totalCorrect} \text{correctNumbers}}{\text{correctNumbers}}
\]  

(5.2)

Where systemSize is the size of the system, totalCorrect is the total amount of correct guesses for the system, correctNumbers is the current winning group to be calculated, and wrongNumbers is the incorrect numbers in the row (which is 7 - correctNumbers). The first binomial of the formula calculates how many combinations there are to chose the incorrect numbers and the second binomial
calculates the same for the winning numbers. The product from the represents the total amount of combinations.

Therefore, to calculate the permutations for four and five correct guesses with the case of a system size of ten and six total correct guesses, the formula yields:

\[
\binom{10 - 6}{2} \times \binom{6}{5} = 36 \quad (5.3)
\]

\[
\binom{10 - 6}{3} \times \binom{6}{4} = 60 \quad (5.4)
\]

Where expressions 5.3 and 5.4 are for five respectively four correct. Therefore, in the case for system size ten and six correct numbers the row wins six correct one time, five correct 36 times and four correct 60 times.

However, the case of six corrects, just as for regular rows, needs some special handling because of the 6+1 group. In this case both the group of 6 correct and 6+1 is calculated in sequence. First, the number of rows with six correct is calculated, with formula 5.2. Within these permutations there are a quantity of rows that belong to the 6+1 group, since some of these rows also have a correct bonus number. The number of rows with correct bonus numbers are determined by the following formula:

\[
\binom{totalCorrect}{6} \times correctBonus \quad (5.5)
\]

Where correctBonus is the number of correct bonus numbers and totalCorrect is the amount of correct regular numbers. So, imagine the scenario to also have two correct bonus numbers. With the correct values in the formulas, it gives the result:

\[
\binom{10 - 6}{1} \times \binom{6}{6} = 4 \quad (5.6)
\]

\[
\binom{6}{6} \times 2 = 2 \quad (5.7)
\]

Where 5.6 and 5.7 are six and 6+1 correct respectively. This means that two permutations of the rows fall into the 6 correct group, because the other two lands in the 6+1 group. To calculate all of the winning groups for a system the program is structured like:
while (correct > 3) {
  if (correct == 6) {
    /* Special Case of Six Correct */
    correct --;
  } else {
    /* Handle All Other Groups */
    correct --;
  }
}

By looping as long as correct is larger than three covers all winning groups. By combining every aspect which have been included in this section, the final structure of the program looks like:

for (int k=0; k<totalNumberOfWagers; k++) {
  for (int i=0; i<numberOfBoardsInWager; i++) {
    if (systemSize == 7) {
      /* Handle Regular Rows */
    } else {
      /* Handle Systems */
    }
  }
}

Where the handling of rows and system also have the additional logic of taking care of 6+1 correct guesses.

The end result is five integers which represents the winning groups of 4, 5, 6, 6+1, and 7 correct guesses. Each time the program finds a row with a Hamming Weight of four or more it increments the corresponding winning group by one each time. At the end of the execution the integers contains the accumulated winners from all winning rows.

5.3.2 Single Rows with Systems

The data presented in section 5.2.2 is the input for this implementation.

This implementation works in the same manner as the one described in section 5.3.1, with the difference that it iterates rows instead of wagers. When the system size is larger than seven, which is calculated by using popcount on the bit-representation of the row, it is a system. The program makes the calculations described in section 5.3.1 to determine the correct values to be added to the winning groups. Additionally, the case of six correct guesses is still a special case because of the 6+1 winning group. This is handled by checking the bonus numbers just as described in section 5.3.1.

Put this together the rough structure of the program looks like:
for (int k = 0; k < numberOfRows; k++) {
    if (systemSize == 7) {
        /* Handle Regular Rows */
    } else {
        /* Handle Systems */
    }
}

Where numberOfRows is the total number of rows in the data and systemSize is the Hamming Weight of the row.

5.3.3 Single Rows

In this section the implementation which utilizes the data described in section 5.2.3 is presented.

The logic in the program that calculates which rows are winners and where they are stored works in the same manner as described in section 5.3.1. The differences lies within how they are iterated, and also a portion of the logic are no longer necessary.

Since the data now consists of an uniformly dataset it is possible to iterate the data with a simple for-loop and use the loop variable to index into the data directly. By using the loop variable to retrieve rows from the data array the bit-representation is found each time. To calculate if the row have any correct guesses the logic bitwise & is used, just as described in section 5.3.1.

However, the logic to handle system rows are no longer needed as they are preprocessed to their single row representation during the preprocessing phase. This eliminates a large chunk of the logic inside the for-loop that handle systems. So, compared to the final structure illustrated in section 5.3.1 it simplifies to:

for (int k = 0; k < numberOfRows; k++) {
    /* Handle Rows */
}

Where numberOfRows is the total number of rows in the data. As shown in this small code snippet, the for-loop which iterated the wagers boards in section 5.3.1 is also removed since all boards are converted to single rows.

5.4 Parallel Implementations

To utilize the preprocessed data structures explained in section 5.2 implementations of algorithms which handles several wagers in parallel are developed. Implementations with OpenMP, Pthreads and CUDA are described in the following sections.
5.4.1 OpenMP

As described in section 2.1, OpenMP makes it possible to modify a program that is running with sequential execution to utilize parallel execution with a few compiler directives.

**Offset Array**

As base for this implementation it is convenient to use the implementation for the sequential case, described in 5.3. However, this implementation uses the preprocessed data described in section 5.2.1 instead of the unprocessed which is used for the baseline implementation.

So, to access the each wager the offset array is utilized. It contains a byte offset indicating where the wagers are located in memory relative to the start of the data pointer. The wagers are retrieved in the following manner:

\[
\text{wager} = (\text{ALIGNED\_XWAGER} \ast) (((\text{char} \ast) \text{pre} \rightarrow \text{data} + (\text{pre} \rightarrow \text{wagers})[i]);
\]

This line works by first reading the address of \text{pre-data}, which is a pointer to where the data starts in the memory. By casting this as a character pointer it is possible to add byte steps to the address with a simple addition. Therefore, by adding the offset contained in \text{pre-wagers} the starting address of the \text{i:th} wager is retrieved, where \text{i} is the loop variable. Casting this address as an \text{ALIGNED\_XWAGER} makes sure that a whole wager is stored in the \text{wager} variable.

The most significant difference between the two implementations is that wagers are now processed in parallel. For the sequential implementation the for-loop that iterates all wagers is implemented like the following example code:

\[
\text{for (int i=0; i<pre->totalWagers; i++)}{
  \text{/* Handle Each Wager Sequentially */}
}\]

Where \text{pre->totalWagers} is returned from the preprocessing phase. For this code to work for OpenMP an additional row is added before the for-loop with a compiler directive, which looks like:

\[
\text{#pragma omp parallel for private (variables)}
\text{for (int i=0; i<pre->totalWagers; i++)}{
  \text{/* Handle Wagers in Parallel */}
}\]

This compiler directive is specifically made to parallelize for-loops. It divides the for-loop boundaries, which are from 0 to \text{pre->totalWagers}, evenly among the active threads to work on. Additionally, OpenMP needs to know which
variables are thread specific to prevent them from alternating each others values. This is achieved via private(variables) where variables is the collection of variables that are private for each thread.

The variables which are not set as private are shared among the threads. For this application there is only one array that falls under this category and that is where the number of winners are stored. The issue behind this is that several threads are storing values to this array without regard to if it is already being accessed, which leads to data races.

To solve this issue OpenMP presents a compiler directive which is used in the following manner:

```
#pragma omp atomic
winners[index]++;
```

The atomic directive makes the operation on the next line in the code to be executed atomically [4]. An atomic operation ensures that no other thread can read or write from the memory location until the operation is complete, which removes the risk of any data race. Therefore, by securing every access to a cell of the array with an atomic directive the program yields the correct results.

### Single Rows and Systems

This is the implementation for OpenMP which utilizes the preprocessed data described in section 5.2.2. It uses the same logic as described previously.

The same for-loop with associated compiler directive shown in section 5.4.1 is also implemented for this version. The difference is that instead of using the number of wagers as the loop variable it now looks at the number of rows, which is returned from the preprocessing phase. Also, as the data now consists of uniform data it is possible to access it directly with the help of the loop-variable, instead of using the offset array.

### Single Rows

For this implementation the data containing single rows, explained in 5.2.3, are used and this implementation builds upon what is described in section 5.3.3. Additionally, the logic in the program is simplified as there are no longer any system rows, only regular rows with a system size of seven.

This implementation still uses the atomic compiler directive explained in section 5.4.1 for the same reason.

### 5.4.2 Pthreads

In the upcoming sections different implementations with Pthreads are described. Compared to OpenMP Pthreads gives the programmer more control of how to divide work among threads.
Offset Array
Since Pthreads are on a lower level of abstraction more parts of the parallelization needs to be handled in code by the programmer. This means that the code and program flow seems to be more complex.

![Diagram](image)

*Figure 5.4. The general program flow of a program implemented with Pthreads.*

The program running with Pthreads follows the flow chart illustrated in figure 5.4. In the first step, the main thread loads preprocessed data into memory and sets up necessary variables.

