Abstract: Applying the currently fastest known algorithm for computing edit distances, the Myers algorithm and variations from Hyyrö, on the graphic card to parallelize and schedule a large number of computations, finding text occurrences with $k$ or fewer differences, and making use of the fast GPU cores. Implementing for users an extendable interface as an enhancement of SeqAn – a powerful, rich sequence analysis library – to enable GPGPU and execute Myers on GPU. The technology behind is described with its programming APIs, the implementations, and empirical test are shown what benefits can be achieved.
# Table of contents

1 Introduction ............................................................................................................ 4

2 Myers Fast Bit-Vector Algorithm ........................................................................... 5
   2.1 Historical development of Myers Algorithm .................................................. 6
   2.2 Details of Myers Algorithm ........................................................................... 9
      2.2.1 Preprocessing the pattern ..................................................................... 10
      2.2.2 Computing the edit distance ................................................................. 11
   2.3 Variations of Myers Algorithm by Hyrrö and for GPU usage ..................... 12
      2.3.1 The unbanded algorithm for computing the Levenshtein distance .......... 13
      2.3.2 The unbanded algorithm for the Damerau distance ............................. 13
      2.3.3 The banded algorithms for Levenshtein and Damerau Distance .......... 14

3 Graphic cards and technical aspects of GPGPU ............................................... 17
   3.1 Evolution of graphic and graphic cards ......................................................... 17
   3.2 The graphic rendering pipeline ..................................................................... 21
   3.3 Modern GPGPU capable graphic cards ....................................................... 23
      3.3.1 Structure of GPGPU capable graphic cards ......................................... 23
      3.3.2 The NVidia® CUDA™ Technology ......................................................... 24
      3.3.2 The ATI® Stream Technology™ and OpenCL™ ..................................... 26

4 Using GPGPU ........................................................................................................ 27
   4.1 Parallel Computing ........................................................................................ 27
   4.2 Concepts of GPGPU ....................................................................................... 28
   4.3 Thread assignment and execution on the GPU ............................................. 29
   4.4 Memory and variable usage on the GPU ...................................................... 31
1 Introduction

4.5 Best Practices for GPGPU ................................................................. 33
4.6 Limitations and usage for GPGPU .................................................... 35
4.7 CUDA™ and OpenCL™ principles .................................................... 37
  4.7.1 Programming with CUDA .......................................................... 37
  4.7.2 Programming with OpenCL ......................................................... 38
  4.7.3 Integrating CUDA and OpenCL .................................................. 38

5 GPGPU implementation for SeqAn..................................................... 40
  5.1 API independent integration ............................................................ 41
  5.2 Basic GPGPU classes ................................................................. 45
  5.3 Bit-vector classes for GPGPU ....................................................... 48
  5.4 Using GPGPU in SeqAn ............................................................... 52

6 Myers Algorithm on GPU ................................................................. 54
  6.1 Myers Scheduler Instance ............................................................. 55
  6.2 Using Myers Scheduler ............................................................... 58
  6.3 Scheduler buffering and data handling ........................................ 59
  6.4 Scheduler workflow ................................................................. 63
  6.5 The grid execution ................................................................. 66
  6.6 The kernel invocation and code binding ..................................... 68

7 Empirical Results ............................................................................ 70
  7.1 Linear scaling performance ......................................................... 71
    7.1.1 Performance for small patterns (m<=w) .................................. 71
    7.1.2 Performance for large patterns (w>w) ..................................... 73
    7.1.3 CPU discharge ..................................................................... 74
  7.2 General performance behavior .................................................... 76
  7.3 High throughput performance ..................................................... 78
### 7.4 Performance influencing parameters

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.1</td>
<td>Double vs. Triple vs. Quad buffering</td>
<td>82</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Influence of hit rate</td>
<td>83</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Influence of GPU architecture</td>
<td>83</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Local vs. shared vs. global memory variant</td>
<td>84</td>
</tr>
<tr>
<td>7.4.5</td>
<td>BitvectorArray vs. BitvectorList</td>
<td>85</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Iterative loops vs. unrolled loops</td>
<td>85</td>
</tr>
<tr>
<td>7.4.7</td>
<td>Cut-Off vs. no Cut-Off mechanism</td>
<td>86</td>
</tr>
<tr>
<td>7.4.8</td>
<td>Alphabet influence</td>
<td>86</td>
</tr>
</tbody>
</table>

### 7.5 Performance of banded algorithm | 88   

### 7.6 Performance of Hamming distance computing | 91   

### 8 Summary and Perspective | 93   

### 9 Literature and references | 95   

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Register of references</td>
<td>95</td>
</tr>
<tr>
<td>9.2</td>
<td>Register of citations</td>
<td>96</td>
</tr>
<tr>
<td>9.3</td>
<td>Register of illustrations</td>
<td>97</td>
</tr>
</tbody>
</table>

### Appendix | 98   

---

Page 3
1 Introduction

Myers Fast Bit-Vector Algorithm for Approximate String Matching, further on referred as Myers algorithm only, is used to solve a string-matching problem in the informatics. String matching problems occurs if one text has to be compared with another text -a matching pattern or needle- for finding equalities, dissimilarities, or occurrences of this pattern in the text. This is often the case in practice if a part of a text needs to be found in documents, or databases, or to query internet search engines finding relevant or adjacent websites with the requested content. Consequently and in times of an ever faster information flow, reliable and fast algorithm are strongly engaged. These string-matching problems are distinguished into exact and approximate matching problems. The Myers algorithm solves an approximate string-matching problem, computing the distance of two texts. Approximate string matching is an important topic in fields of computational molecular biology also. One common problem is to align two sequences of DNA, RNA, or Proteins with each other to find their biological correlation or familiar relations. In praxis, this is used to match DNA probes for crime investigations, declaring paternity, or looking for specific genes occurrence in genomes to predict diseases for example, also called sequence alignments. Sequencing technologies in the second-generation can deliver DNA sequences with an unprecedented high throughput. Mapping the DNA pieces, the reads, to a mostly highly similar reference genome needs fast applicable algorithms. Usually this refers to read mapping methods.

Myers algorithm is currently the fastest known algorithm computing the edit distance of two texts, using only a few bit operations and only a minimal amount of memory for computation. The method uses bit-vectors, which in cases that not more than 32 or 64 bits are needed, a linear runtime of $O(n)$ can be achieved, where $n$ is the length of the text to measure the distance of or find the occurrences in.

The ideas is now to use new generation graphic cards with its computing power, having hundreds of operational fast graphical processing units (GPUs) for calculations, to massive execute the algorithm in parallel in comparison to a serial execution on CPU cores. This is primary being done to deal with the high throughput of sequencing technologies for read mapping applications.

This work will present an implementation and parallelization of Myers algorithm on the graphic card using and integrating them into the SeqAn library, a generic sequence analyzing library for the computational biology developed by the department of bioinformatics from the FU-Berlin. It brings into focus how GPU programming works, what implementations has been applied, which usage strategies are best suitable, and which profiting cases can be obtained with the algorithm by using graphic devices with parallel computing abilities by empirical tests.
2 Myers Fast Bit-Vector Algorithm

Gene Myers published Myers Fast Bit-Vector Algorithm for Approximate String Matching in 1999 [MY99].

Formally, the goal is to find substrings P with k or fewer differences in a text T. In biological computing, the substring is called a pattern or a read, and the text is often denoted as reference genome, a sequence, or a large database query. In the field of the sequence analyses, where genomes are scanned for specific genes with deviants or for overlap mapping, this can be a long-running process due to large sequence lengths. Myers algorithm in combination with other preprocessing and filtering steps is by now the fastest known practicable algorithm that can solves this problem efficiently. It computes the distances between two texts at each position that can then be used to look for k or fewer differences.

The algorithms for string matching problems are usually based on the dynamic programming (DP) paradigm using rather simple recursion formulas, computing matrices of scores or distances between the texts. This got a special interest in focus of genome comparisons and sequence analysis in the beginning of the 70’s for detecting evolutionary distances and equalities. Inside one scoring matrix, they can align or back trace a path to correlate two gene sequences. Later string matching becomes important in information retrieval problems too, for example used by search engines finding many related websites in large databases.

Commonly in information and computer science theory, measuring a distance between two texts is expressed by the number of similarities or dissimilarities to each other, based on calculating the required number of operations to transform one of them into the other. There are different variations to define a so-called edit distance, and the following three are the most relevant in biotechnology:

- The Hamming distance is computed by just counting the differences at each position of the texts, meaning how many symbols has to be substitute to transform the texts into each other. The text strings must be in equal length.
- The Levenshtein distance is defined as the minimum number of operations needed to transform with the allowed operations of insertions, deletions, or substitutions of symbols. This distance score is commonly and widely referred by the term edit distance.
- The Damerau-Levenshtein, in the following only the Damerau distance, allows insertions, deletions, substitutions, and the transposition of two adjacent characters.

The Myers algorithm was originally based on computing the widely used Levenshtein edit distance with unit costs. Myers was inspired by combining several ideas in the topic of sequence analysis. H. Hyyrö later improved and extended the original algorithm. He presents methods for computing the Damerau distance and additionally for banded alignments. Hyyrö variations are primarily used in the implementations of this work.
2.1 Historical development of Myers Algorithm

The early process of solving exact string matching problems starts in 1960 with the beginning of gene sequencing methods.

The first important algorithm was published in 1970 with the Needleman-Wunsch algorithm for finding optimal global sequence alignments with arbitrary costs of substitutions, deletions, and insertions [NW70]. Giving a scoring scheme it computes a scoring matrix where every cell represents the value for the optimal alignment at the current positions between the two sequences, by a recursive rule. This algorithm runs in $O(n^2 m + m^2 n)$ or $O(\max(n, m)^3)$ time where $n$ is the size of the horizontal text –the sequence- and $m$ the size of the vertical text –the pattern-. This method needs $\Theta(nm)$ space for storing the score matrix, and can be used to solve general string matching problems.

Seller specified in his paper of 1974 from a variation with a run time of $O(nm)$ and affine cost for deletions and insertions [Sel74]. The Hirschberg algorithm in 1975 uses a divide-and-conquer strategy reducing the space requirement down to $\Theta(\min(n, m))$ [Hir75]. This method was general applicable for arbitrary matching algorithms. Later on in 1982, the Gotoh algorithm was released with $O(nm)$ time and $O(n + m)$ space complexity for computing with affine costs, applying the device-and-conquer strategy and Sellers adjustments [Got82].

The first algorithm pointing exactly the problem of finding $k$ or fewer differences for edit distances with unit costs was created by Seller in 1980 [Sel80]. This represents the classical approach to solve that problem, using $O(nm)$ time and space.

Let each cell of the score matrix be denoted as $C[i,j]$, then the computation can be described by using the following recursion scheme:

Initializing the first row and first column in the matrix:

$$C[0, j] = \begin{cases} 
0 & \text{for local scoring (occurrences starts anywhere)} \\
 j & \text{for global scoring (occurrences starts at the first position)} 
\end{cases}$$

$$C[i, 0] = i$$

Recursion scheme:

$$C[i, j] = \min \left\{ \begin{array}{l}
C[i - 1, j - 1] + \delta_i \\
C[i - 1, j] + 1 \\
C[i, j - 1] + 1
\end{array} \right\}$$

$$\delta_{i,j} = \begin{cases} 
0 & , \text{if Sequence}[j] = \text{Pattern}[i] \\
1 & , \text{else}
\end{cases}$$
The edit distance at each matrix cell position is determined by adding 1 for an error if sequence and pattern do not match there. The resulting distance scores are located at the lower horizontal row $C[m,j]$. The occurrences with a k or less distance arise if $C[m,j] \leq k$. The recursion formula for one cell depends only by the previous diagonal, vertical, and horizontal value for computation, which will be utilized by Myers algorithm later.

The next remarkable idea was published 1985 by Ukkonen [Ukk85]. He notice the cells of a scoring matrix early reaches values greater than k+1 in algorithms for finding k or fewer differences. Therefore only in a dedicated region it is essential relevant to compute, and to omit a large regions of the matrix with values > k. An imaginary band around k can bound the computation. The region is called the Ukkonen zone. This idea leads to banded alignments, which are suitable for efficiently computing a k-different matching between two sequences. The runtime for approximate matching algorithms could be suppressed to worst-case time of $O(kn)$. Space consumption improvements were made in 1988 by Landau and Vishin to $O(n)$ [LV88], and 1990 to $O(m^2)$ space by Galil and Park [GP90]. Further enhancement in 1992 by Chang and Lamp, and 1996 Wu, Manber and Myers achieves a runtime of $O(kn/\sqrt{\sigma})$ respectively $O(kn/\log s)$ [CL92, WMM96], where $\sigma$ is the size of the underlying alphabet and s a space consumption constant. Both variants run differently fast inside the parameters space of k and $\sigma$.

Baeza-Yates and Gonnet publicize another approach for scoring in 1992, using the capabilities of hardware parallelized bit operations at the processor [BYG92]. They invent an exact matching method in $O(n[m/w])$ time, where $w$ is the machine word size in number of bits, e.g. w=8, 16, 32, 64, … bits. The improvement of Wu and Manber for the k-difference in $O(kn[m/w])$ time [WM92] almost follows. Later in 1996, Baeza-Yates and Navarro realized a variation with a runtime of $O(n[km/w])$ [BYN96]. In circumstance where the pattern length m is lower than the machine word size w, these attempts have the runtime of only $O(n)$ respectively $O(kn)$ using the hardware optimized bit-vector operations.

Finally, 1999 Myers announce his algorithm combining the previous ideas. He used the bit-vectors for coding each column the scoring matrix in a representative way, and making use of the hardware parallelism of bit operations. The procedure equals Sellers column-wise computation. The method uses the preceding column state to determine the next, and identify the scores at the lowest row. The technique achieves a runtime for calculating one single column in the matrix in $O(\lceil m/w \rceil)$. Depending on the pattern length m, if m is below the machine word size w, $O(1)$ is obtained. The total computing runtime sums up to $O(n[m/w])$ in total, or only $O(n)$ especially for $m<=w$. Further, Myers algorithm is independent of k and the alphabet size $\sigma$. This compromise the drawback of the presented
Chang and Lamp and the Wu, Manber and Myers method. Myers fast bit-vector algorithm requires a preprocessing phase of $\Theta(m)$ for one pattern. The total runtime reaches $O(m + n)$ or $O(m + n\lfloor m/w \rfloor)$ in general. The space needed for storing the bit-vector states is $\Theta(\sigma \lfloor m/w \rfloor)$. If a fixed number of alphabet symbols is used, this can be treated as $\Theta(\lfloor m/w \rfloor)$ space requirements only, or even $O(1)$ with $m <= w$. [Mye99]

Heikki Hyyrö presented between 2001 and 2003 a revised version of Myers algorithm for the Levenshtein distance and modified the algorithm to calculate Damerau distances. Furthermore, he described methods for computing k-banded alignments with the Levenshtein and Damerau distance [Hyy01, Hyy02, and Hyy03].

The point of interest of such kinds of fast algorithms, with the limitation of using relatively small pattern length only, get important by inventing advanced filtering techniques for larger queries, like genomes are, and the sequencing devices that producing reads up to length of about currently 500 with a fast and high throughput.

The filter algorithms are used to pre-divide the large query sequence into smaller regions with a certain sensitive or by a mismatch rate. They localize and discard regions where the sequence cannot match with the pattern. The filtration efficiency increases for lower k-values. The more the mismatch rate $\varepsilon = k/m$ falls to zero, the more efficiently is filtered. For example the filter of Wu and Manber 1992 [WM92], Chang and Lampe 1994 [CL94] and Myers in 1994 [Mye94] comes up during the time. These small regions, commonly not more than twice the length of the pattern, where the probability of finding a matching increase to a certain level, or fall below a tolerated error rate, an algorithm can verify these areas in detail. The Myers algorithm and the variations by Hyyrö can be applied here enabling a fast verifications algorithm, the today fastest known.

A fast read mapper with sensitive control that makes use of this principle has been realized 2009 by Weese, Emde, Rausch, Döring, and Reinert with the RazerS application based on the SeqAn library [WDR09].

As Myers himself said that “improvements in verification-capable algorithms are still very desirable, as such results improve the filter-based algorithms when there are a large number of matches” (Citation [MYE]). This diploma thesis connects on the statement by using the potentialities of modern graphic cards to parallelize a massive amount of verifications, profiting of the fast GPU cores for computing, and outsourcing the process on GPU while disengaging the CPU for further filtering. Having $r$ number of verifications, in theory, instead of having a runtime of $O(r(m + n))$ serial executed, achieving a runtime of $O(m + n)$ with parallelization in cases of $m <= w$. Nevertheless, this is not true, since the graphic card has a limited number of cores and limited hardware for usage and is harmed by other restrictions of GPU parallel programming principles. The speedup by parallelism is constrained to the hardware capabilities, but this can be seen as a constant factor for one certain device in the runtime.
2.2 Details of Myers Algorithm

Myers first attempt was to make use of an observation lemma during computations of an edit distance: Each cell value in the matrix differs only by -1, 0 or +1 depending of its previous adjacent upper horizontal, left vertical, and left upper diagonal cell. ([MP80], [Ukk85])

Myers re-encodes the DP scoring matrix based on this lemma by noticing only the vertical and horizontal delta values accordingly between the adjacent cells.

Myers further observed then the general input and output behavior of one cell. Each cell value is dependent only on the 3 possible states of the horizontal delta value $\Delta h_{in}$ (+1,0,-1), the 3 state of the vertical delta value $\Delta v_{in}$ (+1,0,-1), and 0 or 1 state of the diagonal, if the pattern equals the sequence and the current position or not for a penalty. 18 combinations are enumerable possible for one cell, and by simply noting down all combinations a computing formula for one cell has been worked out.

The main idea is to encode one column in the matrix with the $\Delta$-values represented as bit-vectors. One bit-vector is responsible for the vertical negatives values (VN), one for vertical positives (VP), one for horizontal negatives (HN) and one for the horizontal positives (HP). These bit-vectors are in the length of one column, and therefore in the length of the pattern m. Together they represent one column state.

Myers then analyzed the correlation between one cell and the complete column in the bit-vector presentation. He observed and formulates the required bit operations to compute one column based on the previous column bit-vectors. It turned out that only a hand full of bit operations are needed for the vectors to compute one column completely at once instead of each cell separately. The complete scoring matrix can be determined by iterating over the sequence consecutively calculating the bit-vector states and detect the score for each column at the lowest row.
It can be observed that the edit distance scores in the last row, between two adjacent columns or text symbols, can only decrease or increase by 1, or the texts stays in the same distance to each other. Starting with the maximal edit distance of the pattern length m in the first left lower cell, the horizontal positive and horizontal negative bit-vectors at the last bit indicates how the score changes. If the last bit of HP is set +1 is added, if the last bit of HN is set -1 must be subtracted from the score. It is provable impossible that both last bits are set at the same time. However, it is possible that both bits are not set, indicating no distance change to each other.