In step two it creates all threads. For every thread a structure needs to be allocated which contains arguments for the thread to use when it executes. Figure 5.5 displays how this structure looks. Every thread is assigned an unique ID in the variable `id` and they all receive copies of the winning numbers and bonus numbers in the bit-representation illustrated in figure 5.1 as the variables `winNumsBit` and `winNumsBonusBit` respectively. The pointer `threadPre` points to the start of the preprocessed data in memory and `winners` is a pointer to the array where the threads store the results.

```c
typedef struct {
    int id;
    void *threadPre;
    uint64_t winNumsBit;
    uint64_t winNumsBonusBit;
    int *winners;
} THREADARGS;
```

*Figure 5.5. The structure of arguments which every thread needs to execute.*
The array which stores the winners is structured differently in this program compared to OpenMP. Since Pthreads gives the ability to assign unique IDs to the threads it is possible to create thread-private areas of an array. The approach is to make the winners array large enough to hold the calculated winners which each thread finds. The threads store their individual winners on an index which is derived from their own ID.

In step three in figure 5.4 the main thread stalls until it joins every thread which were spawned in step two. Finally, at step four the main thread iterates through the winners array to compile all the results from the different threads.

All logic for the actual winner selection algorithm is located on the pthreads for this implementation. The logic explained in paragraph three in section 5.3 is reused and it is made to run on every thread. Every thread have access to all preprocessed data in memory but each individual thread only works on a small piece of it. To achieve this the boundaries for the outer for-loop are the following:

```c
int start = id * threadWork;
int end = (id + 1) * threadWork;
for (int i = start; i < end; i++){
    /* Handle wagers */
}
```

In the same way as for the winners array the unique thread ID, named id, is utilized to calculate the thread specific loop boundaries. The variable threadWork represents a chunk of the data which every thread gets. When the data is not divisible with the number of threads, the last thread executes all the extra work. In the worst case the last thread get three extra rows, since the implementation run on a four core CPU.

Wagers are fetched from memory with the help of the offset array, explained in section 5.2.1. This is performed in the following manner:

```c
wager = (ALIGNED_XWAGER*)((char*)((PREPROCESS*)args)->threadPre + wagers[i]);
```

Where args->threadPre is the pointer to the data as shown in the structure illustrated in figure 5.5 and the array wagers contains the offsets. This works as the case for OpenMP described in section 5.4.1, where the address first is interpreted as a character pointer which allows to add single byte steps to the address when adding the offset. Then it is interpreted as an ALIGNED_XWAGER pointer to fetch all desired data.

**Single Rows and Systems**

This implementation uses the data with single rows and systems, described in section 5.2.2, which is similar to the earlier implementation with pthreads.
The program flow is the same as illustrated in figure 5.4, where the main thread launches threads, joins them and compiles the results from each of them. Instead of casting as character pointer as in section 5.4.2 the rows are retrieved with following line:

\[ \text{wager} = \text{rowArray}[i]; \]

Where \text{rowArray} is an global array which stores the data of rows and \( i \) is the loop variable. Additionally, the threads are now iterating the number of rows instead of the number of wagers.

It still uses the same logic as described in the previous section where popcount is used to decide if a row is a system row or regular row. Then if it is a system, it calculates all the permutations of different rows to add to correct winning groups using the binomial coefficient. This process is thoroughly described in section 5.3.1.

\textbf{Single Rows}

This is the Pthreads implementation which uses the preprocessed data described in section 5.2.3.

The workflow is still as illustrated in figure 5.4. However, the difference is the code which each thread executes. Rows are now retrieved in the simple manner of indexing in the array which stores them uniformly, just as in section 5.4.2. The difference is that the partition of the implementation that handle system rows are no longer needed, since those are no longer present in the data.

\textbf{5.4.3 CUDA}

In the following sections CUDA is used to utilize the GPU when implementing the algorithms with different input data. Information about CUDA can be found in section 2.3.

\textbf{Workflow}

The core difference in an implementation using CUDA is that the \textit{host}, which is the CPU, needs to communicate with the \textit{device}, the GPU.

Functions are defined as either \textit{host}, \textit{global} or \textit{device}, this determines how the function call is executed. A host function is exclusive for the host while a device function is exclusive for the device, a global function can be called from either (typically from the host) but executes on the device. Functions are assumed to be declared for the host by the compiler if nothing is specified.

A CUDA program typically needs to follow certain steps to allocate memory and move data between the device and host. Figure 5.6 illustrates how this works for the implementations in the upcoming sections.
In step one, preprocessed data is loaded into the memory on the host and variables are initialized. A number of these initialized variables contains information about data sizes derived from the preprocessed data. Step two consists of memory allocation on the device, which utilizes the variables with data sizes from the previous step. The preprocessed data is transferred to the device in step three. In step four the host issues a function call for the kernel to run on the device, which launch the threads on the device. All threads calculate their partial results individually and then compile them to one mutual memory location. Then, in step five the host waits until all threads are done with the iteration of the data before it starts with the next step. Step six simply fetches the compiled results from the device to the host and the final step, step seven, the host free all allocations that exist on the device and host.

Offset Array

The basic implementation requires minimal preprocessing, which is the data described in section 5.2.1. The data needs to be aligned and an index array with absolute offsets from the start of the data is created to access a given point in the data directly.

The first part of this section describes the steps performed by the host and then the implementation of device code is described.

First, the host needs to know how much memory to allocate on the device. These allocations are done with function `cudaMalloc`. The calls for this implementation looks like the following:
cudaMalloc(&d_winNumsBit, sizeof(uint64_t));
cudaMalloc(&d_winNumsBonusBit, sizeof(uint64_t));
cudaMalloc(&d_winners, sizeof(int)*5);
cudaMalloc(&d_wagers, sizeof(uint64_t)*totWagers);
cudaMalloc(&d_data, pre->size);
cudaMalloc(&d_totalWagers, sizeof(int));

There are some single variables which needs to be allocated on the host, like \(d_{\text{winNumsBit}}, d_{\text{winNumsBonusBit}}, \) and \(d_{\text{totalWagers}}\). These variables contains the bit-representation of the winning row and bonus numbers and the total number of wagers. \(d_{\text{winners}}\) stores the final number of winners, which are five integers. This array is allocated by taking the size of an integer and multiplying it by five. The offset array is allocated at the pointer \(d_{\text{wagers}}\). Every offset is stored as an uint64_t and one exist for every wager. It is stored as an uint64_t to be able to represents offsets of several GB, as the largest file for this project is around 2GB. The number of wagers are stored in the variable \(\text{totWagers}\) which originates from that preprocessing phase. Finally, the largest allocation is for the complete dataset. The size of the dataset is returned from the preprocessing, which is stored in \(\text{pre} \to \text{size}\) and the chunk is allocated at the pointer \(d_{\text{data}}\).

For copying the actual data to the device the CUDA library have a function called \(\text{cudaMemcpy}(dst, src, x, flag)\), which copies \(x\) bytes from src to dst. The fourth argument is a flag which indicates if the data is copied from host to host, host to device, device to device or device to host. The same way of calculating the size as in the previous code snippet is used and the destinations are the same pointers. The source pointer is where they are stored on the host.

With everything prepared the host issues the work to start on the device with the following line:

\[
\text{processCudaWager} \ll<\ll < B, T \rr>\rr\rr((d_{\text{totalWagers}} ,
\quad \text{d}_{\text{data}} ,
\quad \text{d}_{\text{wagers}} ,
\quad \text{d}_{\text{winNumsBit}} ,
\quad \text{d}_{\text{winNumsBonusBit}} ,
\quad \text{d}_{\text{winners}} ) ;
\]

Where \(B\) and \(T\) are the number of blocks and threads respectively. This decides how many blocks and threads that are used during the execution. Further on, \(\text{processCudaWager}\) is the function to run on the device and it is described later in this section.

The call \(\text{cudaDeviceSynchronize}()\) from the CUDA library stalls the host until all running kernels on the GPU have reached a synchronization point or terminated.
When all kernels are synchronized the host fetches the results from the device with cudaMemcpy, the same function which was used to copy data from the host to device. However, results are summarized by the threads on the device, which means that only one integer for each winning group needs to be copied from the device. The call looks like:

\[
\text{cudaMemcpy ( count , d_winners , sizeof ( int ) \times 5 , cudaMemcpyDeviceToHost );}
\]

This call copies the size of five integers from \textit{d_winners} to \textit{count}. The fourth argument indicates that the data is transferred from the device to the host. At this point the correct results are stored on the host.

The final step for the host is to free allocated addresses and this is conducted with the function \textit{cudaFree(p)} from the CUDA library, where \textit{p} is the pointer to the address to be freed.

This concludes the work for the host in this implementation, however it is the threads on the device that iterates the data and calculates the winning groups.

The logic to handle rows and system rows is the same as for the sequential baseline case described in section 5.3.1. The difference is that work needs to be distributed evenly among the threads. By knowing the total workload and the total amount of threads and blocks an individual workload can be calculated.

The change is that for this implementation each thread calculates its own ID in the following manner:

\[
\text{index} = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x};
\]

Where \textit{threadIdx.x} is the ID for the thread within the block, \textit{blockIdx.x} is the ID of the block and \textit{blockDim.x} is the number of threads in each block. By doing this calculation every thread have an ID which is unique among all threads.

By using each threads unique index they fetch data to work on with the offset array, in the same manner as described in 5.4.2.

While the threads are working, they store winning rows in an array which is declared in the shared memory scope. The shared memory is on-chip, which enables load and stores with low latency [11]. It is also block specific, therefore every block declares its own array for the threads executing within and this is done with the help of the \textit{__shared__} flag before the array declaration. Each thread store values in this array in cells which are not shared with any other thread, removing any risk of data races.

When all work is done each thread adds its result to the device winner count in global memory, and since all threads are writing to the same position an atomic add is used.
**Single Rows and Systems**

When going through the data and preprocessing it into single rows, wagers that consist of systems are expanded. In the offset array implementation systems are directly calculated as a whole, depending on the amount of correct numbers and the size of the system. A wager that contain a system of size 12 would take up 792 times more space than a single row. Other system sizes to row conversion can be seen in table 5.1.

**System expanded into single rows**

<table>
<thead>
<tr>
<th>System size</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>11</td>
<td>330</td>
</tr>
<tr>
<td>12</td>
<td>792</td>
</tr>
</tbody>
</table>

Table 5.1. *Table that display how many rows each system size represent.*

By preprocessing the data but not expanding systems into single rows and instead saving these just like the normal single rows a slightly more compact data structure is gained. This implementation goes through less rows but have increased complexity that causes some branch divergence and requires more registers per thread.

**Single Rows**

With preprocessed data to only consist of single rows, described in section 5.2.3, the implementation is simplified compared to the previous section. The overall structure of the program resembles what is described in section 5.4.3, but without the need for an offset array since all single rows are of the same size. Also, since the data ensures only single rows the functionality for handling system rows is removed.

Shared memory is used as before for the results. It is saved into global memory and copied to the host.

However, one large difference with this implementation is that a stride is used when accessing data, to maximize the efficiency of the memory transactions of the warps. Information about what warps are and how they work can be found in section 2.5. A stride of 32 is used to satisfy each warp, this will require the amount of threads to be divisible by 32. Some of the threads will exit earlier if the workload can not be perfectly divided between them.

Implementations with both the stride and naive memory accesses are tested in the result section.
5.5 Extended Implementation for All Rows

This is an extended version of the implementation described in section 5.4.3, with single and system rows.

The original algorithms uses one drawn row, which is the particular winning row, and checks every played row against it. The implementation described in this section does not only check all the played rows against one particular row, instead it generates all seven-number permutations of possible rows and uses each as a drawn row separately. The possible combinations of a seven-numbered row with digits between one and 35 without repetition are 6 724 520. This means that this algorithm does more than 6.7M times more work than the original algorithm. This implementation is referred to as the extended implementation.

The outcome of this is the distribution over the different winning groups for all possible rows. This could be utilized in different manners, for example, it would enable the possibility to sort the rows to see which one give the most dividend. This sorted list could also be used to provide premium rows to the customers, which are ensured to be alone in the highest winning group.

However, for this to be feasible the bonus numbers are not taken into account, since this reduces the possible combinations. This reduces the number of possible permutations with a factor of 20 475 and only removes the 6+1 winning group.

With minor alterations and by using grid dimensions, the original implementation can be adjusted into this extended version. The dimension of the grid is a way to decide how many blocks the execution uses. Threads and blocks are defined by a three-dimensional structure. But for instance, if the function is issued like:

\[
\text{function } <<<B, T>>>(\text{arg});
\]

It executes with \(B\) number of blocks and \(T\) number of threads with the other dimensions set to one, since only one value is specified. However, if the function call looks like the following:

\[
\text{function } <<<\text{dim3}(x, B, 1), T>>>(\text{arg});
\]

It adds another dimension the grid, which is the variable \(x\). For the program, this simply means that the program uses \(x*Blocks\) number of blocks instead of just \(Blocks\), but the notation, if used correctly, simplifies for the programmer.

For this case, the added dimension to the grid is exactly the number of possible combinations of seven-digit rows, which is 6724502 rows. By selecting this for the new dimension the original program does not have to be altered vastly, since the new dimension is used as an index to fetch drawn rows.

However, the first change is in the code for the host. An array which contain all of the 6.7M rows need to be allocated on the device. This is done by extended the previous cudaMalloc for the drawn row with a factor of 6.7M:
cudaMalloc(&d_winNumsBit, sizeof(uint64_t)*rowComb)

Where \(rowComb\) is the number of possible combinations. Also, the cudaMemcpy is changed in the same manner.

A new phase is added to the preprocessing phase, which generates all possible row combinations. This is done by nesting seven for-loops, where every individual loop handle one number of the row. The rows are generated by incrementing the numbers separately produce unique combinations.

Additionally the array to store the results need to be extended, since as all possible outcomes are checked they also needs to be stored. This is done by extending the size of the array with a factor of \(rowComb\).

The new device call now looks like:

```
processCudaWagerRow <<<dim3(rowComb,B,1),T>>>(args)
```

The work is now issued with \(dim3(rowComb,B,1)\) to increase the dimension of the grid, and \(args\) is the same arguments as in section 5.4.3.

The first change in the logic on the device is that each thread start with initializing a variable:

```
int x_index = blockIdx.x;
```

Where \(x_index\) is assigned to the value of \(blockIdx.x\), which is the dimension that the thread belongs to in the newly added grid dimension. This value ranges from zero to the number of possible combinations (minus one). The index is utilized in the following manner:

```
winRow = d_winNumsBit[x_index];
```

Where \(d_winNumsBit\) is the array which contain all the possible rows. By using \(x_index\) to fetch a specific row the program can operate as the original program described in section 5.4.3. \(x_index\) is also used to save the results in the result array.
6. Implementation - Bomben

This chapter describes the implementation of the game Bomben, which is described in section 2.7. Implementations of data structures, preprocessing, sequential and parallel algorithms are explained.

6.1 Data Structure

The data is structured following the format described in section 4.1. However, there are some specific aspects about Bomben that are not included.

The data structure for Bomben contain different ways of storing bets. This originates from the fact that users can place their bets in different ways such as; from the web interface, sending a file with rows, from a betting in a shop, etc.

In figure 4.1 the generic XWAGER structure is illustrated, some of these values are specific for Bomben. The field boardsize store how many matches which is included in the bet, for this report this is always five. Also, num_boards contain the number of rows the wager include. This field is not needed for every structure.

How the bet is stored is interpreted from the XBET_FORMAT_PARAMS structure shown in figure 4.2. The field board_flags is an indicator of how many bytes that must be skipped in the boarddata to locate the relevant information.

In the upcoming sections different format of storing bets for Bomben are described.

6.1.1 One Nibble per Cross

This structure is a compact way of storing regular rows but it does not support to store systems.

When f1_type is set to one and f2_type is zero in the format structure it is interpreted as the format of one nibble per cross.

For this structure every cross is stored on one nibble, which make a match occupy one byte in memory as illustrated in Figure 6.1. As one bet consist of five matches it occupy five bytes in memory.

The example match in figure 6.1 display how the data in each nibble is interpreted as a binary number. The example match shown in the figure represent one match where the team playing at their home arena score one point, and the other team score five.
6.1.2 Two Bytes per Match

This structure store the bet data in the same manner as described in section 6.1.1 and illustrated in figure 6.1. However, the difference is that this structure store an additional byte for each match that correspond to which of the five matches this bet belong to. This makes it possible to store a number of bets for each match.

For instance, this structure can store two different rows for a three-match bet in the following manner; \( <0-0: [1-0, 1-1]: 3-1> \). This represents the rows; \( <0-0: 1-0: 3-1> \) and \( <0-0: 1-1: 3-1> \). So, with this structure it is not possible to store systems where ranges of goals are defined for each match, but it can store combinations of results for the matches.

6.1.3 Two Bytes Bitmap per Cross

This approach is not compact for single rows, it is used to store systems.

When the \( f1\_type \) is set to one it is interpreted as the format of two bytes for each cross. When a bit is set within these two bytes, the corresponding index of the bit is the number of goals which the bet is put on, this is illustrated in figure 6.2. The example match display a system where different number of goals have been placed as the bet, a regular bet contain exactly one set bit for each cross.

This structure always has two bytes per cross, no matter how many bets has been placed for each match. Due to this, the variable \( \text{num\_boards} \) is no longer relevant.

The majority of wagers in the original dataset is stored with this structure.

6.1.4 Reduced Game

Reduced game is an extension of the structure described in the previous section 6.1.3. This structure has the flags \( f1\_type \) and \( f2\_type \) set to one.
Figure 6.2. Illustrates how bets are stored in memory for Bomben with two bytes bitmap for each cross.

At the end of the boarddata one byte for each match is stored which represents the reduction bits. Only three bits in the byte are utilized. When the least significant is set all outcomes where the home team win are included, if bit two is set, all ties are included and the third bit indicates to include all combinations when the away team win. If all three bits are set all combinations are included and no reduction is made, note that at least one bit must be set otherwise all outcomes are reduced.

For instance, figure 6.3 illustrates how this data is stored for a row with three matches. The first byte after the boarddata corresponds to the first match. For this case all three bits are set, this means that all combinations of a system are included. For the second match the third bit is not set. This means that no result where the away team win are used. For the last match, no permutation where the home team win are included.

Figure 6.3. Illustrates how the reduced structure is stored in memory.
6.2 Preprocessing

For Bomben the data is significantly altered during the preprocessing. While iterating through all wagers only the relevant information is extracted and stored on the format with one nibble for each cross, explained in section 6.1.1.