The basic idea of Myers algorithm assumes the bit-vector lengths in the size of maximal one processor register of w=32 or w=64 bits that all operations can be executed on the hardware in $O(1)$. This restricts therefore the pattern length. For a general usage of Myers algorithm, not assuming the bit-vectors are suitable for one register, generic bit-vectors of any-desired lengths are needed. This work will present later a generic implementation of these any-length bit-vectors that are operating in $O([m/w])$ time, especially adjusted for the GPU usage, using meta-programming techniques and operator overloading for common accessors. For the following, the bit-vectors will be treated as “built-in” types.

### 2.2.1 Preprocessing the pattern

The state of the pattern is also represented and encoded in bit-vectors for the computation. They are used for the comparing state of $Eq$ at each cell. For each alphabet symbol a bit-vector or bitmask is held, each in the length of the pattern m, denoted with $PEq$. A bit in such a bitmask is set if and only if at one position of the pattern that corresponding alphabet symbol occurs. This concerns the space consumption of $O(\sigma [m/w])$ in Myers algorithm for pattern $P=p_1,\ldots,p_m$ with the alphabet $\Sigma$ of size $\sigma$ and the machine word size of $w$.

We assume that the underlying alphabet $\Sigma$ can be used in an indexed mapped manner. For example DNA with A,C,T and G references to ‘A’=>$0$, ‘C’=>$1$, ‘T’=>$2$, ‘G’=>$3$. This allows accessing an item of the array from the prepared pattern bitmasks directly by a symbol.

The following pseudo code shows the preparation process:

```plaintext
1 Bitvector $PEq[0] = \{ 0^0, 0^1, 0^2, 0^3, \ldots \}$
2 for $i = 1, 2, \ldots, m$ do
3 $PEq[\Sigma[P[i]]] = 0^{i-1}10^{m-i-1}$
```

In the beginning, all bitmasks $PEq$ are reset to zero. Next, the pattern $P$ is traversed and within the third step, the bitmask of the current pattern symbol is selected, and the bit is set at the current position to 1 for marking an occurrence. The preprocessing runtime needs $\Theta(m)$. The bitmasks needs $\sigma * m$ bits for its complete representations of the pattern.
Constructing an example for the preparation step:

Alphabet \( \mathcal{A} = \{0\Rightarrow A, 1\Rightarrow C, 2\Rightarrow T, 3\Rightarrow G\} \) \(|\mathcal{A}| = \sigma = 4\)

Pattern \( P = ATACAGACTG \) \(|P| = m = 10\)

Bitmask \( B[\sigma] \)

\( B \) contains bit-vectors in length of the pattern \( m = 10 \).

The array size is 4, one for each alphabet symbol.

Storing \( B \) need \( 4\times2 = 8 \) bytes, two bytes for each vector with 10 bits.

After preprocessing the pattern as input, the bitmasks \( B \) are initialized as followed:

\[
\begin{align*}
P &= ATACAGACTG \\
B[0 \equiv 'A'] &= 1010101000 \\
B[1 \equiv 'C'] &= 0001000100 \\
B[2 \equiv 'T'] &= 0100000010 \\
B[3 \equiv 'G'] &= 0000010001
\end{align*}
\]

2.2.2 Computing the edit distance

The next step after preprocessing the bitmasks \( PEq \) is to compute the edit distance with unit costs. The Myers original implementation is presented here computing the local Levenshtein edit distances between a sequence and a pattern.

Myers used a slightly different naming for the bit-vectors.

\( Pv \) and \( Mv \) are used for the positive and negative vertical delta values.

\( Ph \) and \( Mh \) are used for the positive and negative horizontal delta values.

\( Xv \) and \( Xh \) are used for the current vertical and horizontal column state for computing.

The input parameters the algorithm is the sequence \( T=t_1,\ldots,t_n \), pattern \( P=p_1,\ldots,p_m \), the alphabet \( \mathcal{A} \) and the value \( k \) for the differences as threshold. The complete algorithm is shown here in pseudo-code:

1. Preprocess \( PEq[\sigma] \) with \( P \) \(
2. \text{Bitvector } Pv,Mv,Ph,Mh,Xv,Xh,Eq
3. \text{Score} = m
4. \text{Pv} = 1^n
5. \text{Mv} = 0^n
6. \text{for } j = 1, 2, \ldots, n \text{ do}
7. \quad \text{Eq} = Peq[\mathcal{A}[T[j]]]
8. \quad \text{Xv} = Eq | Mv
9. \quad \text{Xh} = (((Eq \& Pv) + Pv) \& Pv) | Eq
10. \quad \text{Ph} = Mv | ~ (Xh \& Pv)
11. \quad \text{Mh} = Pv \& Xh
12. \quad \text{if } \text{(Ph} \& 10^{m-1}) \text{ then Score} += 1
13. \quad \text{else if } \text{(Mh} \& 10^{m-1}) \text{ then Score} -= 1
14. \quad \text{Ph} <<= 1
15. \quad \text{Pv} = (Mh \& \sim 1) | \sim (Xv \& Ph)
16. \quad \text{Mv} = Ph \& Xv
17. \quad \text{if Score} \leq k \text{ then}
18. \quad \quad \text{Matching at position } j
\)

\( \rightarrow \) Preprocess bitmasks for pattern \( P \)

\( \rightarrow \) Setup vectors with \( 0^n \)

\( \rightarrow \) start with distance \( m \)

\( \rightarrow \) initialize vertical delta values

\( \rightarrow \) get bitmask for sequence symbol \( j \)

\( \rightarrow \) compute current vertical and horizontal state

\( \rightarrow \) update horizontal delta values

\( \rightarrow \) detect score change

\( \rightarrow \) update delta values for the next column

\( \rightarrow \) report an occurrence
The method performs only 17 bit-vector operations per scanned symbol. The runtime is $O(n)$ in case that all bit-vector operations are done in $O(1)$ time and matching are recorded in $O(1)$ time. In general the processing time is $O(n[m/w])$. Seven bit-vectors are used in the computation and $\sigma$ bit-vectors for the pattern bitmasks. Each vector is in the pattern length of $m$. The total required memory is therefore $O((7 + \sigma)[m/w])$.

An example demonstrates the computed distance scores:

- **Alphabet** $\Sigma = \{A, C, T, G\}$  \( |\Sigma| = \sigma = 4 \)
- **Pattern** $P = TCG$  \( |P| = m = 3 \)
- **Sequence** $T = ATTATCGACGCA$  \( |T| = n = 13 \)
- **Scores** $S = 3233321012333$  \( |S| = n = 13 \)

### 2.3 Variations of Myers algorithm by Hyyrö and for GPU usage

Heikki Hyyrö has published enhancement of Myers original algorithm with a reduction of the needed numbers of bit-vectors. Additionally Hyyrö has advance the algorithm for computing the Damerau distance scores and adaptations to compute the matrix banded that results into some good side effects. ([Hyy01], [Hyy02], and [Hyy03])

Furthermore and based on these implementations of Hyyrö, the methods has been modified by the author for computing and executing them on the graphic card. First, conditional if-statements have been removed where possible during technical reasons of GPU execution, explained in chapter 4. Secondly, only the scores are stored, and the matchings are not been recorded or reported due to the parallelization concept of GPGPU. The matchings are just collected after executing a number of verifications and all scores have been computed completely. In favor to this design principle, only the number of hits is returned by the algorithm variations. Additional and optionally a cut-off mechanism is implemented that may stops and return early in the computations if the score reaches a certain point where it is not possible anymore to fall below the k error factor while proceeding.

Nevertheless, different initialization of VP can be used to change between local and global distance scoring. Setting $\text{VP} = 1^m$ uses local scoring and $\text{VP} = 10^{m-1}$ global scoring. An additional bitmask for the pattern matching can further be introduced, initialized with $1^m$ or $0^n$, for any or invalid symbols to match always or never.
2.3.1 The unbanded algorithm for computing the Levenshtein distance

The pseudo-code for computing the adjusted version of computing the Levenshtein distance:

```
1  <Preprocess B[0] with P>
2  BitVector D0, HN, HP, VN, VP, X
3  Score = m
4  numHits = 0
5  VP = 1^n
6  VN = 0^n
7  for j = 1, 2, ..., n do
8      X = B[ Σ [T[j]] ] | VN
9      D0 = (((X & VP) + VP) ^ VP) | X
10     HP = VN | ~(D0 | VP)
11     HN = D0 & VP
12     X= (HP << 1)
13     VP = (HN << 1) | ~(D0 | X)
14     VN = D0 & X
15     score += (HP & 10^{m-1}) - (HN & 10^{m-1})
16     scores[j] = score
17     numHits += (score <= k)
18     [ if (score > k + n-j) return numHits ]
19     return numHits
```

Hyyrö used the $D0$ bit-vector for representing the diagonal bit states. The bit-vector $X$ is used for temporary computing results, to omit repeating calculations. The if-condition for increasing or decreasing the current score has been replaced by an implicit equivalent additive operation at line 15. The distance scores are saved at each position into a pre-reserved array equal to the length of the sequence $n$. The hits, or number of positions where the score fall below the $k$ allowed differences, are counted and returned by the algorithm. This variation uses 17 bit-vector operations also, but it needs only 6 bit-vectors for processing one sequence symbol. The total runtime equals Myers original algorithm.

2.3.2 The unbanded algorithm for the Damerau distance

To compute the Damerau distance the preceding bitmask for the transposition of two adjacent symbols is necessary in the computation. This is done within the vector $Xp$. Only a few operational changes are needed to adapt the algorithm to this distance measure.

The following pseudo-code shows the adjusted version for calculating the Damerau distance:

```
1  <Preprocess B[0] with P>
2  BitVector D0, HN, HP, VN, VP, X, Xp
3  Score = m
4  numHits = 0
5  VP = 1^n
6  VN = 0^n
```

→ Preprocess bitmasks for pattern $P$
→ Setup vectors with $0^n$
→ start with distance $m$
→ and 0 hits
→ initialize vertical delta values
The method for computing the Damerau distance uses 23 bit-vector operations and need 7 bit-vectors for processing one symbol. The runtime stay the same as the computing the Levenshtein distance.

### 2.3.3 The banded algorithms for Levenshtein and Damerau Distance

Hyyrö has published methods to compute the scores in a banded way for the Levenshtein and Damerau distances with some interesting benefits based on Myers approach.

The basic idea is to band the calculations into a certain ε region like Ukkonen proposed by an error factor. Fortunately, the k-value can be taken to bind the region, for example to the next higher power of 2 of k+1. As the picture beside indicates, there is no need any more that the bit-vectors used for computations are in the length of the pattern. The sufficient size of the bit-vectors can be determined by only the width of the used band. Additionally, it makes usually no sense using more than 10%-20% of the pattern length as an accepted difference rate for finding similarities. Measuring the edit distance for more than k=32 differences is also not reasonable for a text or sequence comparison. In advance of accepting not more than k=32 errors, the size of the bit-vectors will be small enough to fit always into one machine register. The execution time for the bit-vector operations becomes all along O(1) using the banded version, independently to the length of the pattern.

The banded algorithms are separated in two passes, the diagonal and the horizontal run phase. In the diagonal stage the computation goes stepwise downward in the matrix until the lowest boundary is reached. Instead of starting with an distance score of m in the bottom left corner, the distance value is counted up during the diagonal run. Therefore, the

```c
7  for j = 1, 2, ..., n do
8     X = B[ ∑ [T[j]] ] | VN  → bitmask for text symbol j
9     D0 = (((~D0) & X) << 1) & Xp
10    D0 = D0 | (((X & VP) + VP) ^ VP) | X | VN  → compute current delta vector
11    HP = VN | ~(D0 | VP)  → update horizontal delta values
12    HN = D0 & VP  → store old pattern bitmask
13    Xp = X  → update vertical delta values
14    X = (HP << 1)
15    VP = (HN << 1) | ~(D0 | X)
16    VN = D0 & X
17    score += (HP & 10^m) − (HN & 10^m)  → adjust scores
18    scores[j] = score  → save score
19    numHits += (score <= k)  → increase hit if occurred
20    [ if (score > k + n-j) return numHits ]  → abort is score cannot fall below k
21  return numHits
```

The basic idea is to band the calculations into a certain region like Ukkonen proposed by an error factor. Fortunately, the k-value can be taken to bind the region, for example to the next higher power of 2 of k+1. As the picture beside indicates, there is no need any more that the bit-vectors used for computations are in the length of the pattern. The sufficient size of the bit-vectors can be determined by only the width of the used band. Additionally, it makes usually no sense using more than 10%-20% of the pattern length as an accepted difference rate for finding similarities. Measuring the edit distance for more than k=32 differences is also not reasonable for a text or sequence comparison. In advance of accepting not more than k=32 errors, the size of the bit-vectors will be small enough to fit always into one machine register. The execution time for the bit-vector operations becomes all along O(1) using the banded version, independently to the length of the pattern.

The banded algorithms are separated in two passes, the diagonal and the horizontal run phase. In the diagonal stage the computation goes stepwise downward in the matrix until the lowest boundary is reached. Instead of starting with an distance score of m in the bottom left corner, the distance value is counted up during the diagonal run. Therefore, the

```c
7  for j = 1, 2, ..., n do
8     X = B[ ∑ [T[j]] ] | VN  → bitmask for text symbol j
9     D0 = (((~D0) & X) << 1) & Xp
10    D0 = D0 | (((X & VP) + VP) ^ VP) | X | VN  → compute current delta vector
11    HP = VN | ~(D0 | VP)  → update horizontal delta values
12    HN = D0 & VP  → store old pattern bitmask
13    Xp = X  → update vertical delta values
14    X = (HP << 1)
15    VP = (HN << 1) | ~(D0 | X)
16    VN = D0 & X
17    score += (HP & 10^m) − (HN & 10^m)  → adjust scores
18    scores[j] = score  → save score
19    numHits += (score <= k)  → increase hit if occurred
20    [ if (score > k + n-j) return numHits ]  → abort is score cannot fall below k
21  return numHits
```

The method for computing the Damerau distance uses 23 bit-vector operations and need 7 bit-vectors for processing one symbol. The runtime stay the same as the computing the Levenshtein distance.

### 2.3.3 The banded algorithms for Levenshtein and Damerau Distance

Hyyrö has published methods to compute the scores in a banded way for the Levenshtein and Damerau distances with some interesting benefits based on Myers approach.

The basic idea is to band the calculations into a certain region like Ukkonen proposed by an error factor. Fortunately, the k-value can be taken to bind the region, for example to the next higher power of 2 of k+1. As the picture beside indicates, there is no need any more that the bit-vectors used for computations are in the length of the pattern. The sufficient size of the bit-vectors can be determined by only the width of the used band. Additionally, it makes usually no sense using more than 10%-20% of the pattern length as an accepted difference rate for finding similarities. Measuring the edit distance for more than k=32 differences is also not reasonable for a text or sequence comparison. In advance of accepting not more than k=32 errors, the size of the bit-vectors will be small enough to fit always into one machine register. The execution time for the bit-vector operations becomes all along O(1) using the banded version, independently to the length of the pattern.

The banded algorithms are separated in two passes, the diagonal and the horizontal run phase. In the diagonal stage the computation goes stepwise downward in the matrix until the lowest boundary is reached. Instead of starting with an distance score of m in the bottom left corner, the distance value is counted up during the diagonal run. Therefore, the

```c
7  for j = 1, 2, ..., n do
8     X = B[ ∑ [T[j]] ] | VN  → bitmask for text symbol j
9     D0 = (((~D0) & X) << 1) & Xp
10    D0 = D0 | (((X & VP) + VP) ^ VP) | X | VN  → compute current delta vector
11    HP = VN | ~(D0 | VP)  → update horizontal delta values
12    HN = D0 & VP  → store old pattern bitmask
13    Xp = X  → update vertical delta values
14    X = (HP << 1)
15    VP = (HN << 1) | ~(D0 | X)
16    VN = D0 & X
17    score += (HP & 10^m) − (HN & 10^m)  → adjust scores
18    scores[j] = score  → save score
19    numHits += (score <= k)  → increase hit if occurred
20    [ if (score > k + n-j) return numHits ]  → abort is score cannot fall below k
21  return numHits
```

The method for computing the Damerau distance uses 23 bit-vector operations and need 7 bit-vectors for processing one symbol. The runtime stay the same as the computing the Levenshtein distance.

### 2.3.3 The banded algorithms for Levenshtein and Damerau Distance

Hyyrö has published methods to compute the scores in a banded way for the Levenshtein and Damerau distances with some interesting benefits based on Myers approach.

The basic idea is to band the calculations into a certain region like Ukkonen proposed by an error factor. Fortunately, the k-value can be taken to bind the region, for example to the next higher power of 2 of k+1. As the picture beside indicates, there is no need any more that the bit-vectors used for computations are in the length of the pattern. The sufficient size of the bit-vectors can be determined by only the width of the used band. Additionally, it makes usually no sense using more than 10%-20% of the pattern length as an accepted difference rate for finding similarities. Measuring the edit distance for more than k=32 differences is also not reasonable for a text or sequence comparison. In advance of accepting not more than k=32 errors, the size of the bit-vectors will be small enough to fit always into one machine register. The execution time for the bit-vector operations becomes all along O(1) using the banded version, independently to the length of the pattern.

The banded algorithms are separated in two passes, the diagonal and the horizontal run phase. In the diagonal stage the computation goes stepwise downward in the matrix until the lowest boundary is reached. Instead of starting with an distance score of m in the bottom left corner, the distance value is counted up during the diagonal run. Therefore, the
distance scoring results are different for the diagonal processing stage than for the unbanded versions. Different matrix cells are evaluated as drafted in the figure above.

Furthermore, within the diagonal run phase, the pattern bitmasks can directly be initialized in each step. This skips the preprocessing part of the pattern, which was needed in the unbanded algorithms.

To realize one diagonal step, the bit-vectors for the delta values and the bitmask must be aligned accordingly. Going one step down in the matrix means and equals to shift all pattern bitmask 1 to the right and setup the last bit at the corresponding new position. The following pseudo-code shows the computing the j-th column in the diagonal run phase for the Levenshtein distance.

\begin{verbatim}
1  \textless Build the correct bitmask vector into } B[j] >
2  X = B[ \sum T[j]] | VN  \quad \text{bitmask for text symbol j}
3  D0 = (((X \& VP) + VP) \& VP) | X | VN \rightarrow \text{compute current delta vector}
4  HP = VN | \sim (D0 | VP)  \rightarrow \text{update horizontal delta values}
5  HN = D0 \& VP
6  <Update the appropriate cell value at the lower boundary >
7  VP = HN | \sim ((D0 >> 1) | HP)
8  HN = (D0 >> 1) \& HP \rightarrow \text{update vertical delta values}
\end{verbatim}

In addition, for the Damerau distance as followed:

\begin{verbatim}
1  \textless Build the correct bitmask vector into } B[j] >
2  X = B[ \sum T[j]] | VN \rightarrow \text{bitmask for text symbol}
3  D0 = (\sim D0) \& (PM << 1) \& (XP >> 1) \rightarrow \text{compute current delta vector}
4  D0 = D0 | (((PM \& VP) + VP) \& VP) | PM | VN \rightarrow \text{Storing previous bitmask}
5  XP = X \rightarrow \text{update horizontal delta values}
6  HP = VN | \sim (D0 | VP)
7  HN = D0 \& VP
8  <Update the appropriate cell value at the lower boundary >
9  VP = HN | \sim ((D0 >> 1) | HP)
10 HN = (D0 >> 1) \& HP \rightarrow \text{update vertical delta values}
\end{verbatim}

The matrix cell value at the lower boundary of the current position is increased by 1 only due to a mismatch of the symbols. This can be determined by the last bit state of } D0. The contained diagonal state is 0 if the distance increases, or 1 if not.