For each wager the meta data is read to understand in which format it is stored. Depending on the format the wager is stored as, the boarddata is interpreted and converted. Wagers containing systems are converted to all permutations which they represent and stored as single rows. Since all system rows are converted to the corresponding single rows in the preprocessed data and everything is follows the one nibble per cross format, complexity of the algorithms can be reduced.

```
typedef struct {
    uint64_t numberOfEntries;
    uint64_t numberOfMatches;
    uint8_t *result;
    uint64_t size;
} PREPROCESS;
```

*Figure 6.4. The final structure returned from the preprocessing phase for Bomben.*

The final structure returned from the preprocessing is shown in figure 6.4. Where the field `numberOfEntries` is how many entries there are in the `result` array. The result array store single matches and the variable `numberOfMatches` is the number of matches that each bet contain, which is always five for this case. The variable `size` is the size of the result array in number of bytes, which is necessary when allocating memory.

6.2.1 Data Extension

As mentioned in section 2.7, the original data is extended for this project. Instead of consisting of four matches one more match is added with fabricated data which results in a total of five matches for each row. This match will be set to 0-0 for all bets so that some results can be verified easily.

6.3 Sequential Implementation

The sequential implementation iterates all the rows in sequence and finds which ones contains more than three correct guesses.

The data is stored in the structure explained in 6.1.1, with one byte for each match. However, since it is only relevant if the bet on a match is exactly equal as the outcome it is possible to compare the byte-representations with each other.
The program creates an array with three cells to store the winning groups for three, four and five correct guesses. It compare each match from the current row with every match from the winning row while keeping track of how many correct and incorrect guesses that are found. This is done within a for-loop which loops numMatches. If more than two incorrect guesses are found the program continues to the next row, since it is no longer possible for the current row to place in any of the winning groups. When the inner loop is finished the corresponding cell of the winning groups array is incremented.

The rough structure of the program is shown in the following code snippet:

```c
int winGroups[3];

for (int i=0; i<numEntires; i=i+numMatches){
    for (int j=0; j<numMatches; j++){ /* Increment 'correct' if it is Correct */
    }
    winGroups[correct-3]++;
}
```

Where `numEntries` and `numMatches` are the `numberOfEntires` and `num-berOfMatches` from the structure in figure 6.4 respectively. Since the data is stored on a match level, the loop variable `i` need to increment `numMatches` each time to fetch whole rows, which consist of five matches. The index `correct-3` is a translation to store having three correct matches in cell zero, four correct in cell one and five in cell three. The inner loop, with `numMatches` as its limit, iterate the matches within a row. At the end of the execution the number of rows with three, four and five correct guesses are stored in the `winGroups` array.

### 6.4 Parallel Implementations

In the following sections three different parallel versions of the program are implemented with OpenMP, Pthreads and CUDA.

#### 6.4.1 OpenMP

As OpenMP modifies a sequential program to run in parallel with a few compiler directives, the implementation is similar to the previous described in section 6.3. The change to the main loop is the following:
The added compiler directive `pragma omp parallel` automatically parallelize for-loops. It divides the loop variable in different chunks, depending on the thread count. The program within it still looks the same as for the sequential version. The other change is how the program store the winners:

```c
if (correct == 3)
    #pragma omp atomic
    winGroups[correct - 3]++;
if (correct == 4)
    #pragma omp atomic
    winGroups[correct - 3]++;
if (correct == 5)
    #pragma omp atomic
    winGroups[correct - 3]++;
```

It now uses an atomic operation to store the winner, to avoid any data races. It is also split in to three different statements to avoid too many threads storing to the same location at the same time. However, since just a few percent of all the rows actually win, this do not visibly affect the performance.

### 6.4.2 Pthreads

The program flow is still the same as illustrated in figure 5.4 where one main thread creates a number of pthreads, stalls until all threads are joined and summarize the result.

Additionally, the main thread of this program has many similarities with the Pthreads implementation for Lotto, described in section 5.4.2. It allocates a structure to pass arguments to the threads when creating them, this structure is displayed in 6.5. Each thread gets an unique ID as the variable `id`, which is used to divide the work among them. The pointer `threadPre` is a pointer to the structure returned from the preprocessing phase, shown in figure 6.4, which contains the data with some additional variables. Every thread gets a copy of the winning bets, which is stored in the array `winNums[5]`. The winning row is stored with a granularity of one match, as this makes it easy to use later. Finally, `winners` is a pointer to an array where all thread stores their results, where each thread has thread-private areas.
The threads are created and starts their execution on a function called `processWager`. ProcessWager consist of the same logic explained in section 6.3 which now executes on threads in parallel.

However, some additional logic is added. The data needs to be distributed evenly among the threads to avoid load imbalance. The way this is done is similar to the method explained in section 5.4.2 with the use of the variables `threadWork`, `extraWork` and `laterStart`. These variables are used to assign unique indexes in the data array for the different threads to work on. ThreadWork is the largest chunk of data which can be distributed evenly without exceeding the boundaries. ExtraWork is to emit the rest data from the distribution of threadWork, and laterStart is an offset to compensate where to begin in the data array. ThreadWork is defined from integer division between the number of rows and threads. ExtraWork is one if the ID of the thread is lower than the rest from the integer division, else zero. This is explained further with an small example in section 5.4.2.

These variables are used to calculate the loop boundaries which gives each thread a piece of the data to work on. The rough structure of the function and how the boundaries are defined is the following:

```c
typedef struct {
    int id;
    void *threadPre;
    uint8_t winNums[5];
    int *winners;
} THREADARGS;
```

```c

Figure 6.5. The structure for thread arguments.

\[
\begin{align*}
\text{start} &= \text{id} \times \text{threadWork} + \text{laterStart} \\
\text{end} &= (\text{id} + 1) \times \text{threadWork} + \text{extraWork} + \text{laterStart} \\
\text{for (int i=start; i<end; i++)}{ \\
    \text{for (int j=0; j<numMatches; j++)}{ \\
        /* Handle Each Match */
    }
}\]
```

By using threadWork, laterStart, and extraWork in combination with the unique ID it is possible to define thread-individual chunks of the data array. Within the for-loops the program works in the same manner as explained in section 6.3. The only difference is storage of the results, which works like:
if (corrects == 3)  
    winners[id * 3]++;  
if (corrects == 4)  
    winners[id * 3 + 1]++;  
if (corrects == 5)  
    winners[id * 3 + 2]++;  

Where the unique ID is used to create thread-individual areas of the winners array to avoid any data-races. The ID is multiplied with three because of three existing winning groups.

At the end the main thread iterates the winners array to add all the results from the threads.

6.4.3 CUDA

The workflow of the program is the same as explained in section 5.4.3, where the host allocate memory and transfer data to the device, launch threads on the device and fetch the results.

The data is structured in a different manner for this implementation compared to what the Pthreads and sequential implementations used. Instead of storing the bets for each match the data array now consist of 64bit integers. By storing all five matches on 64bits, 24bits are wasted for each bet. The downside of this is that there is 37.5% more data to transfer to the GPU. However, the main focus of this project is to make the compute time as low as possible, as this have the most effect on the extended runs. The benefit from using 64bits is that each thread access the memory once to load all five matches, which is more efficient when the threads are executing in warps. The reason behind this is the topic of section 2.2.2.

The allocations on the device is made with the help of cudaMalloc, they are as following:

cudaMalloc (d_data, size);  
cudaMalloc (d_winners, sizeof(int) * 3);  
cudaMalloc (d_numberOfEntries, sizeof(uint64_t));

Where d_data is the pointer to the complete dataset on the device, this is by far the largest allocation of the three. Size is returned from the preprocessing phase and is simply the total size of all the data. The pointer d_winners is where all the threads store their final results on the device, which is the size of three integers. Lastly, d_numberOfEntries is the number of elements in d_data, this is needed to divide the work.

The function cudaMemcpy is used, just as explained in section 5.4.3, to transfer the data from the host to the device. The number of bytes is the same values used when allocating.
The winning row must be accessible for every thread on the device, it is only read, not written to. It is also accessed often since every thread use it multiple times. Because of this the row is placed in constant memory, which is limited but have a short latency. To do this, a function called `cudaMemcpyToSymbol` is used, which work like cudaMemcpy but copies the data to the constant memory instead of global memory. That call looks like:

\[
\text{cudaMemcpyToSymbol}( \text{d\_winNums}, \text{winRow}, \text{sizeof(uint8\_t)*5});
\]

Where `d\_winNums` is the variable to store the winning row, `winRow` is the location of the data on the host and `sizeof(uint8\_t)*5` is the size of the data.

At this stage the preparation is complete and the work can be issued to the device, and this is done via the following function call:

\[
\text{processCudaWager}<<<B, T>>>(\text{d\_data}, \text{d\_winners}, \text{d\_numberOfEntries});
\]

The variables `B` and `T` indicates how many blocks and threads, respectively, the execution uses on the device. Configurations for the values of `B` and `T` depends on the hardware and the implementation, these are investigated and explained in section 8.2.1. The arguments are the pointers to the allocated memory which was previously described. The function `processCudaWager` is the function which contains all the logic, and which executes on each thread.

The logic in the function `processCudaWager` works as explained in section 6.3, with double for-loops iterating the wagers and the number of matches within each wager.

While iterating the data each thread store their result in an array which is declared in the shared memory with the help of the `__shared__` flag. This array is block specific and therefore one is declared for each block. Each thread within a block have an individual index-range of the array, which avoid all data-races. The accesses to the shared array looks like:

\[
\text{if}(\text{corrects} == 3) \quad \text{winners}[(\text{index}\times3)\mod(\text{blockDim.x}\times3)]++;
\]
\[
\text{if}(\text{corrects} == 4) \quad \text{winners}[(\text{index}\times3)\mod(\text{blockDim.x}\times3) + 1]++;
\]
\[
\text{if}(\text{corrects} == 5) \quad \text{winners}[(\text{index}\times3)\mod(\text{blockDim.x}\times3) + 2]++;
\]

Where `index` is the globally unique index for the thread, calculated as described in section 5.4.3. The number of threads which executes within each block is fetched from `blockDim.x`. The factor of three exists because of the three winning groups and the modulo with `blockDim.x` convert the globally unique index to a block unique index.
The distribution of work is the same as for the implementation for Pthreads as explained in the previous section 6.4.2. Where the globally unique index is used to calculate thread-individual boundaries in the data array for each thread. The calculations are explained in section 5.4.2 with a small example.