After processing the diagonal phase, the computed score could be used the check if a cutoff point has been reached, to abort early before running the horizontal processing phase.

The horizontal stage is computed in the same way as the unbanded algorithms, but are restricted inside the width of the band.
Finally a total runtime of only $O(n)$ is achieved with the banded versions, instead of $O(m + n\lceil m/w \rceil)$ time with Myers original. This is based on the assumptions that $k$ is below 32 and implicit preprocessing is done. Nevertheless the space consumption is suppressed to $\Theta(\sigma)$ or $O(1)$ for fixed alphabets, as we do not need the length of the bit-vectors in the size of the pattern any more. The banded versions will at most negotiate the disadvantages of the unbanded methods. First, it is independent to the pattern length, secondly the bit-vectors will fit into one machine register under certain assumptions and operate in $O(1)$ time always, and thirdly the preprocessing phase is no longer needed. A lower runtime and space consumption is both expected using this method.
3 Graphic cards and technical aspects of GPGPU

Developing applications that are designed to execute calculations parallel on the graphic cards need some knowledge about the technology behind it to utilize their full power and to understand the design restrictions of programming. This work will present the evolution of graphic cards and their technology leading to the modern GPU computing abilities.

3.1 Evolution of graphic and graphic cards

The idea of using graphic cards for computing was not intended originally while coming up with computer graphics. Primary this has been inspired by the evolution of 3D and 2D graphic processing and by the increasing hardware capabilities. It introduces a more general way of using the underlying computing power to parallelize programs on a modern graphic card besides classical graphic rendering.

The very first beginning of computer graphics goes back to the 50’s. At these times simple monochrome calligraphic CRT displays, based on Braun tube, could draw points and lines on a so-called vector monitor. Complex figures like symbols were compounded of separated line segments. These devices were rather slow, expensive, and unreliable for normal users.

In the 60’s, the first wireframe graphic abilities with only line drawing appeared that could simulate 3D-like graphics. Ivan Sutherland, a pioneer of computer graphics, showed in his PhD thesis from 1963 at the MIT the potential of that man-machine interaction. His presented Sketchpad allowed interactive designs on a vector monitor with a light pen as input device. This event marks for the most people the origin of computer graphics. Sutherland has invented many common algorithms that are still used in today current computer graphics, like line clipping and drawing. He also defined the basic interaction loop of first processing the user input, then updating the program state, displaying or refreshing the new graphics and then repeat this procedure.

In that time, special devices called display processors units (DPU) were used to display graphics. The computational power of the host computers were too limited and slow by the time as drawing could executed on it. Rather the host compiles a display lists or file and sends it to the DPU to safe the computing time.

Jack Bresenham, at this time a programmer at IBM, had already presented 1962 an algorithm of how to draw lines on raster driven monitors without using floating-point operations and complex operations with minimal rounding errors while discretizing the line.
Later he extended it to circles too. This introduces the techniques of rasterized graphics. Anti-aliased lines and curve drawing is a major topic by now in computer graphics. The full utilization of these algorithms took place at the end of 1960. Direct-View Storage Tubes (DVST) was invented, having a fine metallic lattice retaining the light at one position for a time. DVST in comparison to vector screens can show and keep graphics without a periodic redrawing. The modern way of drawing with raster graphic monitors is introduced with more powerful adapted hardware.

In the time between 1970-1980, the achievements let go from line drawing wireframe images to rasterized, filled polygons to create the illusion of real 3D image, intimately acquainted by shading mechanisms. The graphical devices had become an accessible frame buffer in memory for picture elements, the pixels, which can be directly mapped by the device into the monitor raster by refreshing in a frequent interval. First graphical applications and games were developed. However, real-time 3D graphics still require too high computing performance for the time. The math behind for the simulated 3D space, polygons, and objects grows further on. In addition, ISO standards for 2D and 3D started to be evolved with “GKS” in European and “Core” in North America. In 1971, Henri Gouraud presented a method for shading polygons, the Gouraud Shading. In 1975, Bui-Tuong Phong presented his methods of a better-colored shading technique, the Phong Shading, and was the first who present a global lightening model for computer-generated images, the Phong Lightning model, giving a good 3D illusion. James F. Blinn improved these algorithms to a local lightening model in 1977 for reflections on surfaces with the Blinn-Phong-Model. Texture mapping methods for polygons pioneered by Edwin Catmull in his Ph.D. thesis of 1974 were described.

Between 1980-1990 more and more colors are moving into the technic, cheaper and faster hardware are constructed, as well as methods for more 3D realism methods are researched. In the early 1980s, graphical output gained importance driven by development of CAD computer aided design programs (CAD) and Graphical User Interfaces (GUI), which got popular by Macintosh computers. In 1981, IBM presented the first color graphical device adaptor, it becomes a standard for personal computers as the graphical processing device. Rendering techniques, for example bump mapping and environment mapping by Blinn, smooth shading, shadow, and reflection mapping techniques were exposed. Unfortunately, they were still too slow for the underlying hardware for presenting and computing them in real-time.
Standard graphical APIs like OpenGL and DirectX for drawing and rendering were released in the 90’s. Newer and faster hardware could be manufactured with a growing amount of memory and hardware support for the implementations of texture mapping, blending operations, buffer manipulation, and other rendering techniques. Practical fast 3D CAD programs were developed and 3D modeling software appears like Maya, Blender3D, and 3D Studio Max. Completely computer-generated movies like “Toy Story” could be rendered. Graphic cards and PCs become affordable for most people, and even real-time 3D games were now possible.

With a continuously demand in 3D applications especially driven by the gaming industry and movie-making ventures, between 2000 and until 2010, more and more photorealism turned into real-time rendering with the massively improvements of the hardware performance for rendering and the shading methods. Further developments on the underlying standards for PC, for computer graphical devices and there APIs propagate the usage of high-end performing graphic cards to the end-users, as they are affordable for most people at lower costs. The PC hardware got rapidly more and more powerful and cheaper in all areas. Almost every household have at least one high performance PC in the industrial countries.

The first graphic cards with GPGPU programming abilities occurred at 2006 allowing developers to make use of the GPU power in a general computing way, beside the rendering math and techniques for 2D and 3D graphics. In 2010, NVidia® with its CUDA™ Technology and ATI® with its Stream™ Technology within OpenCL became the greatest contributors that actuate the development of these GPU computing technologies. (Historical references based on [HGO05], [Kna08])

Until now, applied researches and scientific applications take intensified usage of the computing ability from GPU with massive parallelization. Speeding up and adapting of algorithms and problems that are computational heavy are made, like cosmological space and fluid simulation, solving partial differential equation, computing physical and chemical reactions, and many other interdisciplinary area of informatics. New APIs for physical calculations with PhysX™, Ray tracing with OptiX™ and other simulation engine are development by NVidia using the computing power of GPUs.
During the evolution process, the number of colors in that time increases slightly from monochromic up to 16.7 billion today. More than 3 billion transistors resist on the hardware, allowing several GFlops of operational speed with huge amount of available memory. For example, the NVidia ‘Fermi’ technology from 2010 offers 512 cores with 1.5 GHz each, a memory bandwidth of 192 GBytes/s with a standard configuration of 1.5 GB of memory, with a computing power of about 1.5 TFlops to 2.7 TFlops, dependent on floating point precision, according to the declarations of the manufactures.

A brief comparison shows the performance differences between the current actual GPUs and CPUs offers:

<table>
<thead>
<tr>
<th>Device / Core</th>
<th>GFlops *</th>
<th>Bandwidth *</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVidia GeForce GTX 580</td>
<td>2.7 TFlops</td>
<td>192 GBytes/s</td>
</tr>
<tr>
<td>ATI Radeon HD 6970</td>
<td>2.7 TFlops</td>
<td>176 GBytes/s</td>
</tr>
<tr>
<td>Intel I7-980X</td>
<td>160 GFlops</td>
<td>25.6 GBytes/s</td>
</tr>
<tr>
<td>AMD Phenom™ II X6</td>
<td>65 GFlops</td>
<td>n/a</td>
</tr>
</tbody>
</table>

* taken from specifications from NVidia, ATI, Intel and AMD homepages

A brief comparison shows the performance differences between the current actual GPUs and CPUs offers:

<table>
<thead>
<tr>
<th>Device / Core</th>
<th>GFlops *</th>
<th>Bandwidth *</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVidia GeForce GTX 580</td>
<td>2.7 TFlops</td>
<td>192 GBytes/s</td>
</tr>
<tr>
<td>ATI Radeon HD 6970</td>
<td>2.7 TFlops</td>
<td>176 GBytes/s</td>
</tr>
<tr>
<td>Intel I7-980X</td>
<td>160 GFlops</td>
<td>25.6 GBytes/s</td>
</tr>
<tr>
<td>AMD Phenom™ II X6</td>
<td>65 GFlops</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 1 – GFlops and Bandwidth of GPUs and CPUs

The graphical chart above illustrates up to the middle of 2008 the enormous gap of the computing power between CPU and GPU.

For the future, there is no sign that the rapid growing of the past 10 years will stagnate, corresponding to follow the Moore’s Law, with increasing memory and hardware performance, approximately doubling the computing power every two years ([Moo65], [Sto06]).
3.2 The graphic rendering pipeline

Rendering 3D and 2D graphics usually is proceeded by perspective drawing of polygons in form of points, lines, triangles, or quads, built up by vertices, edges and form descriptions onto a 2D screen or monitor. The graphic card is responsible of processing the polygon data to the resulting pixels, in general by a pipelined dataflow conception. There exist other rendering techniques like ray tracing or real 3D drawing devices, but are by now relatively computational slow, not hardware optimized and out of scope of this work.

The rendering pipeline was primary a fixed staged process in the graphic card, the principal is shown in figure 14. Generally, the polygon data, computed or loaded usually in form of triangle vertex descriptions, are transferred to the graphic card pipeline as input. They are processed throw the pipeline in different stages and computed to the resulting outputted pixels for the screen. In detail, the inputs for the pipeline are vertex definitions of 2D or 3D space positions with attributes to textures, colors, normal vectors and other parameters, and geometric objects descriptions. For describing the world space environment and the perspective drawing, 4D matrices are used and set up for the perspective, the objects, the view, the global transformations, and further. Global and local light sources and other effects like fog may be configured in previous influencing the computations and the resulting drawn pixels for the scene.

The first stage in the pipeline is to transform the vertices to screen or eye coordinates by perspective division using adequate matrix operations with the given view matrices for displaying in 2D space. Adjusting the colors for per-vertex lightning and other effect states are applied within this process by the given input vertex attributes. Also clipping and culling operations are processed here to discard non-visible objects. The outputs of this stage are transformed, normalized vertices for the 2D screen viewport with prepared color states and a depth for drawing. The executed program (except the clipping and culling) is commonly denoted as a vertex shader. For each input vertex of a polygon, a vertex shader program is once applied to manipulate and prepare them as described. During the development, this fixed function of processing has been changed to a user-programmable function, that can be used to manipulate the vertex coordinates and attributes states to achieve certain special effects, for example deformations of objects or glowing.
The next pipeline step is to take apart complex geometry objects into single simple primitives, usually triangles, in the geometry assembler for process the drawing. The normal fixed process does clipping against the screen viewport, does necessary triangle constraint splitting and further, that are then handed over to the rasterizer. The later introduced user-definable *geometry shaders* can add or remove vertices and can generate new primitive assemblies. This can be useful for tessellating objects in respect to the view distance or depth of the object in much lower or much higher resolutions.

The rasterizer fills and shades out the geometric primitives by interpolating the values of color, texture coordinates, normal vectors, and other along the edges of the triangle to produce fragments, the potential pixels. The rasterizes generates for each polygon a number of the fragments for drawing with its position, depth, and color. Blending, texture mapping, per-pixel lightning and other buffer operations are applied in the rasterizer for each fragment. The executed program is called a *pixel or fragment shader*. Primarily this was also a fixed hardware function, which later has been evolved to a user-definable function, to manipulate the fragments in color and state for example for shadowing or material effects.

Finally, the fragments of the rasterizer are evaluated to the resulting drawing pixel and written into the frame buffer. To avoid overwriting and care the logical visibility of the objects depending of their depth, Z-Buffer techniques for culling and clipping on a per-pixel base are applied.

For writing those vertex-, geometry-, and fragment-shader programs, proprietary languages have been develop. For example HLSL used by DirectX and GLSL used by OpenGL API.

Due to the relatively independency of the graphical objects, primitives and the processing of the vertices through the pipeline, multiple processors are used to parallelizing the work. Each vertex and fragment is passed through the pipeline in the same way for one rendering step. These processors are highly optimized and usable only for arithmetic and floating-point operations on their purpose. The *vertex processors* are optimized for vector and matrix calculations. The *fragment processors* for color and value computations. These processors operate extraordinarily fast comparing to a CPU to compute billions of operations and process thousands of fragments, vertices, and primitives per second to achieve near-realistic real-time rendering abilities. But in seeming disadvantage these processors do not have the wide functional power of a CPU.
3.3 Modern GPGPU capable graphic cards

GPGPU stand for General-Purpose computing on Graphic Processing Units, often referred only to GP or GPGP. The vertex, fragment, and other processing cores described in the previous section are replaced for GPGPU by multiple common processors that adopt their tasks. This opened the possibilities for developers to access and use these processors in a more generalized way, besides computing the graphic pipeline with vertex, geometry, or fragment shaders. Of course, this is still available on GPGPU capable devices for graphical rendering. The weak point of having only narrow processors in favor of fast operational speed resist currently further on. This might change in the future technological developments to obtain more functional and powerful computing units.

GPGPU is often referred also as a grid computing technology, executing one single program several multiple times in a grid of threads.

3.3.1 Structure of GPGPU capable graphic cards

The architecture of a modern GPU is slightly different from CPUs as seen in figure 16. A CPU consists commonly of a large cache memory, one or more process controlling units, and one or more computing cores (ALU). CPUs are connected to a moreover large external memory (RAM). In contrast, a GPU is built up of many cores, each with a small cache, mostly there is a compounded memory for a number of cores, and several process controlling units for these cores and for the application flow. A large accessible memory is connected nearly to the GPUs on the graphic device.

![Figure 16 - Architecture Scheme of CPU vs GPU](image)

The NVidia manufacturer calls this design principle CUDA, the Computing Unified Device Architecture, while ATI calls this the Stream Technology. The general conception and idea of GPGPU is the same for ATI and NVidia, but they are using a different programming concept for realizing, terms for describing components, and other technical aspects that will follow on the next pages. Two main programming APIs are offered: OpenCL and the CUDA API.
3.3.2 The NVidia® CUDA™ Technology

An NVidia graphic card with the CUDA Technology is assembled in groups of streaming multi-processors (SM). Each SM is connected to the global device memory of the card. For operational usage, a memory block for constants is departed from the global memory.

Each SM consists of multiple streaming processors (SP), an instruction cache, and a control-processing unit. A small memory block for each SM exists that is used for local or common computing data, which is accessible concurrently by all residing SPs. This memory is referred as the shared memory in CUDA. One SM has further on a fixed limited number of registers for the operational usage, which are partitioned onto the SPs dependently of the program register usage. Each SP is used to process one or more threads. NVidia called this GPU design principle the Single-Instruction-Multiple-Threads (SIMT) technology ([CUD10]).

A thread execution manager unit (TEM) with instruction cache functionality is installed on top of the SMs, used to schedule and control the threads, processors, and application progress. The normal rendering pipeline flow is also controlled and simulated within this TEM in different execution steps. The first run is used for the vertex processing step, followed by the geometry assembly, and at least the rasterization process within the fragment processing is executes. Every single step uses the results of the previous operations. The global memory is used to store the geometry, vertex fragment, texture, and other data. The shared memory is loaded with the actual data that is needed for computations in the shader programs. Nevertheless, it is possible to execute the single steps in parallel for different objects, as the geometry data streams are independent to each other in its processing through the rendering pipeline.
The most technical parameters of NVidia graphic cards are architecture dependent. Varying hardware parameters are the available amount of global device memory, the cores and memories frequencies, and the installed number of SMs for examples. There are two major technology versions of CUDA that specified different hardware capabilities and standards. They can be obtained with the CUDA programming API by a developer.

A table of technical specifications is displayed below of different architecture versions. Some terms and the consideration will be explained later on in this work:

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y- dimensions of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block/SM</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y- dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z- dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8K</td>
<td>16K</td>
<td>32K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. Number of 32-bit registers per threads</td>
<td>124</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td>Device dependent, between 6 KB and 8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width for 1D texture reference bound to a CUDA array</td>
<td>8192</td>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width for 1D texture reference bound to linear memory</td>
<td>22&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width and height for 2D texture reference bound to linear memory or a CUDA array</td>
<td>65536 x 32768</td>
<td>65536 x 65535</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width, height and depth for a 3D texture reference bound to linear memory or a CUDA array</td>
<td>2048 x 2048 x 2048</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. number of textures that can be bound to a kernel</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width for a 1D surface reference bound to a CUDA array</td>
<td>n/a</td>
<td>8192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. width and height for a 2D surface reference bound to a CUDA array</td>
<td>n/a</td>
<td>8192 x 8192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. number of surfaces that can be bound to a kernel</td>
<td>n/a</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. number of instructions per kernel</td>
<td>2 millions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 – Technical specification for CUDA architectures version

Some fixed determinations can be made for these GPU hardware architectures. For example, each SM has the same number of SPs, but the number of SPs in a SM is dependent by the architecture version. The installed shared memory and registers in each SM is fixed for all SMs, but differs on the architecture also.
3.3.2 The ATI® Stream Technology™ and OpenCL™

Basically, the ATI Stream Technology is nearly the same as NVidia Technology but uses slightly different notations for the single components. The underlying OpenCL programming API is a cooperated development of AMD/ATI and NVidia. As the ATI Stream Technology is based on the OpenCL API, or vice versa, they are using the OpenCL annotations for descriptions.

One GPU Compute Device, e.g. a graphic card, consists of multiple Computing Units (CU). These units can have multiple numbers of Stream Cores (SC) that are handling a multiple number of Processing Elements (PE). The CUs equals the SMs, and the SCs the SPs in CUDA. The PEs represents the single threads. Each CU has a so-called local memory that equals the shared memory in CUDA. All CUs are connected to the global device memory.

Each SC has an instruction-controlling unit, a branch execution unit, and access to general-purpose registers that can be used by the single PEs or the working threads. It is not explicit known if ATI also uses an instruction cache. An Ultra-Threaded Dispatch Processor control and schedules the application flow and threads on top of the CUs like the TEM on CUDA devices. It is specified for OpenCL devices that all SCs within a CU execute the same instruction at each cycle following the SIMT principle, and are able to pipeline up to four threads or processing elements concurrently.