The final step of the execution is reduction of the results, each thread use an atomic add to store their results to a common result array, which then is transferred to the host.

6.5 Extended Implementation for Every Bet

The following sections will go through extended implementations for the different implementations. These extended implementations iterates every unique played row as it is the winning row, calculating the winning groups for every case.

6.5.1 Finding all Played Combinations

For this implementation some additional preprocessing is required. After all bets are preprocessed and converted to the representation described in section 6.1.1 they are used to find all unique rows. This processes is needed because the number of possible rows is too large to be realistically used.

For example, the dataset which is used for this project contain five matches. On every match the participant bet two crosses, which can be a number between zero and nine, and an additional option which indicates more than nine goals. This means that the total number of unique combinations are eleven to the power of ten, which is roughly 26 000 000 000 combinations. Three integers are used to store the results from each of these combinations, which is approximately 300GB of data. This data is needed for every instance of a Bomben game. This exceeds the possible limits for this project.

Additionally, most of these combinations are unlikely to occur as an outcome of a match, since a team in the different kind of sports included in Bomben rarely score more than 4 goals in a single match.

The number of unique rows in the data is approximately 58 000, which is only 0.0002% of all possible combinations.

To find all played rows a simple algorithm is developed. First the program takes the initial row from the data array and copies it to an array where all unique rows are stored. Then, rows are fetched sequentially from the data array and compared with every row in the unique rows array. If no identical row is found in the unique rows array the current row is added to it. For each found unique row a variable is incremented which represents the total number of unique rows. This is repeated until all rows from the data array has been checked. At the end of the execution the structure displayed in figure 6.6 is...
typedef struct {
    uint8_t *uRows;
    uint64_t nRows;
} REDUCED;

Figure 6.6. The structure returned from the unique rows calculation.

returned. Where \textit{uRows} is a pointer to the array containing all unique rows and \textit{nRows} is the total number of unique rows in \textit{uRows}.

6.5.2 Sequential
In this section the implementation explained in section 6.3 is extended to handle all played combinations.

The logic in the program remains the same, the change is that an additional loop is added to iterate all unique rows. The following code snippet roughly explains the change:

```c
for (int k = 0; k < nRows; k++) {
    winRow = &uRows[k*5];
    /* Original Program */
```

In addition the the original program, a for-loop is added around it which loops on the total number of unique rows. In this loop different winning rows are fetched from the \textit{uRows} array. This is done by using the loop-variable \textit{k} to index into the array and use the variable \textit{winRow}. Since the rows are stored on a match-granularity a factor of five is used to find the correct location. It is then possible to treat \textit{winRow} as a single winning row, just as in the original program.

Since the results of every unique row is calculated the array which contain the results is extended. In the original program it can store three integers. However, since the program now uses \textit{nRows} number of rows instead of one, the array is extended to \textit{nRows*3} cells. The loop-variable \textit{k} is used to store the results on the correct index.

6.5.3 OpenMP
This section builds on the implementation described in section 6.4.1. However, the modification to the program is the same as described in the previous section 6.5.2, where an additional for-loop is added to iterate all the unique rows. It
retrieves unique rows from the \textit{uRows} array in the same manner, to use it just as in the original program.

Then the result array is extended with a factor of the number of unique rows, to be able to store them all.

6.5.4 Pthreads

This is an extension of the implementation from section 6.4.2.

The main difference in this program compared to the original is how the work is distributed. In the original implementation all bets are distributed among the threads, which are the variables \textit{threadWork}, \textit{laterStart} and \textit{extraWork} as previously explained. However, instead of distributing bets, whole winning rows are divide among the threads. This means that one thread iterates all bets for a number of times with different rows as the winning row.

An additional for-loop is added to the program which iterate the rows. While it retrieve the correct current winning row and store the result in the same manner as shown in section 6.5.2.

6.5.5 CUDA

This section builds on the implementation described in section 6.4.3.

Instead of adding a for-loop to iterate the extended data another approach is taken, and that is to utilize grid dimensions, just as described in section 5.5.

For this case, the added dimension to the grid is the same number as the amount of unique played rows. Because of this approach, the original program is minimally altered. The only change in the logic is the addition of another index, called \textit{x\_index}, which corresponds to the newly added dimension. This index is used to fetch which winning row the program executed with and is achieved with the following line:

\begin{verbatim}
  uint8_t *d_winNums = d_allWinNums + x\_index * 5;
\end{verbatim}

Where \textit{d\_allWinNums} is a pointer to the location of all unique rows and \textit{d\_winNums} is the variable where the current winning row is saved. By using \textit{x\_index} to fetch a winning row from \textit{d\_allWinNums} and the fact that \textit{x\_index} ranges from zero to the number of unique rows (minus one), means that one unique row is distributed to every block. When using the variable \textit{d\_winNums} as the winning row, the rest of the logic to calculate the winning groups works in the same manner as the original implementation.

The final difference is that result for every unique row need to be saved. This is solved in the same manner as in section 6.5.2, where the results array is extended with a factor of number of unique rows.
7. Evaluation - Lotto

The evaluation of the implemented applications is conducted in two phases. The first tests are made to evaluate the number of threads, and blocks, which are optimal for the different implementations. The second part use these numbers to run the applications on the test data to retrieve the calculation time.

7.1 Test Data

The data which is used for the tests is provided by Svenska Spel. It is authentic data from an earlier iteration of the game Lotto with all personal and transaction data removed.

The particular dataset contains 995 643 wagers and is 100MB. Different sizes of the input data is used for the tests, which is constructed by duplicating the original file a specific number of times.

7.2 Results

The following sections presents results from the conducted tests. The first section 7.2.1 decides a number of threads and blocks to use for the calculation time tests in section 7.2.5. Section 7.2.6 presents the results from the extended implementations.

7.2.1 Threads and Blocks

It is not straight forward to select the optimal number of threads and blocks for different applications. Different CUDA devices have different capabilities that needs to be considered. These are maximum number of active threads, warps, registers per thread, registers per block, shared memory per block and blocks per SM.

The most efficient and certain approach is to run tests.

For this case the calculation time for the different applications are tested for different input file sizes, threads, and blocks. For each combination of parameters the function is called ten times, and the value used in the graph is the median after removing the largest value. The used plots are normalized with the factor of duplication of the used input data. For instance, the calculation
time for any program which used the x20, which means that the base file is duplicated 20 times, is divided by 20. This is done to easier visualize where an efficient thread count is located for all file sizes. The calculation time is not the importance of these tests, as this is the main subject for the upcoming section.

The time measured is the calculation time of the actual iteration of the data. So, the preprocessing and initialization is never taken into account in the graphs. Additionally, the time it takes for CUDA to copy the dataset from the host to device is neglected, as this time is not of interest.

**OpenMP**

In graph 7.1 the compute only time for various thread counts are plotted for the implementations with the offset array, single rows and single rows with system rows. The x-axis represent the number of threads used and the y-axis is the time in ms. Since the CPU have four cores and no hyper-threading no more than four threads are tested.

As illustrated in the graph, all of the algorithms scales well with the increasing number of threads. This means that none of the algorithms visibly suffers from any load-imbalance and can efficiently divide the amount of work. However, from table 7.1 the exact speed up achieved is presented and it shows that the implementation with single and system rows utilizes four threads most efficiently.

<table>
<thead>
<tr>
<th>OpenMP Imp.</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset Array</td>
<td>1.73</td>
<td>2.46</td>
<td>3.26</td>
</tr>
<tr>
<td>System Rows</td>
<td>1.91</td>
<td>2.81</td>
<td>3.61</td>
</tr>
<tr>
<td>Single Rows</td>
<td>1.90</td>
<td>2.62</td>
<td>3.48</td>
</tr>
</tbody>
</table>

*Table 7.1. The speed up achieved by the different implementations depending on the number of threads, compared to the same program running with one thread.*

**Pthreads**

In the graphs plotted in figure 7.2 illustrates the compute only time for the different implementations with Pthreads against number of threads. The blue color represent the result from the execution of the input data with the offset array, red represent single rows data and the green shows the single rows with system rows.

All three of the implementations follow the same pattern, where they can utilize the number of threads. In table 7.2 the exact achieved speed up depending on the number of threads is shown. Just as for OpenMP it is the implementation with single and system rows that can utilize the number of threads the most.
Figure 7.1. The compute only for different implementations for OpenMP plotted against number of threads.

Table 7.2. The speed up achieved by the different implementations depending on the number of threads, compared to the same program running with one thread.