There does not exists a common specification of hardware capabilities for ATI graphic cards with the Stream Technology like NVidia CUDA cards, but the underlying hardware abilities and parameters can also be requested using the OpenCL API.
4 Using GPGPU

4.1 Parallel Computing

Developing for GPU and on the graphic cards means to use the massive amount of processors and their computational power for calculations, algorithms or for a program to parallelize them in many threads. In general, there are two strategies of using parallel computing. One possibility is to parallelize only parts of the algorithm or program itself that can be independently computed. An example is the matrix multiplication where each cell can be calculated individually. The second strategy is to run an algorithm or program multiple times concurrently with different data. For example, this is applied in the graphic pipeline for processing the geometries, vertices, and fragments with the shader programs. This principle is also called the Single-Instruction-Multiple-Data (SIMD) method. The adaption of the Myers algorithm for GPGPU uses the second strategy to parallelize the computing of many different distance scorings.

Parallelization is not suitable for all kinds of algorithms and problems. Dynamic programming algorithms often use precedent computed results in its recursion formulas for example. In consequence, it is difficult to parallelize and synchronize the process in itself. It is very inefficient that all residing cores have to wait for one core until the operations has finished. In these cases, the SIMD strategy should be the suggested way of parallelizing. Another unpractical example is if one algorithm uses a large amount of data in contrast by using only a small number of threads and parallelized parts. Memory handling, scheduling overhead and executing time latency may be greater than doing in directly on the CPU, or with multiple CPUs, rather than on GPUs.

All applications that are using multiple numbers of cores for parallelizing are subjected to the Amdahl Law [Amd67]. This law says that the performance and speedup gain of an application using multiple processors is not simply a factor of the number of used processors. For instance, a program running on a single CPU compared to running parallelized on a dual processor is generally not twice time faster. The Amdahl law specifically denotes that the speedup of application using parallelization is bounded to its sequential portions, and not by the numbers of processors used for the concurrent executing parts. The “peach” of an application depicts this, where the complete runtime of a program is a partition with a sequential core, the “pit”, and a parallelizable portions, the “pulp”. The factor of speedup with parallelization always just depends on the

![Figure 20 - The "peach" of a parallel Program](image-url)
percentage of the “pulp” in the application that is executed in parallel with a number of processor, compared to a serial execution of these parts. In fact, there must be sequential portions of an application logic needed for program invocation, initializing and/or pre- and post-processing of data that cannot run in parallel or in a concurrent execution. In addition, data management, synchronizing and scheduling must be considered as overhead in the runtime of the program to the sequential processing parts when parallelizing. The pit of the peach always exists and is a fixed sequential process, and only the pulp can be distributed onto multiple concurrent processes. Technically the execution latency can be increased by using multiple cores. Hence using more cores requires more distribution logic. Due to the additional overhead of the fixed sequential parts the benefit and speedup does not have to meet the expected factor when parallelizing a program. The complete runtime of a serial executed program cannot be used to simple predict the speedup factor just by multiplying the numbers of cores in general. The increasing factor is always lower than a multiple of the number of used computing cores.

4.2 Concepts of GPGPU

CUDA and OpenCL uses different terms for their technology, but its further commonly referred to the CUDA declarations. The implications to OpenCL and the ATI Stream Technology can be taken from the table in the appendix A.

The basic concept of GPGPU programming is based on a grid-computing manner, handling and executing thousands of threads inside a grid. Especially for GPU that means having one single program or algorithm, the kernel, that is executed multiple times in the grid threads. The computing grid is formed by blocks of threads, or work groups in OpenCL. The dimension of the grid can be one- or two-dimensional. Each of the blocks is organized into a one-, two- or three-dimensional net of threads, or work items in OpenCL.

The maximum number of threads in a block is hardware dependent. For CUDA architecture 1.x it is 512, for CUDA 2.0 this is 1024. Because of this limitation, only a countable number of combinations exist for the block dimensions that not exceed the boundary. For example, 512x1x1, 256x2x1, 128x4x1, or 8x8x8 will satisfy CUDA 1.x requirements for a block, where 64x64x1 with 4096 thread will exceed the possible number of threads in a block. The number of blocks in a grid is technically not limited, but a maximal range of currently 65.535 blocks in each dimension is specified. Therefore, a maximum number of 65.535 x 65.535 blocks each with maximal 1024 threads is applicable for a grid with CUDA 2.x abilities, which is more than 4000 billion of possible threads all together.
Each thread in the grid, more exact in the blocks, is identified by its block index and thread index, and can be determinate with the given dimensions of the grid and blocks inside of a kernel program.

Seven important abilities are not directly realizable or recommended by design for GPGPU programs and must be considered before writing or adopting an application:

1. no dynamic sized arrays, only arrays with a fixed length are allowed inside a kernel
2. no recursions, no endless programs, only straight forward programming
3. no pointers to functions
4. no direct break-and-continue semantic
5. no direct interactions with the user are possible (no std::cout, std::cin or print or other)
6. no (virtually) unlimited memory access and computing register usage
7. only a simple flow control, avoid conditional statements and heavy branching paths

As seen, it is not always possible and useful to adopt every algorithm on a graphic card due its computational complexity, the resource usage, and by the design restrictions. The GPGPU programming design cannot cover the full spectrum of functional range as a common CPU offers, and disallows further direct user interactions, that a developer must deal with. The ability of parallelizing parts of the algorithm anyway is one factor as states previously, but also the technical hardware and kernel program limitations must be respected. The circumstances of limited memory and registers for the threads reduce the possibilities of adoptions, which is even differently for each graphic card. It is possible that an algorithm exceeds the hardware limitations and cannot be compiled or executed. However, new developments and researches are made to bear down some design barriers that are briefly described in the perspectives at chapter 8.

### 4.3 Thread assignment and execution on the GPU

Basic understanding of how the graphic card schedules the grid and assigns the threads is essential to realize the best performance and to optimize a GPU program. Further knowledge of the thread execution is necessary to understand why certain GPGPU design restriction exists and which weak points must be considered. Bad implementations can annihilate the profits of using GPGPU.

The Thread Execution Manager, or Ultra-Threaded Dispatch Processor in OpenCL, cares about scheduling of threads, blocks, and the grid execution. Although that the fundamental scheme of thread distribution is clearly described, the manufactures hide some implementational details. The scheduling process and thread assignment of a grid on the GPU can be depicted by the following mechanism.
Each grid block is assigned to one Streaming Multiprocessor, or Compute Unit in OpenCL. The blocks are allotted consecutively to the available SMs on the graphic card until all the blocks of the grid has been fully processed.

Each thread inside a block is assigned to one Streaming Processor, or Stream Core in OpenCL. As there can be more threads assigned to block then physically exists SPs in a SM, the threads are grouped into a set of active working threads, called the warps or wave fronts in OpenCL. All threads inside a warp can concurrently be executed. For example, a CUDA card with 16 SMs and warp size of 32 can execute 512 threads simultaneously. These threads groups are consecutively invoked until a block is completely processed. Awareness of the warp size can be useful to optimize the threading performance. In addition to that, the warp size of a block is also higher than the number of physical SPs in a SM. The CUDA specification declares 8 SPs at each SM with a warp size of 32. It seems that NVidia graphic cards contain for each SP even 4 processing lanes as well. The OpenCL documentation clearly specifies these 4 processing elements inside a streaming core that can execute 4 instructions in parallel. This is not officially documented for NVidia CUDA devices and may only be the case for ATI graphic cards.

Moreover, the device can accelerate the warp scheduling process advantageously. If a warp or threads inside the warp execution must wait for accessing the memory, which is usually a case when accessing the global memory that need some cycles to acquire, other warps can be proceeded in advance. Knowledge of this scheduling strategy and avoid concurrent memory access can hide latency time and improve the performance of the execution.

The concrete disposals of scheduling the blocks, warps or wavefronts, or the thread execution planning are not directly specified. This is part of a graphic card specific behavior and driver and manufacture dependent.

Two important aspect of the execution and flow control of threads and the warps is only explicitly documented in OpenCL, the realization of loops and conditional if statements. In the stream cores of OpenCL compatible devices exists an instruction and flow control unit. Since NVidia card support OpenCL usage, there must be at least some equal underlying mechanism for CUDA devices.
The OpenCL specification noticed that if \( t \) is the time it takes to execute a single iteration of a loop, and within all threads in a warp execute the loop one time, except for one single thread that executes the loop 100 times, it takes 100t to execute that entire warp ([OCL10]). In consequence large varying loops should be avoided or grouped in previous, since no direct influence of the warps, blocks, and threads scheduling process is possible.

Further, the OpenCL specification explains that if a kernel program has two execution paths A and B, taking the same amount of time \( t \) to execute over a warp or wavefront, the total time of execution, if any thread diverges, is 2t. This factor is called the branch granularity of a kernel, which equals also the wavefront granularity. Branching is done by combining all necessary paths as a wavefront. If threads within a wavefront diverge, all paths are executed serially. For example, using a kernel that contains a branch in two paths, the wavefront first executes one path, then the second path. An important note to that is that even if only one thread in a wavefront diverges, all other threads in that wavefront execute the branch too. The total execution time is the sum of all paths ([OCL10]). That means in consequence, having many executional paths raise the execution time to the number of the divagating paths. Avoiding conditional branching statement like if is mandatory wherever possible.

### 4.4 Memory and variable usage on the GPU

There exist three different kinds of installed memory on a GPU with different access time and usage proposals for a kernel program. The data and variables used in a kernel are declared with certain labels, having different locations, lifetimes, and usage scopes. For the performance of a GPU program it is helpful to understand their behavior.

The global device memory is the largest, up to several GB, and slowest accessible memory for a kernel. This memory is mainly used to store general data on the graphic device for the computations, and even for the results. Usually data is first transferred onto the device, then the computations take place, and after that, the results are transferred back to the host RAM. Each block and thread, concrete the SMs and SPs, are connected to the global device memory through a loading and storing unit. As every thread can access this memory, the driver implements a locking and accessing scheme in that units to avoid critical overlap readings and writings of memory cells. Therefore, it takes some waiting cycles for a kernel to access the content and to schedule the global memory access. The programming APIs provides atomic operations for these concurrent accesses for better regulations. Data in the global memory have non-expiring lifetime from the viewpoint of a kernel. The program at the host must care about the allocations, deallocations, and the transfers. Experimental, non-overlapping read and write accesses on the global memory do not have a significant performance loss of waiting. It is possible by the OpenCL API to hint a memory block to have read- or write only access to improve their behaviors. Also prefetching of data from global memory is supported.
The CUDA and the OpenCL API offers both the functionality to allocate memory on the host RAM and on the global device memory. Methods for transferring, reading, and writing into these memories are provided. The allocation routines enable a special memory behavior that the data can be acquired as page lockable memory. Only these page lockable memories can be used to operate asynchronous within a stream for transferring or copying memory in background. They have the tendency to use longer reservation times. Dynamic management of data, for example growable and shrinkable arrays with large portions of data flow, should be aware of that behavior.

Departed from the global memory, a block of memory for constants is dedicated from that with the same premises for read-only data and variables.

The second memory in a graphic card is the shared memory. This memory is rather small, only up to several KB, but has a fast accessing latency. Shared memory is placed in each SM accessible to all SPs inside of one SM. Therefore, only a few cycles are needed for read and write accesses as it does not need strict locking schemes. This memory is used on the one hand for common data that can be used by all threads in a block, and on the other hand for local variables and data in one single thread. Usually for good performance, the threads of a block copy parts from the global memory into the shared memory, using them in their computations, and copy the results back into the global memory. In effect that all threads in a block shares this memory for their data and computation resources, this depends the number of threads that can be assigned for a block. For example, a CUDA card with 1.x ability having only 16 KB of shared memory where each thread uses 512 bytes only up to 32 threads are feasible for an SM to not exceed the limitation, and a maximal block size of only 32 that can be used.

The third kind of memory is a block of registers for the variables of a kernel used for computations, installed at each SM. The registers are distributed and reserved for each thread to their own usage and have no access latency. For each SM only a limited number of registers are available for the threads. For CUDA 1.0 architecture they are limited to 8192 registers per SM. Depending on the code and compiler settings a kernel have different register usages. This affects the number of assignable threads in a block and for a SM also. The CUDA compiler allows a register usage only up to 124 in architecture 1.x and 63 in 2.x for one thread. Using more registers will be swapped out on demand when compiling for architecture 2.x. Compiling on architecture 1.x and exceeding 124 register may fail and corrupt the kernel execution unpredictable.

Variables and arrays used in a kernel program are usually automatically assigned to a memory location. However, they can be declared using API specific modifiers. The following table describes the memory locations, scope, and lifetime for variables that can be declared.
Variables and data that used only locally in one thread are placed in the shared memory and are referred as local memory in CUDA or private memory in OpenCL. There are some differences of storing arrays in CUDA. Compiling for architecture version 1.x the array variables are automatically placed in the shared memory block. They have a fast access, but with a large size, this decreases soon the available number of threads in a block and may exceed the available memory. For version 2.x array variables are automatically stored into the global memory, negotiating the thread decrement in a block, but having an increased accessing time. Further, global memory pointers in OpenCL must be explicitly specified.

### 4.5 Best Practices for GPGPU

NVidia and ATI give several best practice directives for performance optimizations ([BGL10]). They should be minded in the implementations when using GPGPU, because of either technical issues or the design principles. In addition, some feasible programming tricks will increases the speed for GPU and CPU executions too. This work does not show all suggested directives in detail, focusing on the applied principles for the GPGPU implementation of Myers. Below are the most important points:

- Avoiding heavy mathematic operations such as division and modulo. Given a value $n$ in the power of 2, equal bit operations can be performed instead:

  A division of $i/n$ can be equally expressed by $i >> \log_2(n)$.
  
  A modulo operation of $i%n$ equally is computed by $i \& (n-1)$.

- Using inline functions and additionally forcing the compiler to inline. As for GPU programs all functions and codes are inlined by the compiler in the binary code by default, on CPU it gives an performance gain, depending on the largeness of the inlined code and the available instruction cache.

- Avoiding conditional statements and different diverging code paths. This yields the branch granularity during the warp executions and slows down the execution time. The GPU processors are primary designed for straightforward operational usage.
- Avoiding loops that differ largely in the number of iterations in different threads. This increase the wavefront granularity as described in 4.3 and lowers the execution time.

- Unrolling loops with a fixed size manually, using meta-programming techniques or by a compiler hint. This results in a good speedup for GPU and CPU. With meta-programming techniques, this is applicable to relative small loops with a fixed size of iterations only, as the GPU compilers do not allow recursion with an infinite depth. The developer can give a hint with the "#pragma unroll N" directive before the loop statement to advise the compiler to unroll the loop with N steps.

- Using meta-programmed linked lists for fixed arrays may result in better and faster accessing. The addressing scheme for array items is indirect, based on referencing the content by their item offsets from the current starting memory location. For example the i’th item of an `int` array will be located with `startOffset+i*sizeof(int)`.
  With linked lists, the item content is directly accessed and referenced. It can be loaded into a register directly that offers a faster operational usage as indirect addressing. A fixed linked list has the tendency to need slightly more registers on the GPU, and may increases the branch granularity for locating and accessing a specific list item depending on the implementation.

- Try to utilize the full hardware capabilities. Be aware of the warp or wavefront size and the installed SPs per SM to maximize the usage occupancy of the SMs. The warp size physically represents the number of concurrent executed active threads at one time.
  Using a block size of at least the warp size will fully occupy an SM. Using a block size in a multiple of installed SPs per SM will still occupy an SM good. Using a multiple of the warp size will achieve a good scheduling behavior. A lower or uneven number will decrease the level of parallelism, as the available cores of an SM are not completely used. Tracking the kernel resource usage is necessary to obtain an optimal configuration setup by their requirements. Setup the number of blocks in a grid to a multiple number of installed SMs will fully occupy the graphic card abilities. A lower number will not use all available SMs and waste capacities. Some user experiences are engaged here for an optimal setup of the arrangement of threads, between the block size and number of blocks.

- Try to use as less memory resources and registers as possible. The hardware only have limited resources of memory and registers, a higher consumption lowers the number of threads that can be assigned and computed in an SM, and decreases the level of parallelism.

- Using the shared memory for common data that is used by all threads within a block. Shared memory typically has much faster access and needs only some cycles to acquire compared the long latency time and many waiting cycles of accessing a global memory cells. Generally, common shared data is once setup or copied by one kernel thread in a
Using global memory for writing data, it should be aware of memory coalescing to result in faster access schemes and a decreasing level of latency. Additionally memory flags for read-only and write-only hints, if provided, should be considered.

Using build-in, hardware optimized variable types. In both APIs exists several predefined types that may have optimized hardware implementations. For example vectors types from one up to four dimensions. Modern graphic cards are using the RGBA notations for colors and XYZW vectors for 3D space locations, each with 4 components and 32 bits. These types are adapted for proper faster computations on the GPU cores.

Using asynchrony and streaming mechanisms for memory transfers and grid executions. Both GPGPU concepts of NVidia and ATI support asynchronous operations and methods for copying buffers and computing a grid in the background. This should be used to not block the CPU. Instead of busy waiting for completing a memory transfer or a grid execution the application can continue to run alongside with sequential parts.

Using a large amount of data for an algorithm often needs a reasonable buffering technique. Transferring one large memory block to the graphic card at once has much better performance than copying many small parts over the bus system. Page lockable memory, that is allocated over the programming APIs, has the disadvantage of a slow allocation time but can be used to operate asynchronous. If continuous reallocations and copy operations takes place this may sums up to a great loss of performance. Consequential, collecting data should be done with normal not page lockable host memory. This data should then first copied into page lockable memory on host RAM in one block, and afterwards this block is transferred completely at once asynchronous to the graphic card. This achieves the most suitable memory and transferring behavior when dealing with a large amount of dynamic data management.

4.6 Limitations and usage for GPGPU

Developing a program for GPU always means to deal with the impossibilities and restriction due to the design principles. Before implementing a program on GPU, developers must consider if an adaption is possible and convenient for a realistic satisfying result.

A first criterion is if parallelizing is anyway expedient, under the circumstances making use of a massive amount of parallel executions. GPU programs are best suitable for small programs that are demanded to be computed many times. Heavy long-time operational programs with
only a few times of executions will profit more on a multicore CPU environment. Algorithms that making use of a large amount of memory within only a small number of threads is not a good initial situation. Memory handling and transferring, scheduling and the grid execution time latency might be greater than doing it directly on the CPU or with multiple cores.

Secondly, the impracticalness of the GPU program as stated in 4.2 must be taken into consideration. No recursion, no infinite running programs, no dynamic arrays, no direct user interaction will exclude several demands of applications. Additionally heavy branching programs into several different paths during the flow control are not recommended. Even with different kernel programs, it will not be satisfiable, as only one kernel can be executed at one time.

Programmers should be aware of the efficiency for graphic cards. High optimizations are recommended due to technical and programming issues, otherwise the gain of using GPU may be perished in cases of bad realizations.

The hardware abilities itself limits the usable range of GPU programs. It is specified that up to 2 million instructions can be maximally used for a kernel code on CUDA devices. The number of used registers, the shared and local memory usage, and used constant memory for a kernel program qualify the parallelizing factor. They influence the number of assignable threads in a block, subsequently also the number of active concurrent working threads. The available resources are all hardware dependent and it is possible to exceed the installed amount that a kernel is not compilable or able be executed.

The global device memory is also a limited resource. Commonly the OS reserves and occupies a large amount in advance for graphics up to several hundred MB. Additionally, allocation of one data block is restricted to a maximal size. In OpenCL this is specified to at least a quarter of the available global memory or 128 MB. CUDA does not specify this directly, but more than 256 MB on a device with 1 GB is not allocable by experiences. However, dependent on the size of global memory data one kernel program used for processing, it is a limiting factor of parallelization. The number of total executable threads for one grid launch is restricted by the free available global device memory.

In conclusion, GPU programs should rather be used as extensions for existing programs or to advance algorithms in there computational speed, make utilization and profit of the massive parallelization abilities, instead of replacing casual applications.
4.7 CUDA™ and OpenCL™ principles

The starting point of developing with NVidia CUDA API and AMD/ATI OpenCL follows different design philosophies.

Programming with CUDA always means developing software for NVidia CUDA capable graphic cards only. Programming with the OpenCL API is designed for ATI graphic cards with their stream technology. Recently NVidia also supports OpenCL compatibility for their graphical devices. Some might claim that OpenCL is a good choice as all common systems and devices are supported, but this is not true for all use cases since this API does not offer certain rich programming functionalities a developer would demand.

The main difference between both design philosophies of GPGPU programs is their compilation and integration behavior. CUDA is mainly used to create offline-compiled kernels programs within the build step of an application integrated in the program code. The OpenCL principle is based on online-compiled kernels at runtime of the application by an extra source file. Both principles exclude each other and have some main advantages as well as main disadvantages.

More information about the CUDA SDK and OpenCL API is attached in the appendix point B.

4.7.1 Programming with CUDA

CUDA programs and kernels are normally developed inside a program than can be compiled like normal projects, but are wrapped by using the NVidia compiler. The API can simply be integrated into development platforms and inside the build process for projects. One advantage of using CUDA for GPGPU comes with the NVidia compiler and the offline compilation. It allows a seamless integration of the application and the kernel code. Type and functions parameter checks help to avoid consistent and structural errors in advance. The programming API itself is easier to understand and to use, especially in the beginning of developing GPU programs. Novices profits from the C-project-like behaviors to create a properly working applications. Runtime errors are better coverable and also debugging is supported through a CPU emulation mode. Another advantage is the ability to support standard C concepts of OOP, templating mechanisms, meta-programming functionality and more. These are important paradigms especially for the SeqAn library and its integration. One disadvantage of programming with CUDA is that only one graphic device can be used for computations at a time, even if there are more installed. However, that this might change in future releases of the API. Furthermore, the template functionality is by now not bug-free. Default templates arguments are not correctly parsed which is really harmful to some SeqAn implementations. This might be fixed in future releases too. Nevertheless, code changes always need a recompile of the complete project that can consume a vast time.
4.7.2 Programming with OpenCL

In contrast to CUDA, the kernel programs in OpenCL are out-sourced separately in a file or by a written source string that is loaded and compiled during the runtime of an application. This way of online compilations allows the separation of program logic and GPU kernel runtime code itself, and enables an easy exchange of the implementation. It supports multiple devices that can be used for computations. Unfortunately, the OpenCL programming languages does not support any templating functionality and only the basic C conceptions. Only rather simple classes, variable typing, and functions are possible. Further, using structures and classes in the application, that are also used in the OpenCL kernel must be implemented and aligned according to the kernel code again, as program code und kernel source is separated from each other. This raises the error-proneness to correct align and order the used structures and even to deal with different variable type sizes. For example, the integer type may be 32 bit long in applications program, but the corresponding OpenCL type is 64 bit. The online compilation principle prevents a validation of that and results into unspecific runtime errors. This includes the calling parameters for a grid launch, in precise for the entire kernel function. The OpenCL code compiler enables only a lazy evaluation at the compile time for the bindings. Error reporting is not developer-friendly for novices, frequently only a general error during the build process is messaged instead of reporting the concrete lines of code of contradictory terms. This happens even for syntax errors. That makes it difficult to wrap out errors especially in the beginning. Also the complete application may fail to run at all if the source is missing or incorrect. Additionally, a real step-by-step debugging service is not offered, as the code is previously not known at point of application compilation. Since the building step is done at runtime, it needs some sequential time to parse and compile.

4.7.3 Integrating CUDA and OpenCL

There is a functional gap of both principles as described in the previous sections. On the one hand, there is the demand of a changeable, extensible, and rich functional usage with focus of templating and a seamless project integration that is realizable with CUDA only. On the other hand, only OpenCL offers a comprehensive graphic device independent usage. The overall goal is to implement a maintainable state of Myers algorithm for the different distance measures and with any-length bit-vectors.

One solution is developing exclusively for OpenCL while abstain making use of the complexity and functionality of CUDA programs with templates and other language features. The naïve approach is to work out all variants, especially for the different bit-vector implementations, with different kernel source files to avoid extensive if statements for switching the specific variations in the kernel code. Moreover, a suspect code may have to
be designed to advance, replace and adjust the kernel code string or file in runtime by different types and values used to simulate flexible templating behaviors.

The other solution to support CUDA only, and therefore NVidia graphic cards only, is not satisfying. A possible accessible solution could be to use the NVidia compiler that creates the device *.ptx* code files and *.cubin* binaries in precompiled form for a concrete GPU architecture during the program compilation. These precompiled files could then be used for OpenCL and CUDA as source input. Indeed these precompiled files are supported and compilable by OpenCL, while NVidia stated to generate OpenCL compiler compatible interpretations. Unfortunately the kernel entry function declaration are experimental not OpenCL callable. This may change in future to bridge this leak. It would solve the problem to make use of higher language-level features and constructions CUDA offers, and supporting ATI devices as well.

A compromised workaround has therefore been designed during the diploma thesis to realize a maintainable state for both APIs. In primary, this work has been focused on a CUDA implementation to allow the most flexible, reliable deployment scheme, and a good adaption with templates and classes crossing with SeqAn library coding paradigms. Later a possible OpenCL implementation was requested. To support both APIs a common extendable interface has been developed. For OpenCL a nearly equivalent kernel source file has been added and adopted to the implementation. A trick has been used to deal with the different template arguments for configuration and settings to combine both principles with deductions. OpenCL supports at least the *#define* and *#if-else* preprocessor directives. This is used to switch between the different template manifestations. A pre-code is generated and added in before the OpenCL code by manually creating corresponding *#define* statements with the given template configuration. More details that are implementational are explained in the following chapter 5 and chapter 6.
5 GPGPU implementation for SeqAn

One focus of this diploma thesis is the implementation of Myers algorithm for GPU. It should meet the conditions to not be an outstanding closed program, applicable, usable, and extendable for wide range of users with different concerns in computational biology. The SeqAn Sequence Analysis Library fulfills these requirements. The arrangement was to integrated the GPU implementations into SeqAn. The SeqAn library is developed and maintained by the Freie Universität of Berlin under the bioinformatics department. The official webpage for this project is located at [http://www.seqan.de](http://www.seqan.de). The library releases, documentations, code repository and further readings is located there for developers and users, as well as the documentation for the GPU integration and the Myers implementation done in this diploma thesis.

SeqAn is a platform independent, open source C++ library of efficient algorithms and data structures for the analysis of sequences with the focus on biological data. SeqAn applies a unique generic design that guarantees high performance, generality, extensibility, and integration with other libraries. SeqAn is easy to use and simplifies the development of new software tools with a minimal loss of performance. [Citation: SEQ]

Before the SeqAn project, there was a lack of available implementations in sequence analysis, even for standard tasks. Implementations of needed algorithmic components were either unavailable or hard to access in third-party monolithic software products. Addressing these concerns, the developers of SeqAn created a comprehensive, easy-to-use, open source C++ library of efficient algorithms and data structures for the analysis of biological sequences. [Citation: BSA]

The SeqAn project was initiated by Andreas Döring and Knut Reinert in spring 2003. Döring was the major designer of the library, and Reinert is the overall coordinator for staffing and the research direction. The maintainer team of SeqAn was extended in 2006 with Tobias Rausch and David Weese, and 2008 with Anne-Katrin Emde. Beside the core team, several Ph.D. and M.Sc. contribute and enhance the content of the library.

The main design idea of SeqAn is to use the advantage of templates and meta-programming concepts for its efficiency. This design has been chosen because it offers the most powerful way of independence, fastness, scalability, and extensibility to the algorithms as the research of Döring has shown besides other classical approaches like object orientated programming.

Therefore, the integration of the Myers Algorithm and the GPU support into the SeqAn library should be following their design paradigm. Unfortunately, there exist some difficulties using GPU APIs that contradicts these principles. For example OpenCL does not support templates or extensible meta-programming, and OpenCL provide standalone code only. The NVidia CUDA compiler by now has some bugs working with templates with default template parameters and anonym declarations that make it harder to include it with the full SeqAn
library functionality. Even just using existing code cannot simply be taken to run with GPGPU.

One application, which is based on SeqAn, is the RazerS project, that could be integrate and benefit from the GPU implementation of Myers. The SeqAn core team maintains this project also and is available from the webpage at [http://www.seqan.de/projects/razers.html](http://www.seqan.de/projects/razers.html). The importance of RazerS is that second-generation sequencing technologies deliver DNA sequence data at unprecedented high throughput. Common to most biological applications is a mapping of the reads to an almost identical or highly similar reference genome. Due to the large amounts of data, efficient algorithms and implementations are crucial for this task. RazerS is an efficient read mapping tool that allows the user to align sequencing reads of arbitrary length using either the Hamming distance or the edit distance. This tool can work either lossless or with a user-defined loss rate at higher speeds. Given the loss rate, the approach guarantees not to lose more reads than specified. This enables the user to adapt to the problem at hand and provides a seamless tradeoff between sensitivity and running time.

[Citation: RSW]

The RazerS application fits into the use cases of the Myers algorithm, which is the currently best-known algorithm for finding matching with an appropriate error rate. Computing on GPU enables here a method for massive parallel-verified amount of the reads, using fast computational power of the cores, discharging the CPU for filtering, and can highly enhance RazerS runtime.

### 5.1 API independent integration

The first need of integration is to establish an infrastructure for GPGPU and to provide a common interface for SeqAn to use the CUDA, OpenCL and if necessary other platform APIs in a unified way. An alternative CPU emulation mode should be offered for testing and for environments where no GPGPU capable devices exist.

The main strategy to achieve this is to define common generalized GPU access methods and declare common usage types for each platform API. These methods may not support or work for each platform in the same way or make use of the full functionality range of the API, but fulfills basic usage requirements to give an independent access to the GPU. Also dealing with some leaks in the APIs is appropriated, as OpenCL for example does not offer a method to receive the current free graphic device memory, only the total size of the memory.

Classes that will make use of GPU are build up and should use mainly these common interface methods to attain maintainable objects without different implementations.
The process of binding the code to the different implementations platform methods is done by re-including these GPU classes for each platform again, based on the previously defined common access methods. This avoids implementations for each class on each platform in advance for its own adaptions, and reduces maintaining the overhead as only the interface methods have to be changed or maintained as well as only one single class implementation exists for each platform. For each programming API, all classes and methods are separated in their own namespace to avoid name clashes and to provide better accessibility.

For CUDA, all the classes and methods are encapsulated into the `seqan::cuda` namespace.

For OpenCL, all classes and methods are loaded into the `seqan::opencl` namespace.

For the emulations on CPU, they are placed into the `seqan::gpu_cpu` namespace.

Primary some fundamental common types and shortcuts are defined API specific for a simple and unified usage that are referenced in the appendix point C.

The basics about GPU programming is that functions have different declarations deciding their usage, whether they are accessible by the GPU device and/or by the host/CPU. Furthermore, variables have certain declarations to their memory location. The following table shows the common interface denotation for the different platform declarations to enable an API independent access:

<table>
<thead>
<tr>
<th>Function declaration</th>
<th>Description</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>__gpu_inline_func</td>
<td>Inline function</td>
<td>inline</td>
<td>inline</td>
<td>inline</td>
</tr>
<tr>
<td>__gpu_host_func</td>
<td>Function that is callable by the host</td>
<td><strong>host</strong></td>
<td>inline</td>
<td>inline</td>
</tr>
<tr>
<td>__gpu_device_func</td>
<td>Function that is callable by the device</td>
<td><strong>device</strong></td>
<td>inline</td>
<td>inline</td>
</tr>
<tr>
<td>__gpu_common_func</td>
<td>Function that is callable by the host and the device</td>
<td><strong>device</strong> <strong>host</strong></td>
<td>inline</td>
<td>inline</td>
</tr>
<tr>
<td>__gpu_kernel_func</td>
<td>Kernel entry function for a grid execution</td>
<td><strong>global</strong></td>
<td>“__kernel”</td>
<td>static</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Description</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>__gpu_shared_var</td>
<td>A variable in a kernel resisting in shared memory</td>
<td><strong>shared</strong></td>
<td>“__local”</td>
<td>-</td>
</tr>
<tr>
<td>__gpu_global_var</td>
<td>A variable in a kernel resisting in global memory</td>
<td>not needed</td>
<td>“__global”</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4 – Common GPU function and variable declarations

Note: As OpenCL code for the GPU is separated from the application code, loaded and compiled at runtime, there is no need to assign special function behavior in the application and to assign variable declaration, as this is done in the separated OpenCL source. GPU functions are further always inlined for the device by the compiler.
For each platform a specialized context object is provided. That object contains information and handlers for the device, hardware, memory and other controlling objects.

<table>
<thead>
<tr>
<th>Common object</th>
<th>CUDA specialization</th>
<th>OpenCL specialization</th>
<th>CPU specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_context</td>
<td>GpuContextCUDA</td>
<td>GpuContextOpenCL</td>
<td>GPUContextCPU</td>
</tr>
</tbody>
</table>

A global context object is appropriated in the implementation and can be used as the default context to assign with. This can be received using the provided `gpu_get_default_context()` function. Each context has common methods to receive information about the free and usable memory, maximal threads per grid launch, and other hardware states, if available. More information about these context classes can be obtained from the SeqAn documentation under the GPU section. Every object using the GPGPU is engaged to assign to a context object, which therefore will be bind to a certain device. That enables the ability to create one or more contexts and assign them to different objects for multiple GPGPU capable graphic devices installed on the system. The platform interface methods are then using this context object for its operations.

Some common objects are declared for memory, stream, and execution management:

<table>
<thead>
<tr>
<th>Common Object</th>
<th>Description</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_pointer</td>
<td>Memory reference</td>
<td>void*</td>
<td>cl_mem</td>
<td>void*</td>
</tr>
<tr>
<td>gpu_program</td>
<td>GPU program</td>
<td>CUmodule</td>
<td>cl_program</td>
<td>- (int)</td>
</tr>
<tr>
<td>gpu_kernel</td>
<td>GPU kernel function</td>
<td>CUfunction</td>
<td>cl_kernel</td>
<td>- (int)</td>
</tr>
<tr>
<td>gpu_stream</td>
<td>GPU stream object</td>
<td>cudaStream_t</td>
<td>cl_command_queue</td>
<td>- (int)</td>
</tr>
</tbody>
</table>

Table 5 – General GPU objects
Note: For the CPU emulation mode, only pseudo placeholders are used.

The first common methods are defined for streaming and synchronization routines on GPU, that are used to execute operations asynchronous like reading from and writing to memory and launching kernel functions.

<table>
<thead>
<tr>
<th>Streaming methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_create_stream</td>
<td>Create a new stream object</td>
</tr>
<tr>
<td>gpu_is_stream_finished</td>
<td>Test if a stream is completed</td>
</tr>
<tr>
<td>gpu_wait_for_stream</td>
<td>Wait for a stream to completed</td>
</tr>
<tr>
<td>gpu_release_stream</td>
<td>Release a stream</td>
</tr>
<tr>
<td>gpu_wait_for_device</td>
<td>Wait for the device to complete all its operations</td>
</tr>
<tr>
<td>gpu_sync_kernels</td>
<td>Synchronisation point for threads inside a kernel/block</td>
</tr>
</tbody>
</table>

Table 6 – Common GPU stream functions
Note: The implementation details of these common interface methods are left out in this work, as they are miscellaneous API specific and out of current interest. They can be individually read out form the corresponding programming APIs [OCL10] and [CUD10] or directly from the implementations.
The next common methods deal about memory handling on GPU. Memories are allocated into either the host RAM or the global device memory, and created as page lockable with the API functions, except for the CPU emulation.

<table>
<thead>
<tr>
<th>Memory methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_alloc_memory</td>
<td>Allocate memory of a given size, type and location</td>
</tr>
<tr>
<td>gpu_copy_memory</td>
<td>Copy memories, can work asynchronous, and cares about memory transfers between host and/or device</td>
</tr>
<tr>
<td>gpu_write_memory</td>
<td>Write into memory, can work asynchronous, and cares about host and/or device locations</td>
</tr>
<tr>
<td>gpu_read_memory</td>
<td>Read from memory, can work asynchronous, and cares about host and/or device locations</td>
</tr>
<tr>
<td>gpu_free_memory</td>
<td>Release and free a memory object</td>
</tr>
</tbody>
</table>

Table 7 – Common GPU memory functions

For allocating GPU memory, some flags have been defined to control their behavior. Methods are provided to receive the states of a memory object, notice in the appendix D.

<table>
<thead>
<tr>
<th>Memory flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU_DEVICE_MEMORY</td>
<td>Memory should be allocated on global device memory</td>
</tr>
<tr>
<td>GPU_HOST_MEMORY</td>
<td>Memory should be allocated on host RAM memory</td>
</tr>
<tr>
<td>GPU_READ_ONLY_MEMORY</td>
<td>Memory is hinted as read-only inside a kernel</td>
</tr>
<tr>
<td>GPU_WRITE_ONLY_MEMORY</td>
<td>Memory is hinted as write-only inside a kernel</td>
</tr>
<tr>
<td>GPU_RESERVE_ONLY_MEMORY</td>
<td>Memory is hinted as reserve-only</td>
</tr>
</tbody>
</table>

Table 8 – Common GPU memory flags

Note: Reserve-only do not copy or transfer data to the device. This is used for buffer objects that are only reserving memory for result for example to omit unnecessary transfers.