<table>
<thead>
<tr>
<th>Pthreads Imp.</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset Array</td>
<td>1.68</td>
<td>2.52</td>
<td>3.29</td>
</tr>
<tr>
<td>System Rows</td>
<td>1.91</td>
<td>2.75</td>
<td>3.48</td>
</tr>
<tr>
<td>Single Rows</td>
<td>1.91</td>
<td>2.58</td>
<td>3.35</td>
</tr>
</tbody>
</table>

CUDA

Several factors needs to be taken in to consideration when selecting a thread and block count for the different CUDA implementations. The number of registers each thread use and how many threads each block holds limits how many blocks each SM can have. The hardware limit of NVIDIA GeForce 1060 with compute capability 6.1 are 65536 registers per SM. There are at most 2048 active threads per each SM, which means the product of the threads and blocks per SM should be 2048. Shared memory is also a limiting factor. This memory is declared from every block and the limit for the setup of this project is 98 304B per SM. Finally, it is important to have a thread count within each block which is divisible by 32 to make maximum use of the warps.
Figure 7.2. The compute only time for different implementations for Pthreads plotted against number of threads.

Four different CUDA implementations for various input data is tested and plotted for every tenth block and 32:nd thread. The right column in figure 7.3 display the implementation for the single rows input data with naive memory accesses, and the left column show the same with strided memory accesses. The graphs have block count on the y-axis, thread count on the x-axis and the color scale indicates the calculation time, where green is low and red is high.

The only difference between these two implementation is how the threads accesses memory, and this results in vastly different graphs. For the strided memory pattern it show that when the thread count is greater than 192, all combinations of threads and blocks yields a low calculation time not depending on the size of the input data.

However, the configuration of a thread count of 512 and block count of 40 fulfill the factors mentioned earlier. The implementation use 21 registers per thread and each block allocates 10kB of shared memory. 21 registers per thread and 512 threads per blocks make it possible to have five blocks for each SM. The shared memory of 10kB allow nine blocks per SM. When each SM use 4 blocks with 512 threads all threads can be used efficiently in the warps and the number of active threads are 2048.

As for the naive implementation, the graphs changes depending of the size of the input data. The execution time increase with higher number of threads.
and blocks. This is because each warp accesses the memory inefficient, meaning that every warp require several memory loads and using a high number of warps cause too many memory loads to occur. This is also why every graph have a green area at the same location close to the bottom left corner and along the axes.

However, in the graph with the base file a green area is present in the upper right corner. This is because at this point there are enough threads and blocks to hide the memory load latencies. This area require more threads and blocks with the increasing file size, which is why it is not present in the other graphs.

Additionally, there are some yellow ripples in the red areas of the graphs. These occur under certain circumstances where the combination of blocks and threads match a workload multiple of the data.

These factors make it difficult to pick a thread and block combination that works for all file sizes. The configuration that used have 512 threads and 2000 blocks. This is because 2000 blocks works to hide latencies for the smallest file size, which is the only file size used for the extended runs. By using 512 threads the different factors of register count, shared memory and such also works as describes previously.

In figure 7.4 the left column display the compute time for the implementation using the offset array which is forced to 32 registers and on the right side is regular implementation with 38 registers. As shown in the graph, the implementation with 32 registers do not perform as well as the normal implementation and it does not scale as good for larger file sizes.

So, for the configurations it is shown that approach of using a large number of blocks to hide latencies is the most effective technique. For both of these implementations the configuration of 1024 threads and 1000 blocks is used.

The final implementation using single and system rows results in the same graph as for the single rows implementation with strided memory accesses, which is shown in figure 7.3 in the left column. The implementation registers and shared memory which it uses also results in the same configuration as for single rows, which is 512 threads and 40 blocks.

7.2.2 Summary
This section is a summary of the results from the tests in section 7.2.1, where all results are compiled in tables which maps a function with a number of threads, and blocks for CUDA, that is suitable for the different file sizes.

OpenMP and Pthreads worked in the same manner, as all implementations scales with the number of threads the optimal configuration is four threads for every implementation.

The results from the tests of the CUDA implementations are shown in table 7.3. As seen from the graphs in previous section, and argued for, these configurations suits the corresponding implementation.

54
Table 7.3. Threads and block suitable for all file sizes for the CUDA implementations.

<table>
<thead>
<tr>
<th>CUDA Implementation</th>
<th>Threads</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset Array 38 reg</td>
<td>1024</td>
<td>1000</td>
</tr>
<tr>
<td>Offset Array 32 reg</td>
<td>1024</td>
<td>1000</td>
</tr>
<tr>
<td>Single Rows Naive</td>
<td>1024</td>
<td>1000</td>
</tr>
<tr>
<td>Single Rows Strided</td>
<td>512</td>
<td>40</td>
</tr>
<tr>
<td>System Rows</td>
<td>512</td>
<td>40</td>
</tr>
</tbody>
</table>

7.2.3 Execution Time Comparison

In graph 7.5 the total execution time for the different implementation is displayed where the original file size of approximately 1 000 000 wagers is used. The configurations summarized in 7.2.2 are used for the tests.

The dashed areas represent the preprocessing time for the corresponding algorithm and the white areas with a colored outline represents the overhead for CUDA. The black bar is the baseline algorithm which requires no preprocessing. As shown in the graph, no implementation has a execution time that is faster than the original baseline implementation. This is simply because the preprocessing time, and overhead for CUDA, is too large compared to the compute time.

However, all achieved a compute speed up and the extended algorithm still do the same preprocessing but instead of using the data once they utilize it several times. Therefore, in the upcoming sections the compute only time is investigated for the different implementations over various file sizes.

7.2.4 Compute Time Comparison

In the following tests the number of threads, and blocks, presented in section 7.2.2 are used. The graphs in this section only show the compute time for the different algorithms, to find the most efficient of each.

Figure 7.6 display the compute time for the sequential implementations with the offset array, single rows and single rows with systems. The different calculation times grows linearly with the increasing file size. The best version for all file sizes is with the single rows data.

Figure 7.7 display the plot for the Pthreads implementations. The bars grows linearly with the file size and the implementation with single rows always outperform the other implementations.

The calculation time for OpenMP is plotted in figure 7.8. From the graph it is shown that the different bars scales in the same manner as for sequential and Pthreads, where the implementation with single rows is the fastest.

Finally, the calculation time for the CUDA implementations is plotted in figure 7.9. CUDA also include a bar for the version which forces the implementation to use 32 registers, instead of the original 38, and another bar with
naive memory access pattern for the row implementation. The graph show that
the naive implementation grow exponentially between some file sizes, which
is not desirable. The reason behind this is that the warps need to make more
memory accesses when there is more work, and this affects the performance.

The implementation which is forced to 32 registers for the offset array im-
plementation is faster for the smallest file size, but loses performance on larger
file sizes. This means that the extra six registers are more important for the
execution time than achieving 100% occupancy.

The implementations with single rows and single with system rows are sig-
ificantly faster than the other versions, even though the system and single
rows implementation suffers from divergence it still is the most efficient since
in this case it does less work on the unexpanded systems that it would if they
were expanded.

7.2.5 Final Compute Time Comparison
In figure 7.10 show the comparison between different implementations. The
used version from each individual category is the most efficient version, which
is sequential and Pthreads with single rows, OpenMP with offset array and
CUDA with single and system rows. The black bar represents the baseline
algorithm which requires no preprocessing of the data.

<table>
<thead>
<tr>
<th>Imp.</th>
<th>Time(x1)</th>
<th>x5</th>
<th>x10</th>
<th>x15</th>
<th>x20</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>81.2</td>
<td>406</td>
<td>812</td>
<td>1218</td>
<td>1624</td>
<td>1</td>
</tr>
<tr>
<td>Sequential</td>
<td>20.18</td>
<td>100.98</td>
<td>205.72</td>
<td>301.39</td>
<td>408.71</td>
<td>3.99</td>
</tr>
<tr>
<td>OpenMP</td>
<td>7.20</td>
<td>35.84</td>
<td>71.47</td>
<td>108.10</td>
<td>143.06</td>
<td>11.32</td>
</tr>
<tr>
<td>Pthreads</td>
<td>6.97</td>
<td>34.50</td>
<td>68.93</td>
<td>103.47</td>
<td>138.04</td>
<td>11.77</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.32</td>
<td>1.49</td>
<td>2.96</td>
<td>4.42</td>
<td>6.37</td>
<td>266.34</td>
</tr>
</tbody>
</table>

Table 7.4. Summary of the calculation time in milliseconds of the most efficient
implementation for each category. The columns are labeled with the multiple of the base
file, which is the file size. The speed up column is the average speed up for all file sizes
compared to the baseline algorithm in the first row.

Table 7.4 display a summary of all compute times and the speed up compared to the baseline implementation. As presented in the table, all algorithms
achieve a speed up, which means that when using the data enough number of
times all algorithms will at some point be faster than the baseline algorithm.
This is covered in the next section.

**Roofline**
By counting the number of operations that are made for each byte loaded in
the algorithms one can calculate its operational intensity and plot a line in the
roofline model.
In the CPU roofline model, figure 7.11, it shows that it is memory bound. This reflects how uniforming the data improves performance because features like prefetching of memory can be utilized.

The GPU roofline model for lotto can be seen in figure 7.12 shows that the GPU implementation is limited by memory bandwidth, since its operational intensity is low. This reflects the large difference in performance when ensuring strided memory for efficient memory accesses within the warps.

7.2.6 Extended Implementation

As shown in section 7.2.3 the preprocessing time is too large to make it possible for a single run to outperform the baseline algorithm. In this section the extended implementation is presented.

The general idea is that all the rows are allocated and transferred to the GPU, and then instead of just using them all once to check against a particular row they are used several times against different winning rows to calculate different outcomes.