For loading a binary or source file, compiling a kernel, launching a grid the following common methods are implemented:

<table>
<thead>
<tr>
<th>Kernel methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_load_program_from_binary</td>
<td>Load a GPU program from a binary file</td>
</tr>
<tr>
<td>gpu_load_program_from_source</td>
<td>Load a GPU program from a source file (OpenCL only)</td>
</tr>
<tr>
<td>gpu_unload_program</td>
<td>Unload a program file</td>
</tr>
<tr>
<td>gpu_create_kernel</td>
<td>Create a kernel, referenced by a function name</td>
</tr>
<tr>
<td>gpu_set_kernel_arg</td>
<td>Set a kernel parameter argument</td>
</tr>
<tr>
<td>gpu_set_kernel_arg_size</td>
<td>Set the kernel parameter size</td>
</tr>
<tr>
<td>gpu_launch_kernel</td>
<td>Launch a kernel</td>
</tr>
<tr>
<td>gpu_release_kernel</td>
<td>Release a kernel</td>
</tr>
<tr>
<td>gpu_get_kernel_usage</td>
<td>Get kernel resource usage information¹</td>
</tr>
</tbody>
</table>

Table 9 – Common GPU kernel and program functions

¹ Note: GPU kernel resource usage information is available for OpenCL only.
A difficult part that cannot be unified is to start the grid computation. Invoking the kernel function requires a different mechanic for each concept. The CPU emulations just loop through all desired threads calling the entire kernel function. For that, these methods are mostly empty and placeholders. OpenCL programs are always loaded and compiled by a certain source file within the application, and the kernel is launched by setup all parameters manually and invoked using the provided methods. For CUDA applications there exist two alternatives for launching. The first usual way is to call a kernel inside and from the application code using the appropriated instructions syntax from CUDA. The other method is to use a binary file, either precompiled by the NVidia compiler as a .ptx device independent file on a hardware-abstracted code or written by hand. This equals the OpenCL methodic. Details of these mechanisms are nearer described in chapter 6.6.

Further methods are provided API specific to detect and initialize a device, receiving and printing information of the current graphic card abilities, and kernel usage parameters. They are noticed in the appendix point E. Moreover, some further OpenCL specific methods and structures must be implemented, to extend the API with general-purpose functions, listed in the appendix point F.

---

1 To obtain the kernel resource parameters a common structure is used:

<table>
<thead>
<tr>
<th>Common structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GpuKernelUsage</td>
<td>Kernel resource information of the shared, local, constant memory amount, register usage and max. threads per block</td>
</tr>
</tbody>
</table>

### 5.2 Basic GPGPU classes

Base on the previously defined common GPGPU interface methods, object classes are designed for a general usage in SeqAn. By design reasons these classes do not follow the SeqAn conception of global access functions, like the length() function of getting the length of an arbitrary string type. In most cases, this would have needed a fundamental refinement of the SeqAn functions declarations with the basic GPU function declarations. Neither existing algorithm methods like the already implemented Myers could be adapted. They are using a Pattern/Finder model that is mutually incompatible with the design of GPU programs, for example no recursions, no dynamic arrays, no return-and-continue semantic, different memory handling, flow control and other aspects.

The GPGPU integration instead is built on top of SeqAn, supporting the basic SeqAn types converted GPU representable. The implementation is supposed to be an extension of the SeqAn library, not a real integration, but try to use the templating conception of SeqAn.
The following briefly described basic classes have been developed. Detailed information of the classes and their functions can be found in the SeqAn documentation under the GPU section.

### GpuObject (gpu_context)
Base class for all GPU objects. Holds and is initialized with a reference to a `gpu_context` to associate the object with. All classes are engaged to extend from this class.

### GpuString <TType=char> (usage, gpu_context)
Encapsulates a normal character string typed by the template parameter `TType`. The class is constructed with the memory usage flags. Setter and Getter methods are defined to access or change the content, optional asynchronous within a `gpu_stream`.

### GpuAlphabet <TAlphabet, TType=uchar, TInvalidChar=255, TAnyChar=*> (gpu_context)
Wrapper class for SeqAn alphabets. The `TAlphabet` parameter accepts a common SeqAn alphabet like DNA, RNA, Iupac, or AminoAcid. The second `TType` parameter denotes the type of the alphabet symbols. The third `TInvalidChar` parameter specified the invalid symbol character to use, and `TAnyChar` defines the any symbol character.

The class holds one array internally on the device and one on the host, those they are simultaneously accessible from both the CPU and the GPU. The array to store the alphabet has the default size of 256. The given alphabet is mapped to their indices that the array buffer can be used as lookup table. For example DNA: ‘A’ => 0, ‘C’ => 1, ‘T’ => 2, ‘G’ => 3. All other unaccepted characters are set to the invalid character symbol.

A function is provided that adjusts a string along this alphabet to discard invalid symbols or map them to the any or invalid character. This is used to stunt on controlling a string in a GPU kernel for erroneous symbols, saving if-conditions.

### GpuArray <TItem> (usage, gpu_context)
A class for a dynamic grow-able array of items. The `TItem` parameter specifies the type of the array items. This class is constructed with the memory flags for either device or host usage and/or read-only or write-only.

The array grows doubling its size until 16.384 items has been reached, and then it will grow only linear further. This limit can be changed using the `GPU_MAX_ARRAY_GROW_SIZE` constant.

The class provides methods for adding, setting, and getting items singly or completely for better efficiency. Further methods for reserving, cloning, and changing the usage from host to device are offered. All memory function has the ability to operate asynchronous within a `gpu_stream` reference.
GpuBuffer <TData> (usage, gpu_context)

A class for a dynamic grow-able buffer of data blocks. The TData type defines the type used for data. This class is constructed also with the memory flags for device or host usage and further.

This buffer is used to add and receive several data blocks with an encountered length. The data blocks are stored in one large linear buffer and the locations/offsets of the data and lengths are stored in a separated indices buffer.

![Figure 23 - GPU Buffer Structure Scheme](image)

The first large buffer memory block stores several data items consecutively aligned. The indices buffer, a GpuArray, is used to store and find the location of the data item in the buffer. The used type for the indices is named as index_t. The GpuBuffer grows doubling its size, beginning with 1 MB of initial size until 16 MB of the growing size has been reached. Afterwards only in steps of 16 MB the buffer is resized. This is done by performance reasons to not continuously allocating a small memory size only, as new allocations needs to reallocate memory and copy the data that continuously take certain time, and to not exceed the buffer size to fast. The bounding values can be changed by common `GPU_MIN_BUFFER_SIZE` and `GPU_MAX_BUFFER_GROW_SIZE` constants.

Methods for reserving, cloning, and changing the usage for host and device are provided also, that again can operate asynchronous within a gpu_stream.

Note: The GpuArray and GpuBuffer class are restricted by the maximal allocable memory on GPU as described in section 4.4.

Different classes have been developed for realizing general buffering mechanism based one holding a GpuArray or GpuBuffer objects with different memory locations.

GpuDoubleBuffer <FrontBufferType, BackBufferType>

The double buffer class for two buffer class objects, either GpuBuffer or GpuArray with the same typing, specified by the template parameters.

The class is designed to be using for example the front buffer in host RAM, and the back buffer for global device memory, and allow to copy, clone or swap them with each other. All transfers are possible to operate asynchronous within a gpu_stream.
The triple buffer is an extension of the double buffer to provide a triple-buffering mechanism. The double buffer class is extended with an intermediated buffer, that usually is used to transferring data on GPU device.

This can be used for example to specify the front buffer in normal host RAM for collecting data, then first to copy the whole data at once into the transfer buffer which should be declared as pack lockable host RAM. Then recently that data is transferred onto back buffer in the global device memory, in advance asynchronous. This achieve an efficiently way to handle and transfer large data sets.

As an extension of the triple buffer class, the quad buffering class is enhanced with a delayed buffer, to provide a staged quad-buffering mechanism.

This class is useful when a large data amount is used in a chain of consecutive grid launches. Assuming the buffer are used to collect data and transferred to the device, in the same way the GpuTripleBuffer operates. Then it is possible that the computations on the GPU are running in the background using the data in the back buffer, while continuously adding new data further into the front buffer. Then, in the next grid launch step, the previous operations on GPU may not be finished. For that, the front buffer can be stored or directly swapped with the delayed buffer first, to operate further on instead of waiting for the device, and recently processing this buffer later.

### 5.3 Bit-vector classes for GPGPU

Myers algorithm is based on using bit-vectors for the computation and encoding of the columns of the distance-scoring matrix. In assumption, the number of bits used equals the length of the underlying pattern. Intendedly, these bit-vectors fits into the one machine register to operate fast and use the hardware parallelism of bit operations. For general purpose, to support arbitrary length of the pattern, an implementation of any-length bit-vectors is necessary. The GPU programming principles disallowed using dynamic arrays at runtime. Therefore, a template parameter is used to fix the number of bits in the incarnations. For an optimized implementation, the bit-vectors classes are differentiated between using a simple type, using an array, or using a linked list for storing the bits. Further special GPU optimized types like 4D vectors are used for the bit-vectors. Loop unrolling and optimized math is applied as further enhancements. Additionally these classes should offer a common equal interface to operate independent of the type. Nevertheless, a maintainable and changeable way should be suitable.
The following different classes are provided for the realization of different bit-vector types:

**GpuBitvectorSimple <numBits, TType>**
This class is designed for bit-vector lengths less or equal to the machine word size. A bit-vector is stored using one simple variable of the type `TType`.

The `numBits` parameter specified the number of bits to store and should not exceed the maximum number of `sizeof(TType) * 8` bits.

**GpuBitvectorArray <numBits, TType>**
This class is used for general any lengths bit-vectors.

A bit-vector storing the bits in an array of items of the type `TType`. The size of the array is determinate and fixed to `[numBits/sizeof(TType)]`

**GpuBitvectorList <numBits, TType>**
A bit-vector storing the bits in a fixed linked list of `TType` values.

The number of items in the list is fixed to `[numBits/sizeof(TType)]`. This class uses metaprogramming functionality to realize the list.

**GpuBitvectorX<numBits,TVector,TType> / GpuBitvectorXY<numBits,TVector,TType>**
**GpuBitvectorXYZ<numBits,TVector,TType> / GpuBitvectorXYZW<numBits,TVector,TType>**
A bit-vector storing the bits in a 1D, 2D, 3D, or 4D component vector, like (x,y,z,w).

The type `TVector` classifies the build-in vector type, where each component is a `TType`. These classes make use of the hardware optimized vector types on the GPU.

For example CUDA supports vector types like `ushort1...ushort4` or `uint1...uint4` each of them having x, y, z and/or a w component. CUDA registers are only 32 bit that allows only a vector with maximal 128 bits.

Details of the bit-vector objects and methods are further documented in the SeqAn documentation.

Generally, these classes are highly optimized for GPU and CPU usage, make using of less branching as possible, and forcing all functions to be inlined. This can be toggled by the `GPU_FORCE_INLINE` constant. Used loops for iterations are completely unrolled due to their fixed length, which can also be toggle with the `GPU_UNROLL_LOOPS` constant. The performance gains of this variants will be shown later in the empirical results in chapter 7.
To provide a common and intuitive interface for all these classes, they are implemented with same functionality and overloaded operators to handle these bit-vectors like built-in types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator &amp;</code></td>
<td><code>C = A &amp; B</code></td>
<td>Bitwise AND operation</td>
</tr>
<tr>
<td>`operator</td>
<td>`</td>
<td>`C = A</td>
</tr>
<tr>
<td><code>operator ^</code></td>
<td><code>C = A ^ B</code></td>
<td>Bitwise XOR operation</td>
</tr>
<tr>
<td><code>operator ~</code></td>
<td><code>B = ~A</code></td>
<td>Bitwise NOT operation</td>
</tr>
<tr>
<td><code>operator &lt;&lt;</code></td>
<td><code>B = A &lt;&lt; 1</code></td>
<td>SHL operator, only shift 1 is support during efficiency</td>
</tr>
<tr>
<td><code>operator &gt;&gt;</code></td>
<td><code>B = A &gt;&gt; 1</code></td>
<td>SHR operator, only shift 1 is support during efficiency</td>
</tr>
<tr>
<td><code>operator +</code></td>
<td><code>C = A + B</code></td>
<td>ADD operator, implemented carry-ripple-ahead</td>
</tr>
<tr>
<td><code>operator =</code></td>
<td><code>C = 1, A = B</code></td>
<td>Set Operator</td>
</tr>
<tr>
<td><code>operator []</code></td>
<td><code>A[n]</code></td>
<td>Get the n bit of the bit-vector, 1/true or 0/false</td>
</tr>
<tr>
<td><code>operator ()</code></td>
<td><code>A(n,x)</code></td>
<td>Set the n bit of the bit-vector to 1/true or 0/false. For a constant value of x the set/unset operation should be used which are more efficient, without an if.</td>
</tr>
<tr>
<td><code>set()</code></td>
<td><code>A.set(n)</code></td>
<td>Set the n bit to 1/true.</td>
</tr>
<tr>
<td><code>unset()</code></td>
<td><code>A.unset(n)</code></td>
<td>Unset the n bit to 0/false.</td>
</tr>
</tbody>
</table>

Table 10 – Common bit-vector class operators

The operations are implemented straightforward, for example the OR-Operation of two bit-vectors loops through the value items and “or” them each other sequentially. The ADD-operations are computed in a usual carry-ripple-ahead way, which adds the carry-over bit of the last addition to the next one. For efficiency reasons, the SHL and SHR operations are implemented to be shifted only by 1 for not-simple bit-vector types. This needs only to remember the last shifted out bit of an item to shift them into the next item at the first positions.

The applied interface offers a compounded usage regardless to the specific implementation of the bit-vector type, for example with A,B,C a statement like `A = (B+C) | (B&A) ^B` is possible for all classes.

In general, the user will not be concerned for a suitable instantiation with one of these bit-vector classes by the number of bits, by type, and even for the current GPGPU API. The common proxy object `GpuBitVector<numBits>` has been implemented to choose an optimal type for a platform by a given number of bits. This is realized with a meta-programmed if-statement by the following selection scheme.
The construction of this proxy object and the bit-vector classes allows a seamless integration of Myers algorithm with optimal typed, optimized any-length bit-vectors, using a common interface and applying them unified in different computations without certain needed specializations of the underlying GPU platform or class type.

Unfortunately, these classes cannot be used or adapted for an OpenCL implementation. The kernel code for OpenCL is separated from the project code by design. Templating functionality is not supported. A dedicated version for the simple bit-vector up to 64 bits is implemented in this diploma thesis, using a single variable for representation that fit into one machine register.  

To distinguish between the usage of GpuBitvectorArray and GpuBitvectorList class for larger bit-vectors, the constant `GPU_USE_LIST_FOR_LARGE_BITVECTORS` is used. The GpuBitvectorList implementation shows better performance only on CPU and a decreasing performance on GPU. To locate an item in the linked list by an index, it is necessary to use if-statements. This increases the branch granularity and raises the runtime on GPU. List item contents can be directly referenced instead of indirect addressing used by an array for the items. This enables a better mechanism on CPU.

Notice that CUDAs default registers size is 32 bit for operational usage, therefor only up to a length of 32 bits is used within the simple bit-vector class. All modern CPU supports 64-bit values as basic hardware types. For that reason, up to 64 bits are using the simple bit-vector class for CPU.

The diploma thesis was primary appointed on focusing the CUDA API. An adoption for larger any-length bit-vectors for OpenCL would have required further advanced programming efforts that were out of scope in this thesis. The main requirements and difficulties was to build up the common unified infrastructure and controlling logic for each API. Nevertheless, the k-banded algorithms can be use within the OpenCL implementation, where the size of the bit-vectors are determined by the ε environment range for the band, in assumptions that this range is not greater than 32 or either 64.

---

<table>
<thead>
<tr>
<th>numBits</th>
<th>CUDA API</th>
<th>CPU emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 8 bits</td>
<td>GpuBitvectorSimple(n, \text{uchar})</td>
<td>GpuBitvectorSimple(n, \text{uchar})</td>
</tr>
<tr>
<td>&lt;= 16 bits</td>
<td>GpuBitvectorSimple(n, \text{ushort})</td>
<td>GpuBitvectorSimple(n, \text{ushort})</td>
</tr>
<tr>
<td>&lt;= 32 bits</td>
<td>GpuBitvectorSimple(n, \text{uint})</td>
<td>GpuBitvectorSimple(n, \text{uint32})</td>
</tr>
<tr>
<td>&lt;= 64 bits</td>
<td>GpuBitvectorXY(n, \text{uint2}, \text{uint})</td>
<td>GpuBitvectorSimple(n, \text{uint64})</td>
</tr>
<tr>
<td>&lt;= 96 bits</td>
<td>GpuBitvectorXYZ(n, \text{uint3}, \text{uint})</td>
<td>GpuBitvectorArray/List(n, \text{uint64})</td>
</tr>
<tr>
<td>&lt;= 128 bits</td>
<td>GpuBitvectorXYZW(n, \text{uint4}, \text{uint})</td>
<td>GpuBitvectorArray/List(n, \text{uint64})</td>
</tr>
<tr>
<td>&gt; 128 bits</td>
<td>GpuBitvectorArray/List(n, \text{uint})</td>
<td>GpuBitvectorArray/List(n, \text{uint64})</td>
</tr>
</tbody>
</table>

Table 11 – Bit-vector class selecting scheme with the GpuBitVector proxy object
5.4 Using GPGPU in SeqAn

The integration process and using GPGPU in SeqAn should be as easy as possible for the user. To use the SeqAn GPGPU implementations only one header file has to be added that cares about the including of the classes for each platform by defined constants:

```
#define SEQAN_USE_CPU_GPU    // include Seqan CPU platform
#define SEQAN_USE_CUDA_GPU   // include Seqan CUDA platform headers
#define SEQAN_USE_OPENCL_GPU // include Seqan OpenCL platform headers
#include  "seqan/gpu.h"     // include Seqan GPU header
```

Note: Before including and using the SeqAn GPGPU header files, certain runtime libraries for the platforms must be added to the project linker, which are listed in the appendix point G.

For all enabled platforms, the SeqAn-GPU header re-includes all basic GPU types, classes, and the Myers implementation into the different namespaces `seqan::cuda`, `seqan::opencl`, and `seqan::gpu_cpu` of the platforms.

The next steps is to test and detect GPU devices and receive there usage contexts. Exemplary this can be done the following way:

```
using namespace seqan;

// test if CUDA device exists
bool has_CUDA_Device = cuda::cudaHasDevice();

// test if OpenCL GPU device exists
bool has_OpenCL_Device = opencl::clDeviceExists();

// a CPU device always exists
bool has_CPU_Device = true;

// get global CUDA context
cuda::GpuContextCUDA* cudaContext = cuda::gpu_get_default_context();

// get global OpenCL context
opencl::GpuContextOpenCL* clContext = opencl::gpu_get_default_context();

// get global CPU context
gpu_cpu::GpuContextCPU* cpuContext = gpu_cpu::gpu_get_default_context();
```

Then the device or devices for a platform can be initialized. First for CUDA cards:

```
// initialize CUDA device, and print device properties
if (has_CUDA_Device) {
   if (cuda::cudaInitContext(GPU_Arch_10, cudaContext) != cudaSuccess)
      has_CUDA_Device = false;
   else
      cuda::cudaPrintDeviceProperties(cudaContext);
}
```
Then for OpenCL supporting devices:

```cpp
// initialize OpenCL device, and print device properties
if (has_OpenCL_Device) {
    if (opencl::clInitContext(GPU_Arch_10, clContext) != CL_SUCCESS)
        has_OpenCL_Device = false;
    else
        opencl::clPrintDeviceProperties(clContext->clDevice);
}
```

The CPU emulation does not need any initialisation.

Now the GPU devices and contexts are intialized and ready to be used.

The different created device context objects can now be assigned to the GPGPU classes to associated them with to a certain graphic card.

Before finishing the application the different contexts should be released.

```cpp
// release CUDA context
#ifdef SEQAN_USE_CUDA_GPU
    seqan::cuda::cudaClearContext(cudaContext);
#endif

// release OpenCL context
#ifdef SEQAN_USE_OPENCL_GPU
    seqan::opencl::clClearContext(clContext);
#endif
```

For the CPU emulation no release step is required.
6 Myers Algorithm on GPU

The key to profit of GPGPU for Myers algorithm and its varieties is to schedule a vast amount of distance scoring measurements. This process is departed into the three main tasks:
1. Collecting and buffering the measurement data.
   This includes the sequences, the patterns, the configurations, the scores, and further.
2. Controlling, transferring, and executing the distance measures in a grid.
   This means computing the distance values with different scoring schemes on the GPU.
3. Determining and collecting the matchings by the resulting scores and the k-value

The main object for these tasks for managing, buffering, controlling the data flow, and to launch the computation on GPU is the MyersScheduler. Each distance computing configuration is supposed to be processed in one thread of the computing grid.

The implementation of Myers algorithm on GPU is extended and integrated based on the SeqAn GPU infrastructure and adaptations, illustrated in chapter 5, and the strict adherence of the design concept. For a maximum of performance, scheduling must conform to the memory and runtime qualities for different GPU devices and the different platform API capabilities.

First, to include the Myers implementation for GPGPU in SeqAn, a flag is used to enable the header files within the SeqAn-GPU headers inside a project:

```
#define SEQAN_USE_CPU_GPU    // use SeqAn CPU platform headers
#define SEQAN_USE_CUD_A_GPU   // use SeqAn CUDA platform headers
#define SEQAN_USE_OPENCL_GPU  // use SeqAn OpenCL platform headers

#define SEQAN_MYERS_ON_GPU    // enable Myers GPU headers
#include "seqan/gpu.h"       // include SeqAn GPU headers
```

For each platform, in this example for CUDA, OpenCL and the CPU emulation, the Myers classes and runtime methods are loaded into the corresponding namespaces.

Before going into the details of the scheduling and execution process for a consistent, adaptable, and changeable usage common type are defined, noted in the appendix point H. Changing these types is directly affecting the memory and buffer sizes used to store the data and the runtime behavior. If larger types are used, more memory is used, the global device memory exceeds earlier, and this leads to constraint the number of distance scorings executed at once. Copying and moving memory on and between the host RAM and the GPU device results into a slower runtime, as clearly more data needs to be transferred. Nevertheless, these common types help to avoid misalignments of types in the implementation and the different platform APIs. 1
There exists usage limitations of the measurement data for the Myers runtime on GPU, by different reasons which are explained in the ongoing chapters 6.5 and 6.6. These restriction are specified and can be adjusted by the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MYERS_MAX_SEQUENCE_LENGTH</td>
<td>16384</td>
<td>Max. allowed sequence lengths</td>
</tr>
<tr>
<td>MYERS_MAX_PATTERN_LENGTH</td>
<td>1024</td>
<td>Max. allowed pattern lengths</td>
</tr>
<tr>
<td>MYERS_MAX_BITVECTOR_LENGTH</td>
<td>1024</td>
<td>Max. allowed bit-vector lengths</td>
</tr>
<tr>
<td>MYERS_MAX_ERRORS</td>
<td>32</td>
<td>Max. allowed errors</td>
</tr>
</tbody>
</table>

Table 12 – Constraint constants for Myers implementation

The distance measurement data are validated in the scheduler against these maximal values. The values have been chosen to reach a well-defined compilation and usage behaviour and to not transcend certain, especially older, device architecture capabilities.

1 For CUDA and the use of in-project compiled GPU programs, the NVidia compiler can automatically check the functions and kernel call parameters types for validity. For OpenCL programs, that are lazy compiled at runtime, no direct type check for function or kernel parameters can be provided as the NVidia compiler does. For example, assigning and passing an input variable to a kernel function, but implement it inside the separated OpenCL code with a different type, yields into incorrect assigned calling parameters. In general, this only reports to be a faulty assignment, but not what exactly is mistyped. Secondly creating a memory buffer of a certain type, setting the kernel input parameters to use that buffer, but implement and reference inside the OpenCL code with a different type, may corrupt the kernel in runtime while referencing to the faulty typed memory pointer. This results into a general runtime error only, without directly knowing why this happened. Thus to prevent those typing errors, these named types are defined and used for each API implementation, according to the different platform types specifications, to obtain equally aligned and sized types, and maintain a consistent and fault tolerant usage of the whole project.

### 6.1 Myers Scheduler Instance

The scheduler class handles and controls the data and their computations, and is designed as the frontend interface to the user. The object instance has the following signature:

MyerScheduler<Alphabet, ScoringScheme, KernelVariant, BitvectorSizeHint=-1> (UsageFlags)

One of the common SeqAn alphabets like DNA, RNA, AminoAcid, and Iupac is assigned as first template parameter to the scheduler. This alphabet is used for all distance scorings computed within this instance. Internally the GpuAlphabet wrapper class is used. The wrapper class assigns an any and invalid character, if not given, to enable matching, mismatching, or discarding against arbitrary, non-valid symbols by default.
The second template parameter specify the used scoring scheme or distance measure type for the computations. Four different methods are currently implemented, and defined as tags:

<table>
<thead>
<tr>
<th>Scoring Scheme Tag</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Myers_Hamming</td>
<td>Hamming distance measurements</td>
</tr>
<tr>
<td>Hyyro_Levenshtein</td>
<td>Unbanded algorithm for the Levenshtein distance</td>
</tr>
<tr>
<td>Hyyro_Damerau</td>
<td>Unbanded algorithm for the Damerau distance</td>
</tr>
<tr>
<td>Hyyro_Levenshtein_Banded</td>
<td>k-banded algorithm for the Levenshtein distance</td>
</tr>
<tr>
<td>Hyyro_Damerau_Banded</td>
<td>k-banded algorithm for the Damerau distance</td>
</tr>
</tbody>
</table>

Table 13 – available scoring scheme tags

The scoring scheme implementations are applied based on H. Hyyrö advancements and modifications. The extensions and optimizations as described in chapter 2 for a GPU execution are used. The defined constant `MYERS_USE_CUTOFF` can be used to enable or disable the cut-off mechanisms in the scoring computation methods.

The third template parameter denotes the memory usage constraint for each kernel. Three different kinds of usage specifications can be chosen:

<table>
<thead>
<tr>
<th>Kernel Tag *</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local_Variant</td>
<td>Using local memory for bit-vectors and global memory for the data</td>
</tr>
<tr>
<td>Shared_Variant</td>
<td>Loading the alphabet buffer in shared memory for a concurrent faster access to the alphabet symbols</td>
</tr>
<tr>
<td>Global_Variant</td>
<td>Reserving the bit masks for pattern representation in global memory for each measurement, using more global space consumption with less kernel resource usage</td>
</tr>
</tbody>
</table>

Table 14 – available kernel memory variant tags

* Note: The memory tags only affect the CUDA and OpenCL runtime, as the emulation on CPU does not take care of specific memory locations.

The `Local_Variant`-Tag is the most effective one and might be used as default option. With this option, all bit-vectors are placed in local memory or registers and all distance measuring data remains in the global memory.

The `Shared_Variant`-Tag does not seem to be a faster alternative. This option copies the alphabet from global into the shared memory. Hence, synchronizing of threads in a grid block is demanded to suspend until the copy process has finished. The copy process and reading the alphabet from shared memory do not result into significantly better effect on the runtime at all. By experience, if all computing data is copied into the shared memory first results even in a decreased runtime. Copying and synchronizations need more time than the benefit from faster access of shared memory will achieve. The non-overlapping read-only access of data at the global memory is practical better. Further, as more shared memory is used, a fewer number of threads can be proceeded at once concurrently.
The Global_Variant-Tag is used to reserve and store the array of bitmask for the pattern representation in the global memory. This method consumes largely more global memory, leading to a decreased number of executable measurements at one launch. This variation achieves a possible larger number of concurrent executed threads inside a block, as less local and shared memory is used. The kernel memory usage mainly depends on the size of the used bit-vectors, mainly the bitmasks for the pattern representations. Using them in global memory will reduced the level of resource usage for one kernel, but will increase their access times.

The best variant choice is use case and hardware dependent and has to be considered experimental.

As fourth template parameter, a hint to the expected size of the bit-vectors can be specified in advance. This denotes a fixed maximal number of bits used and needed for computations, or -1 for automatic detection. The automatic selection scheme can be provided only in a best-effort way, nearer described in section 6.4. Giving a hint in previous, for example knowing a pattern lengths of 300, result in adjusted resource usage of the kernel function depending on that size. The automatic selection scheme would choose 384 as suitable bit-vector size, and dissipated capacities of memory and registers. This concern the number of threads assignable and concurrent executed in a block, which therefore will be decreased using an automated detection. Nevertheless, this parameter can be used to create multiple instances of a scheduler class to group between small and large patterns in previous. This achieves better performance when using unpredictable, random, vast divergent pattern lengths.

The scheduler is initialized within the constructor with usage flags to a user controllable behavior:

<table>
<thead>
<tr>
<th>Usage flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Myers_Scheduler_Count_Hits_Only</td>
<td>Count hits only, don't collect matchings</td>
</tr>
<tr>
<td>Myers_Scheduler_Reuse_Sequences</td>
<td>Do not clear sequences after usage</td>
</tr>
<tr>
<td>Myers_Scheduler_Reuse_Patterns</td>
<td>Do not clear patterns after usage</td>
</tr>
<tr>
<td>Myers_Scheduler_Reserve_Mem</td>
<td>Do not free reserved memory after usage</td>
</tr>
<tr>
<td>Myers_Match_Invalid_To_Any</td>
<td>Set invalid sequence and pattern symbols to the 'any' symbol, otherwise invalid characters are discarded</td>
</tr>
<tr>
<td>Myers_Match_Any_With_One</td>
<td>Match 'any' symbols to 1, meaning 'any' matches to all other symbols, else 'any' matches with no one.</td>
</tr>
<tr>
<td>Myers_Scheduler_Auto_Launch</td>
<td>Auto-launch verifications on demand</td>
</tr>
<tr>
<td>Myers_Compute_Matchings_on_GPU</td>
<td>Enable collect matchings on GPU (else never)</td>
</tr>
</tbody>
</table>

Table 15 – Myers Scheduler usage flags
The default behavior is to assign invalids characters in sequences or patterns to the ‘any’ symbol, to no deallocate memory after usage for sparing reallocations in further use, and to enable the collecting of the matchings even on GPU if this is advantageously.

### 6.2 Using Myers Scheduler

To use Myers algorithm implementation on GPU in a SeqAn application, an easy-to-use applicable scheme has been designed for the user to handle the main tasks of adding data, launching the computation, and gathering matchings.

Previously the user is responsible to setup and select the preferred GPU platforms, as demonstrated in chapter 5.4. After initializing a scheduler instance, the user adds and composes the data for the distance measurements into the scheduler. After injecting all data and finishing all computations the user is able to receive all the resulting matchings and post-process them.

The main usage loop is figured here in pseudo-code:

```plaintext
1   scheduler.reserve (<numMeasurements>,<numSequences>,<numPatterns>)
2   while ( <moreData> ) do
3       MyersId seqId = scheduler.addSequence(<Sequence>[,sequenceMarker])
4       MyersId patternId = scheduler.addPattern(<Pattern>[,patternMarker])
5       scheduler.addMeasurement(seqId, patternId, allowedError)
6       if (scheduler.mustLaunch()) // test if buffers are full
7           scheduler.launch(false) // execute computations asynchron
8     end
```

The first depicted statement is optional. It is used to hint the environment to pre-reserve memories and saving runtime for allocations in the progress, when knowing the number of measurements, sequences, and/or patterns in previous.

The main loop first adds sequences and patterns data at line 3 and 4. A unique marker can be given by the user to recover the pieces in the matchings later. The strings are adjusted internally according to the specified alphabet, to discard invalid characters or map them to the ‘any’ symbol. This is done to prevent non-valid symbols in the scoring functions. This achieves a better GPU performance, as conditional controlling if-statements in the kernels are avoided.

Afterward the user is engaged to combine one sequence with one pattern for a distance computing with an accepted difference rate, the k-value, on line 5. In that way, the setup of the computing data is independently decided by the user and enables multiple usage schemes. It is possible to set N:1, 1:M and/or N:M configurations, where N is number of sequences and M the number of patterns.
In each loop step, after adding an distance measuring configuration the scheduler can check with the provided `mustLaunch()` method if computing the current collected data is necessary. If this is the case the scheduler must start the computation immediately with the `launch()` command to proceed the adding process. This could be automatized also, setting the `Myers_Scheduler_Auto_Launch` usage flags of the scheduler. The launch signal is detected while adding and collecting the data into the buffers of the scheduler. Internally the scheduler takes care of the used memory of the buffers. Tracking the free global memory on the graphic device as well as the host RAM memory is necessary to not exceed the usable amount for a GPU launch. This also includes observing the current number of measurements, and therefore grid threads, that are optimal suitable for a device execution dependently on the hardware. The launch process tries to operate asynchronous wherever possible, detailed described in section 6.4, to not block the CPU.

After all data has been exposed in the main loop, the scheduler is advised to a final launch, finishing all the left open computings in the buffer chain. This process includes waiting until all matching results are collected. Afterwards the user can finally acquire and process the resulting matchings as presented by the following pseudo-code:

```c
< Main computing loop >
scheduler.launch(true) // execute all left open computings
numMatchings = scheduler.getNumMatchings() // get the matchings data
MyersMatching* matching = scheduler.getMatchings()
< Process matching data >
```

For more information about the concrete method declarations, parameters and structures used, the reader may refer to the SeqAn documentation and examples.

### 6.3 Scheduler buffering and data handling

Handling, storing and transferring the large amount of data, for sequences, pattern, scores, configurations, and matchings, needs a reasonable buffering method with a good performant usage in the runtime. Several storage buffers are used for the data. Each of these storage buffers is departed into different buffer layers with allotted purposes, depending on the buffering mechanism used.

The common denotation and behavior of these layers are specified and listed below:

- The front buffer layers are capable for collecting the input data.
- The transfer buffer layers are suitable for moving the data mostly onto the graphic device.
- The delayed buffer layers are used for intermediate buffering of data.
- The back buffer layers are utilized for the computations on the device and for storing results.
Three different kinds of buffering mechanism have been developed to enable certain use case scenarios. The mechanisms are extended and placed on top of each other. The user can switch the method used by the scheduler with the `MYERS_BUFFERING_METHOD` constant with following available values:

- **MYERS_DOUBLE_BUFFERING** - a double buffering methodic:
  Useful if the number of distance measurements and the size of sequences and patterns used are known in advance. Buffers could be previously reserved, avoiding unnecessary reallocations while adding data. Using one host front buffer and one device back buffer which are directly copied into each other, avoids overhead of moving data between the buffer layers in this case. This method is not useful in cases where consecutively multiple launches are done, as the front buffer is occupied during the copying process preventing to proceed the program with further data processing.

- **MYERS_TRIPLE_BUFFERING** – a triple buffering methodic:
  In advance to the double buffering, an extra buffer is used to transfer the data to the device memory. First, the front buffer is mirrored into the transfer buffer and then transferred into the device back buffer. The advantage of that is that the transfer process can be done asynchronous. This allows a continual adding of data into the front buffer, while transferring and computing on the device is proceeded in background, without blocking the CPU and occupying the front buffers. This is useful in cases where producing or sustaining the data is slower than the time needed for computations on the GPU.

- **MYERS_QUAD_BUFFERING** – a quad buffering methodic:
  In addition of the triple buffering, a fourth buffer is introduced to delay the grid launch if already computations are running and occupying the GPU device. In this case, the current front buffer preliminary is stored to the delayed buffer to immediately continue the application process, instead of waiting for the computations to finish. This is useful for all cases where a high amount of throughput of data is needed, without accepting too much time of blocking.

The most efficient processing in general, where neither the number of distance measurements, nor the number of sequences and patterns are known, is the quad buffering methodic. Thus, this method is set as default value.

For realizing the different buffering mechanisms, where the certain buffer layers have different usage attributes and memory locations, some convenient classes are designed. These classes are based on the dynamic growable SeqAn GPU buffers and array classes described in chapter 5. They have been modified and extended in their handling methods for a simpler and customized usage to copying data between the layers.
This scheme of buffer declarations has been chosen to achieve the best performance and throughput results. Layers marked with “host” are placed in RAM memory. Layers marked with “device” are located in the global GPU device memory. Normal, not page lockable RAM memory using `malloc()`, has significant faster attitudes of allocation and copying/writing. In general, continuously adding data requires continuously reallocation and copying of the buffer, as these buffers grow automatically in its size, if not pre-reserved. Therefore, the front buffers are usually using the normal not page lockable memory for the best collecting performance.

To enable asynchronous operations and memory transfers page lockable buffers must be used. The transfer buffers used for moving the data on the GPU are dedicated to be page lockable. These buffers are located in host memory also, as copying from the not page lockable front buffers to the page lockable transfer buffers in one block is suitable fast in RAM only. Otherwise a transfer to a device buffer over the bus system must be performed which is rather slow, and cannot run asynchronous when the front buffers are not page lockable and will obviate block the program. The actually fastest way to transferring data from host memory to the device memory is by using page lockable memory. Both, the transfer buffer on host and back buffer on device fulfill this requirement, and are able to move the data asynchronously in advance.

Nevertheless, if quad buffering is used, instead of copying front buffer data to the delayed buffer, a fast swap method is used for a better performance.

The scheduler uses multiple storage buffers, each with the different layers, to store and manage the computation data. These storage buffers are marked with special usage flags for better performance efficiency at runtime, if supported by the underlying API. Depending on the buffering method one of the corresponding specialized Myers buffers is used for storing, for example using MyersDoubleArray/Buffer for double buffering.
The single storage buffers, their usage flags, and data type are listed as followed:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Usage</th>
<th>Storage Type</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequences</td>
<td>Read-Only</td>
<td>Buffer</td>
<td>MyersString</td>
<td>Sequence data buffer</td>
</tr>
<tr>
<td>Patterns</td>
<td>Read-Only</td>
<td>Buffer</td>
<td>MyersString</td>
<td>Pattern data buffer</td>
</tr>
<tr>
<td>Configs</td>
<td>Read-Only</td>
<td>Array</td>
<td>MyersConfig</td>
<td>Measuring setups</td>
</tr>
<tr>
<td>Scores</td>
<td>Reserve-Only</td>
<td>Buffer</td>
<td>MyersScore</td>
<td>Resulting distance scores</td>
</tr>
<tr>
<td>Bitvectors</td>
<td>Reserve-Only</td>
<td>Buffer</td>
<td>void*</td>
<td>Buffer for bitmasks</td>
</tr>
<tr>
<td>Matchings</td>
<td></td>
<td>Array</td>
<td>MyersLen</td>
<td>Matching counters</td>
</tr>
<tr>
<td>Matchings</td>
<td></td>
<td>Host - Array</td>
<td>MyersMatching</td>
<td>Matchings Positions</td>
</tr>
</tbody>
</table>

Table 17 – Myers storage buffers and their declarations

Sequence, pattern, and configuration buffers are marked as read-only as they are not changed during a computation of the distances.