Graph 7.13 illustrates the execution time for the different implementation in milliseconds plotted against the number of time which the algorithm utilizes the data. After three runs all implementations that require preprocessing runs faster than the baseline implementation and around five runs the CUDA implementation is the fastest. So, by using the data more than five times all overhead to prepare the data is overcome by the reduced compute time.

The extended implementation, which use every combination of seven-numbered row as the drawn row, run the same algorithm on roughly 6.7 million additional rows. This means that the data is used 6.7 million times, which is well beyond the five time threshold. For this the most efficient implementation is used, which is the CUDA version with single and system rows.

The measured time for this test include all parts of the program from copying the data to the GPU, running the calculations and writing the results to a text file. The final execution time is 15 minutes.

Since this is the only implementation a theoretical time is calculated for the baseline algorithm. So, one run for the baseline algorithm is measured to take 81.2ms. To get the total time for the extended run this time is multiplied with the number of combinations of rows, which is 6.7 million. This gives the execution time of roughly 150 hours.
Figure 7.3. Graphs displaying the calculation time for the implementation with CUDA using the single rows data. The right column using naive memory accesses and left column accesses memory with a stride.
Figure 7.4. Graphs displaying the calculation time for the implementations with CUDA using the offset array. The left column display the implementation with 32 registers and the right column with 38.
Figure 7.5. The total execution time for all algorithms on 1,000,000 wagers.

Figure 7.6. Calculation time for sequential implementations for different file sizes.
Figure 7.7. Calculation time for Pthreads implementations for different file sizes.

Figure 7.8. Calculation time for OpenMP implementations for different file sizes.
Figure 7.9. Calculation time for CUDA implementations for different file sizes.

Figure 7.10. The final calculation time for all implementations.
Figure 7.11. The roofline model applied on the single rows algorithm running on the CPU for Lotto. All implementations have a similar operational intensity.
Figure 7.12. The roofline model applied on the system and single rows algorithm running on the GPU for Lotto.

Figure 7.13. The theoretical runtime for the different algorithms when increasing the number of times the algorithm is used on the preprocessed data.
8. Evaluation - Bomben

The evaluation of Bomben is conducted in the same manner as for Lotto.

8.1 Test Data

The test data which is used is authentic data from an earlier iteration of the game Bomben at Svenska Spel, which contain 246,290 total rows and is 297kB. However, as mentioned in section 6.2.1, the data is extended with an additional match. This is because the data need to be more complex for it to be viable to make an parallel implementation of it.

The original version of Bomben has a very strict win-condition where all rights are needed, meaning that it only need to count the amount of entries that match the winning result. This results in a trivial algorithm, which performance well for given data sets even when using unrealistically scaled data at 1000 times the original size.

Also, Svenska Spel have offered game modes in the past which used this kind of data, and will possibly do so in the future.

8.2 Results

In the following sections the results from the tests are presented. Section 8.2.1 investigate the number of threads and blocks to use for the tests conducted in section 8.2.3. Finally, section 8.2.4 contain the results for the extended implementations.

8.2.1 Threads and Blocks

Thread and block count is determined in the same manner as for Lotto, with tests iterating on different number of threads, blocks and sizes on the input data. However, from the results of Lotto different implementations have been omitted to only implement efficient versions.

In the two following sections the conducted tests for OpenMP, Pthreads and CUDA is presented. The tests are conducted in the same manner as described in section 7.2.1.
OpenMP
The computation time for the smallest and largest file size depending on the number of threads is in graph 8.1. The values for the largest file size are normalized to make them easier to compare against the smallest file size.

The total compute time decrease with the increasing amount of threads. However, the speed up is not significant when going from two to three threads, but it is more efficient when increasing the data size.

The exact speed up compared to running the algorithm with one thread is shown in table 8.1.

![OpenMP - Compute only](image)

*Figure 8.1. Compute time for the OpenMP implementation for the smallest and largest file size plotted against thread count. The values for the largest file size are normalized with a factor of 20, to make them easier to compare against the base file values.*

<table>
<thead>
<tr>
<th>OpenMP Imp.</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Rows</td>
<td>1.61</td>
<td>2.12</td>
<td>2.41</td>
</tr>
<tr>
<td>Single Rows x20</td>
<td>1.98</td>
<td>3.00</td>
<td>3.89</td>
</tr>
</tbody>
</table>

*Table 8.1. The speed up achieved depending on the number of threads.*
The graph in figure 8.2 show the compute only time for the Pthreads implementation for the smallest and largest file size plotted against the number of threads. The values for the largest file size are normalized to make them easier to compare against the smallest file size. The compute time decrease for every added thread, but the implementation do not gain much speed up going from two to four threads. However, when increasing the input data size the algorithm makes more efficient use of the increasing number of threads.

The exact speed up compared to running the algorithm with one thread is shown in table 8.2.

**Figure 8.2.** Compute time for the Pthreads implementation for the smallest and largest file size plotted against thread count. The values for the largest file size are normalized with a factor of 20, to make them easier to compare against the base file values.

<table>
<thead>
<tr>
<th>Pthreads Imp.</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Rows</td>
<td>1.57</td>
<td>1.71</td>
<td>1.97</td>
</tr>
<tr>
<td>Single Rows x20</td>
<td>1.88</td>
<td>2.73</td>
<td>3.43</td>
</tr>
</tbody>
</table>

**Table 8.2.** The speed up achieved depending on the number of threads.
CUDA
The execution time plotted against threads and blocks for CUDA is displayed in figure 8.3. In the top graph the base file is used, and in the other graphs the file is duplicated 5, 10, 15, 20, and 25 times to simulate larger input data. This make it possible to find a combination of threads and blocks which works with varying size of input data.

The graph representing the base file has a yellow area in the top right corner. This is because the number of threads is too large for the amount of work that is possible to distribute. When using 320 blocks with 1024 threads you get a total of 327 680 threads. There are approximately 250 000 rows in the original data. This means that in some cases there are not enough rows to even distribute one to every threads, and most cases there are one or two for each thread. For these cases the overhead exceeds the time which is possible to save from parallelizing.

The graphs show that all the implementation yields good results for all file sizes as long as enough threads and blocks are used. As for the final configuration 1024 threads and 20 blocks are used. The implementation use 26 registers per thread and 12kB shared memory per block. Therefore, by using 1024 threads with 26 registers each there is possible to have two blocks for each SM, which results in 2048 active threads. The shared memory per block is small enough to support this number of blocks.

8.2.2 Execution Time Comparison
In the graph 8.4 the implementations are tested on the original file size of around 250 000 rows. The configurations presented in 8.2.1 are used.

The dashed areas represent the preprocessing time, the white area is the overhead for CUDA and the black areas are the computation time.

No execution time of any of the algorithms are better than the baseline case which require no preprocessing. The CUDA overhead is by far the most time consuming aspect of all the implementations. This is because the data is so small that it is not the actual transfer time that is the most significant part, instead it is all the set up.

However, the time spent preprocessing the data and overhead for CUDA have significantly reduced the compute only time for the algorithms. So, in the upcoming section the compute only time is investigated, to make implementations for the extended algorithms.

8.2.3 Compute Time Comparison
The compute times for the different implementations, using the threads and blocks from the tests in section 8.2.1, are plotted in figure 8.5. As illustrated, the CUDA implementation have an apparent performance gain compared to
both the Pthreads and sequential implementation. However, all implementations achieve a significant performance gain in compute time compared to the baseline implementation.

The exact compute times in milliseconds for the different implementations are assembled in table 8.3. The column labeled Speed Up contains the comparison of the summarized calculation times relative to the baseline implementation.

**Roofline**

Unlike Lotto the CPU roofline in figure 8.6, shows that the algorithm is close to the ridge point. This means that it theoretically can utilize the hardware fully.

The GPU roofline, figure 8.7, shows the same as Lotto with it being memory bound. Since Bomben has lower operational intensity than Lotto, it can not utilize as much of the benefits the GPU has to offer.

*Figure 8.3.* Runtime for the CUDA implementation with different file sizes plotted against thread and block count.
Table 8.3. Summary of the calculation in milliseconds for each implementation. The speed up column is the average speed up for all file sizes compared to the sequential algorithm in the first row.

<table>
<thead>
<tr>
<th>Imp.</th>
<th>Time(x1)</th>
<th>x5</th>
<th>x10</th>
<th>x15</th>
<th>x20</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>5.2</td>
<td>26</td>
<td>52</td>
<td>78</td>
<td>104</td>
<td>1</td>
</tr>
<tr>
<td>Sequential</td>
<td>0.52</td>
<td>2.55</td>
<td>5.03</td>
<td>7.56</td>
<td>9.99</td>
<td>10.34</td>
</tr>
<tr>
<td>OpenMP</td>
<td>0.14</td>
<td>0.69</td>
<td>1.38</td>
<td>2.25</td>
<td>3.08</td>
<td>35.14</td>
</tr>
<tr>
<td>Pthreads</td>
<td>0.44</td>
<td>1.71</td>
<td>3.29</td>
<td>3.32</td>
<td>6.47</td>
<td>17.41</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.03</td>
<td>0.10</td>
<td>0.17</td>
<td>0.25</td>
<td>0.33</td>
<td>299.02</td>
</tr>
</tbody>
</table>

These rooflines show that the implementations for Bomben might not be as beneficial as for Lotto, since the CPU implementation is better utilizing the hardware.

8.2.4 Extended Implementations

As shown earlier, the baseline implementation is faster than any of the parallel versions, since the preprocessing takes too much time. In this section the extended algorithm is presented.

In graph 8.8 the theoretical for the different algorithms are plotted depending on the number of times the preprocessed data is used. At five iterations of the data all algorithms, except the CUDA implementation, are faster than the baseline implementation. Even though not shown in the graph, CUDA is the fastest implementation at around 1000 iterations of the data. This is because of the large overhead of allocation and copying data to the GPU.
This extended implementation run the algorithm on ever unique row from the original Bomben data, which is about 61,000 rows. This is large enough to make every algorithm run faster than the baseline algorithm, according to the graph.

As the extended implementations for Bomben are not as time consuming as for Lotto it is possible to make an implementation for every approach to see if the actual difference in execution time. The final documented time for each implementation is the median from three runs and it include all parts of the program.

However, all implementations uses the same function to determine all unique rows from the original data, as described in section 6.5.1. This implementation run sequentially on the CPU and have an execution time of 15 seconds.

Table 8.4 contain the execution times for the different algorithms with and without the overhead of calculating all unique rows, with the corresponding speed up compared to the baseline algorithm.

As shown when running the extended implementation the speed up achieved is getting closer to the compute speed up, since the preprocessing is an insignificant part of the execution time. It is also shown that Pthreads performs better when it had more data to work on.

However, when including the overhead time of 15s to determine all unique rows CUDA goes from a 227x to 20x speed up. This indicates that this preprocessing is still large enough to be significant bottleneck for this implementation, since it has such a low execution time.
Figure 8.6. The roofline model applied on the single rows algorithm running on the CPU for Bomben. All implementations have a similar operational intensity.

Table 8.4. Summary of the execution time in seconds and the speed up compared to the baseline algorithm. The execution time includes all preprocessing and CUDA overhead for the corresponding implementation. The columns with "+OH" also includes the overhead of calculating all unique rows.
Figure 8.7. The roofline model applied on the single rows algorithm running on the GPU for Bomben.

Figure 8.8. The theoretical runtime for the different algorithms when increasing the number of times the algorithm is used on the preprocessed data.
9. Conclusion

The problem at hand of this report is to make parallel implementations for winner selection algorithms for the games Lotto and Bomben at Svenska Spel. By making an implementation that is fast it not only ensure scalability for more player and more compute-heavy games, it could also enable the possibility to add more features that is not possibly with the current implementations. These features could for instance be so called "what if"-prognosis, where the users want to know the outcome of multiple different scenarios. This would require it to be possible to calculate the winning groups for several different outcomes in the matter of minutes or even seconds depending on the type of game.

The data used is authentic data for both Lotto and Bomben from earlier iterations of the games, with personal and transaction data removed. The Lotto file contains roughly 1 000 000 wagers and is 100MB, while the file for Bomben is 300kB and contains around 250 000 rows. These files are multiplied up to 20 times to simulate a larger player base.

Challenges for this project come in several different forms, but many of them do not just revolve around making a parallel implementation but an efficient parallel implementation.

The original format of the data is structured in a manner where it is impossible to locate a wager out of order. This makes it impossible to split the data efficiently for it to be processed in parallel. To utilize the GPU the data need to be allocated and transferred to it, which can be a large bottleneck of the performance. Additionally, the code running on the GPU need to consider, among other things; warps of threads, memory accesses, and utilizing shared memory.

Three different approaches are used for the parallel implementations; OpenMP, Pthreads and CUDA. This is to evaluate how they perform compared against each other and what the challenges are with them individually. Three different preprocessing approaches of the data are investigated which are referred to as; offset array, single and system rows, and single rows. All three are used for every implementation of Lotto and single rows for the Bomben implementations.

The offset array is an array with absolute byte offsets to the individual wagers to make it possible to accesses them out of order, other than that the original data is intact. The single and system rows preprocessing includes extracting only the relevant information from the wagers, which are the rows. These are stored uniformly in an array to make them easy to access. The last method includes generating all the rows which every system represents. This
makes the data larger, but it reduces load-imbalance and divergence for the CUDA implementations.

The results for both Lotto and Bomben shows that no parallel implementation has a shortened execution time compared to the baseline implementation. This is simply because the preprocessing of the data is too time consuming. Also, for both cases it is the CUDA implementation which is the slowest, since it got the additional overhead of allocating and transferring the data to the GPU.

However, for all implementations a speed up is gained when just comparing the compute time. By preprocessing the data to single rows the sequential algorithm gained a 4x speed up in compute time for Lotto and a 10x speed up for Bomben. OpenMP and Pthreads performs equally for Lotto where they achieve around 11x speed up, and single and system rows for CUDA got a compute speed up of 266x. OpenMP for Bomben has a speed up of 35x and Pthreads 17x, while CUDA has a compute speed up of 299x, all with single rows.

The extended implementations of the algorithms can utilize these speed ups fully since it runs the required preprocessing, allocation and data transfer only once, but uses the data several times. For Lotto this implementation is made only for CUDA. This implementation includes running the algorithm on every seven-numbered, order independent, permutation of rows with numbers from one to 35.

This means that the data is used roughly 6.7 million times and it takes 15 minutes for it to execute. Since the function still require the same preprocessing as previously, of roughly 200ms, it now occupies an insignificant part of the execution time. A theoretical execution time of the same implementation of the baseline algorithm is estimated to take 150 hours.

Since the data for Bomben is only 300kB, compared to 100MB for Lotto, an extended implementation is made for every approach. It uses every unique played row as input data, which is roughly 61 000 rows.

The execution time for the different extended implementations are; baseline 317.4s, sequential 29.2s, OpenMP 8.4s, Pthreads 8.9s and CUDA 1.4s. This means that all implementations achieve significant speed up compared to the baseline implementation. However, the overhead to calculate all unique rows is 15s, which becomes a large bottleneck for all the implementations. It reduces the speed up gained by the CUDA implementation from 226x to 20x, which brings it close to the performance of OpenMP and Pthreads.

So, as for conclusion, no parallel implementation show any potential for a single run because of the preprocessing of the data. Additionally, since the compute time is so low relative to the allocation and transfer time, there is no benefit in using the GPU for the calculations.

However, if the preprocessed data is used several times there is large potential for some of the algorithms. The extended CUDA implementation for Lotto show a significant performance gain. As for Bomben it comes down to how
fast the algorithm needs to run. If it is enough to run the algorithm once every 15min, the baseline will do good enough. But if more frequent updates are required, like once every minute, a parallel implementation is needed. However, the difference between the CPU and GPU implementation is small, making the GPU implementation only useful in special time constraint cases.
10. Discussion

10.1 Future work

In this section theoretical future improvements are discussed. It is mainly ideas without any specifications about how they actually should be implemented.

10.1.1 Hardcode outcomes

Hardcode all outcomes for the single rows with system implementation for lotto. This can be done by hardcoding all possible outcomes in a two-dimensional array and using the size of the system and amount of correct numbers as indexes.

This would ensure that no divergence or load-imbalance occurs on the GPU. However everything is treated as a system with potential winnings in every group, even tho most entries are single rows. This extra logic might slow down more than it improves.

10.1.2 Vector Reduction

In the current implementation for CUDA each instance make atomic calls to calculate the sum for each group. This can be replaced with a parallel vector sum reduction. All threads in a block can then use the built-in function shuffle down to effectively summarize the output instead of having each core do several atomics adds. This can bee seen in figure 10.1. Note that only arrows that are contributing to the final reduction are included in the figure. In reality all threads will be shifting values even though they are not needed in the reduction.

10.1.3 Persistent Allocation of Data on the GPU

The main issue for using CUDA on smaller problems is that the time consumed by transferring the data to the GPU is the majority of the executing time. However, if the necessary data always is located on the GPU this overhead is removed. Even if it is not possible for all of the data to be located on the GPU, schemes could be developed to minimize the amount of data which is needed to be transferred.
10.1.4 Popcount Optimizations

For lotto a large bulk of the compute time is with popcount and the default implementation on the CPU on our test system can be improved by using a different combination of bit operations and bit shifts.

10.1.5 Adding Bonus Numbers to the Lotto Extended Implementation

No bonus numbers are included in the current extended implementation for lotto. This can be implemented by doing additional runs with bonus numbers on the results for certain rows. However this workload grow very quickly and need to have some form of reduction to be feasible to execute since the estimated runtime of our extended implementation with bonus numbers would take more than 200 days to finish.

10.1.6 Sorted System Preprocessing

Sort rows in the preprocessing phase based on system size. This removes most of the divergence created by having systems spread out in the data. For example, the data given consist of 1.4% systems which creates a total of 31.9% divergence, since warps execute in groups 32 threads and only one thread causes all other to stall if it encounters a system.

10.1.7 Evaluate GPU algorithms on different hardware

Testing the scaling and performance on different hardware that have a capacity for a different amount of cores and different clock speed.

Figure 10.1. Example of the shuffle down call
References


Appendix A.
Work Distribution

Large parts of the project have been done together, other sections have been led by either one of us.

Joint work

- Understanding the original data structures.
- The sequential baseline implementation for Bomben and Lotto.
- Offset array preprocessing.
- The initial core concept of CUDA.
- Test suit for testing threads/blocks and final calculation times.
- Roofline research.

Work focused by Johan

- Refined preprocessing; uniforming of data.
- Refined CUDA; warps/occupancy optimization and memory patterns.
- Lotto extended implementation.
- Python code for graph generating and test management.

Work focused by Jimmy

- Core Pthreads implementations.
- Core OpenMP implementations.
- Improved data structures.
- Bomben extended implementations.