The buffer to store the distance scores and bit-vectors are marked with the reserve-only flag. This specifies to not copy the buffer content when reallocating memory, for example while growing the buffer dynamically. Further, no data content is transferred onto device to lower the amount of overall transferred data. Since the scores and bit-vectors are not needed to be initialized or filled with certain values, hence in the computation phase the score are initially determined, this achieve an optimization to saving processing and copying time within this buffers. Therefore, only the necessary memory reservations are made with this flag.

Adding configuration data, sequence, or pattern to the scheduler will automatically care about the preparations, reservations, and insertion into the corresponding storage. While adding a distance measurement setup to the scheduler, the score buffer reserves memory for each configuration in the length of the corresponding sequence. The bitvector buffer is only used within the global memory usage variant. For each configuration, the array in the size of the alphabet of pattern representational bitmasks is reserved in this case.

The matching-counter buffer is used to hold one counter for each distance computation to count the number of matching accordance. The buffer for matchings is a host-only buffer that is filled after computing the scores, during the collecting process, according to determined distance scores and the allowed k-differences. This buffer holds all the detected matchings of all processed measurements, receivable for the user.

Write-Only buffers are not used since all buffers might be used for reading in a kernel.
6.4 Scheduler workflow

The general workflow of the scheduler is a pipelined process, dependent on the used buffering method. The scheme is outlined for the quad buffering mechanism as followed:

![Diagram](image)

**Figure 24 - Scheduler processing scheme**

The front buffers, allocated in normal host RAM, are filled continuously with the data, until a launch signal has been given. At the point of the first launch call, the data of the front buffers are copied all over at once into the host page lockable transfer buffers and cleared after. This process is completed rather fast and does not consume significant runtime even with larger amounts of data by experience. Next, a stream is created to proceed the followed steps asynchronously. The transfer buffers are advised to be moved into the device back buffers memory in background within the created stream. Afterwards the kernel for computing the distances is started, assigned to the same stream used for transferring. This enables to complete the whole transfer and computing stage alongside. As the stream starts executing the tasks in the background, the control is given back to the application. This allows to work and fill the buffers further on as soon as possible, not blocking the CPU for processing any longer.

At the point of the second launch call, before starting the next round of distance computations on the GPU, the matchings are collected based on the previously computed distances. In the computation process, the distance scores were computed and stored, and additionally the number of hits, the amount of occurrences where score <= k, is detected. The collection process evaluates the scores and stores the determined matchings for each position in a certain presized matching buffer. This process can be executed either by the CPU or by the GPU device with an appropriate kernel function. Both methods are suitable in a certain use case, having different advantages and disadvantages. The decision principle is commented after describing the general workflow.

Beside the process of collecting the matchings, the pending data are treated the same way as the first launch workflow operates. Except if the GPU has not finished the previous computations. In this case, as the created stream is still ongoing, the front buffers are stored temporarily into the delayed buffers by fast swapping. Immediately is returned to the application then, and the computation is suspended. Subsequent in the next launch step, the
delayed buffers are used to copy into the transfer buffers first, and the new pending front buffers are stored into the delayed buffers again. If in a followed launch call the stream is operating and the delayed buffer are already filled, the scheduler must wait to resume until the device has finished, as the complete buffer chain is occupied. The matchings must always be collected after the computations has finished, clearing the back buffer after, to release the buffer chain for continuing operations.

On the last final launch call, all residual measurings are constrained to be computed. The complete buffer chain is worked off consecutively for that. This process consumes the most runtime, as this need to wait until a computation has finished, collecting the matching, and moving on with the next pending buffer. Resuming to the application is inconceivable until all distance computing and matchings are processed and gathered. The user demands the complete list of results after the final launch call.

The decision principle at the launch time can be described with the following flow chart:

![Scheduler Launch Process Scheme](image)

In experiments, this proceedings reveal to have the most efficiently throughput with lowest latency that give back control to the application as fast as possible, blocking the CPU as less as possible. In the cases where an indefinite amount of distance measurements is produced, with multiple consecutively launch executions, the quad buffering mechanism is the most suitable. It enables the application progress producing further more data, while running and computing the distances in background fast on the GPU. Empirical results to that are shown in chapter 7.
As stated previously, collecting the matchings can be done on either GPU or CPU. The process is depicted in figure 26. Collecting the matching on the GPU with a kernel program, safes time by not transferring back the resulting scores and configuration data from the back buffers to a host RAM. Advantageously this can make use of the GPU parallelism for each configuration with the same grid configuration used for the distance computation. However, this method needs to prepare the matching buffer, transfer them to the device, and afterwards back to host, and adding them serially in the buffer of matchings. Nevertheless, the process can operate asynchronous while copying meanwhile the front buffer data into the transfer buffers.

With the CPU collecting method, the configuration data and resulting scores must be transferred back to the host RAM to determine the matchings. The matchings can be assessed, collected, and directly added to the matching buffer, but this serially executed only.

During the kernel execution latency and amount of data transfers, it can be a better choice to collect the matchings by the CPU rather than GPU. The determination steps to add a matching need to check if a certain distance score falls below the specified maximum allowed k error. This requires conditional if-statement. In general, this causes raising the branch granularity that decreases the execution time on GPU.

The number of encountered matchings can decide the tradeoff between both variants. If not more than $\frac{1}{4}$ of all configurations has matchings, or the total number of occurred matchings is below than $\frac{1}{4}$ of the number of configurations, the CPU method has more efficiency.

But this decision may not be optimal, as it is not known in previous if only within a few configurations a large number of matchings occurs, or having many configurations each with only just zero, one or two residing matchings. If likely having every second configurations one matching, the benefit of parallelized collecting on the GPU device can be faster. This depends also on the GPU device capabilities. The user can disable the GPGPU method by not specifying the `Myers_Compute_Matchings_on_GPU` flag for the scheduler instance. The user is advised to examine which method results in a better performance at his use case.
6.5 The grid execution

The conceptional workflow above drafts the grid computing process. Once Myers scheduler is signalized to start the distance computing, it prepares and fills the device buffers, and a dedicated grid launching function is called. This launching function determines the grid parameters, the kernel parameters, and invokes the main entry kernel function with the grid configuration to run on the GPU device. This process is different for each GPU programming API and cannot be completely unified with common interface methods. The difficulties are described in chapter 6.6.

To obtain the grid dimensions the kernel parameters for shared, local, constant memory and the register usage are determined by provided API functions for the kernel entry function. This includes to detect the maximum number of allowed assignable threads for one grid block. This maximal number is then used to calculate an optimal block size, rounded over to the next lower number to a multiple of the warp size. This achieves a better thread scheduling on the GPU. The number of blocks needed to fully work off all the distance computations in the grid is calculated based on this block size. The ranges for the grid and blocks are declared one dimensional with the identified block and grid size values.

\[
\text{threadsPerBlock} = \left\lfloor \frac{\text{maxThreadsPerBlock}}{\text{warpSize}} \right\rfloor \times \text{warpSize} \\
\text{numberOfBlocks} = \left\lfloor \frac{\text{threadsPerBlock}}{\text{numberOfAlignments}} \right\rfloor \\
\text{blockDimension} = (\text{threadsPerBlock}, 1, 1) \\
\text{gridDimension} = (\text{numberOfBlocks}, 1, 1) \\
\text{totalGridSize} = (\text{numberOfBlocks} \times \text{threadsPerBlock}, 1, 1)
\]

This achieves a distribution that occupies the available streaming processors for a block with the current kernel resource usage optimally with a good scheduling behavior.

As the grid is launched on the GPU, the entire kernel function is executed for every thread in the grid, with all buffer data as input parameters needed for the distance scoring. The kernel function or current thread determines first it current unique location in the grid, exemplarily by \( \text{threadId} = \text{threadIndex}.x + \text{blockIndex}.x \times \text{blockSize}.x \).
This thread identifier in the grid is used to locate to current data for the distance scoring in the buffers. This data is liberated from the buffers.

After extracting the nested single computation data, the data is delegated to one of the different kernel memory variant function. These methods prepare the data for one of the memory handling scenarios described in section 6.1. After that, the specific scoring algorithm implementation is called to measure the distances. These functions evaluate and compute the scores finally as described in chapter 2.2 and 2.3.

The complete process on the device is constraint early with a suitable bit-vector type for the computations by a template parameter, which has been specified in chapter 5.3. The type is determined already before launching the grid for diverse reasons, instead of instantiate a concrete type in the kernel for one computation lately. The selection process is done in a best-effort way to appoint the needed number of bits. Generally decided with a step size in the power of 2 and some interstates to enable special effective bit-vector classes, trying to avoid wasting capacities too much. The following table shows the decisions scheme:

| \( m / \epsilon \) | \( |B| \) | \( m / \epsilon \) | \( |B| \) | \( m / \epsilon \) | \( |B| \) |
|-----------------|--------|-----------------|--------|-----------------|--------|
| \( \leq 8 \)    | 8      | \( \leq 96 \)   | 96     | \( \leq 384 \)  | 384    |
| \( \leq 16 \)   | 16     | \( \leq 128 \)  | 128    | \( \leq 512 \)  | 512    |
| \( \leq 32 \)   | 32     | \( \leq 192 \)  | 192    | \( \leq 768 \)  | 768    |
| \( \leq 64 \)   | 64     | \( \leq 256 \)  | 256    | \( \leq 1024 \) | 1024   |

Table 18 – Automatic bit-vector size selection scheme

To accomplish that choice, the scheduler must internally keep track to the maximal used pattern length or \( k \) error factor while adding measurements or pattern data into the front buffers to determine an appropriate bit-vector size at launch time.

This needed fixed selection scheme is one reason why a maximal pattern length and error factor restriction is defined. A developer can easily extend this with more interstates and even to larger bit-vector sizes if desired. However, larger pattern lengths above 1024 are even not widely in praxis with Myers algorithm, and may exceed hardware capabilities.

One disadvantage of this strategy is that at launch time only one certain dedicated kernel function is started depending on the currently decided bit-vector type, by either the detected maximal used pattern length or the maximal band width \( \epsilon \). For example having many small patterns but beside only one large pattern will select the kernel instance for this largest pattern as base for computing. This will generous waste capacities of the GPU for computing the small patterns. To deal with that, the user can group the configurations previously into two different scheduler classes, one for larger and one for smaller patterns for an efficient usage, and additionally hint a bit-vector size in advance.
A bit-vector class cannot be instantiated by a variable for a given number of bits. Remind that GPGPU does not support dynamically sized arrays. They are created with a template argument for their desired size. This requires a fixed selective choice in runtime. Either the scheduler has been given a hint of the bit-vector size or it must be automatically chosen on demand. For the unbanded algorithms, the length of the pattern is crucial for the size of the bit-vectors. For the banded scoring schemes, the width of the band $\epsilon$ determines the number of needed bits.

The second reason why the selection process is not done lately in the kernel runtime has compiling and execution reasons for GPGPU. Primary, this saves conditional if statements and branching paths inside the kernel program, which would affect the GPU runtime for the selection choice always negatively. Secondary, selecting the bit-vector type just in a GPU kernel will result in one large kernel entry function where the kernel resource parameters are determined by the heaviest operational path. The kernel resource usage directly affects the maximal number of assignable threads for one block by the hardware limitations. This concerns the numbers of parallel-executed threads inside a grid block. Using more resources, a lower number of threads can be executed concurrently in a streaming multiprocessor. However, to achieve a most effective grid setup with a maximum number of concurrent threads, depending on the use case, the computing methods for GPU are bounded to the specific bit-vector type by a template parameter early.

The main parameters that depend the kernel resources are the alphabet, scoring scheme, kernel variant, the bit-vector size, the used device architecture, and the kernel input parameters. If a kernel is compiled, the usage parameters of local, shared, and constant memory and used registers are assessed. The memory usage mainly depends on the array of bit-vectors used for the pattern representations in size of the alphabet. The number of operations and the length of the bit-vectors affect the register usage of a kernel program. An exemplary data table of assignable threads for one grid block based on the amount of used memory can be taken from appendix point I. The NVidia CUDA compiler creates for each different template parameter a different entry kernel function with different resource usages. The size of the bit-vectors has been restricted to 1024 because using more will exceed the maximum number of rationed registers of 124 in architecture version 1.x, and is vulnerable to fail the kernel in the runtime. With architecture 2.0 this does not happened, even with only maximal 63 assignable registers.

6.6 The kernel invocation and code binding

A difficult part is to invoke the main entry kernel function and start the grid computation on the GPU, as for each platform API a different conception must be realized. The invocation schemes for starting computing the distances and start collecting the matching are nearly in the same way.

For the CPU emulation, a loop iterates each desired thread in the grid, consecutively calling the entire CPU kernel function, to simulate a serial grid execution.
For CUDA two alternatives for the runtime exist. The first alternative is to use the native in-project grid and kernel function call construct "kernel<<<gridConfig>>>(parameters)". The NVidia compiler generates for every different template variation used, an own kernel entry function with an internal reference name. This name is required to obtain the kernel resource parameters. The naming scheme is officially not documented in the CUDA API so far. An example is noticed in the appendix point K. The second alternative for CUDA is to use a device compatible ptx-file ¹. Using this method, the code file is once loaded at the construction phase of the Myers scheduler. For the kernel invocations, all input parameters to the function must be set up accordingly correct by the developer manually. This has tendencies to be error-prone as changing the declarations, types or ordering must be harmonized in order to be executed and the ptx-file must be recompiled. At least for referencing the kernel entry function in the ptx-code, the internal dedicated kernel function name must be used in cases the file has been precompiled. ²

OpenCL uses a separated source file for the GPU program. This code file is loaded and compiled in the construction phase of the Myers scheduler with the provided OpenCL API functions. ³ Invoking the kernel function equals the CUDA methodic used for the ptx-files, including the tendency to be error-prone. To deal with the OpenCL language limits, as templating is not supported and moreover only simple static constructions are achievable, an implementativ trick is used to enhance the flexibility of OpenCL code. The language supports the #define and #if-else constructs. These are used to define and distinguish for example between the concrete scoring scheme function to call, the kernel memory variant , enabling the cut-off mechanism, specify the used bit-vector type and other. This allows providing a template-like behavior with #defines, additionally avoiding if-conditions at runtime in the OpenCL code. To channel in these constants with the actual used template parameters of the scheduler, while loading the OpenCL source code, a block of special code of #defines is generated and added in the beginning. The used constants are listed in appendix point L.

¹ The PTX-ISA language is closed hardware like assembler for parallel computing. The official documentation is located at the reference [PTX10]. A ptx-file for example can be a precompiled using the compiler to enable a faster project-compiling time, as compiling the different variations take some amount of time. Nevertheless, this offers an OpenCL like behavior of application and kernel code separation, as this ptx-file could theoretically be written by hand.

² Both CUDA runtime variants can be toggle by the MYERS_CUDA_USE_PRECOMPILED_SOURCE constant. The location to load the source file is specified in MYERS_CUDA_COMPILER_SOURCE constant. The default is disabled, to use native compiling.

³ The location of the OpenCL code file to use is defined in the MYERS_OPENCL_COMPILER_SOURCE constant. In this diploma thesis, an implementation with simple bit-vector types is provided in the outlying code file, which supports value type dependent up to 64 bits.
7 Empirical Results

One primary part of this work was to research what performance and speedup if any can be gained by using the GPU to compute a large number of distances with Myers method. Several kinds of performance measures were made with different settings to get comparable statements. The test sets are based on and varies by these parameters:

- “hard” parameters: alphabet, scoring scheme, kernel variant, GPU architecture
- “soft” parameters: number of distance measurements, hit rate/number of matchings, sequences length, patterns length, k-value / accepted differences
- other parameters: buffering method, cutoff mechanism, loop unrolling, forced inlining

The test setups were usually compared between the different APIs for CUDA, OpenCL, the CPU emulation, and the SeqAn methodic. This work will not present all possible settings and limits the presentation to several expressive setups only. The empirical results can only present an estimation of what GPU can offer compared to a specific hardware configuration.

The tests were executed on the following hardware configuration:

Host: Intel i7 930 with 2,8 GHz (4/8 core), 4 GB RAM, 64 Bit
Device: NVidia GeForce 460 GTX with 768 MB, Core 727 MHz, Memory 1.8 GHz, 7 SM a 8 SP = 56 cores, CUDA 2.1 support, OpenCL 1.0 support
API: CUDA SDK Version 3.2 with OpenCL 1.0 support

The default test settings are using the quad buffering mechanism, without the cut-off mechanism, using the local kernel memory variant, computing the local Levenshtein distance, with enabled loop unrolling and forced inlining, using the BitVectorArray class for large bit-vectors, a hit rate of 25% in the number of total distance measurements, DNA as the default alphabet, the CUDA architecture 1.0, and an accepted distance of \( k=0 \). This work will present the influencing factors of the different settings in section 7.4. The test setups are always randomly generated. Performance and time are measured in seconds for adding and computing the setups, collecting the matchings and processing the whole tests, but without the data generation time for assessable results.

Two issues needs to be considered before interpreting the results. First, the CPU emulation will simulate the same behavior as the GPU version, except for computing the grid that is done straight forward in a loop and the memory issues of asynchronous transferring to a device and dealing with page lockable memory. The emulation gives an expression of how the same implementation will behave in a serial execution. Secondly, the SeqAn tests are based on Myers using the Finder/Pattern model, but being only straight iterated over all setups, without storing any of the data or matchings. This gives a valuation of the needed raw computing time with the usual methods in SeqAn. Therefore, a 1:1 comparison between the methods may not reflect the real applicative conditions.
7.1 Linear scaling performance

The analysis will start with the performance behavior of the parameters that are expected to show a linear increase of the runtime, based on the default test settings. Primary this is dependent by the number of setups. Doubling the amount should need twice of the runtime even with parallelization. The quantity is stepwise doubled from 16.384 up to about \( \frac{1}{4} \) million, in precise 262.144. Secondary the sequence length \( n \) has properly a linear influence as the main computing loop iterates over the sequence.

7.1.1 Performance for small patterns (\( m<=w \))

Beginning the tests with pattern lengths, resp. bit-vector sizes, that fit into one hardware register of size \( w \). At least for CUDA this assumes \( m=32 \). The sequence lengths are characteristic sized from \( n=64,...,8192 \) by doubling each time. The tests measures the complete processing time in seconds.

![Graphs showing linear scaling performance for different pattern lengths.](image-url)
Empirical Results

Some conclusions can already be made here. First notice, the time measures have a limited precision in about a $\frac{1}{100}$ of a second. Secondly, the computations underlay natural fluctuations in its speed. Both effects are noticeable especially for smaller sequence lengths with runtimes of below one second. However, the expected linear scaling behavior can be observed for each method for the growing number of setups and a growing sequence length. It is reasonable that doubling the number of setups doubles the amount of time, even with GPU parallelization, as only a limited number of computing GPU cores exists that compensates against the large number.

It can be seen that the SeqAn method is the slowest even with no data management. The results show a speedup of two times for CUDA, and a speedup of four times in OpenCL compared to the SeqAn methodic. The CPU emulation is in the beginning nearly as fast as the CUDA implementation, but shows an adjustment to the SeqAn runtime by larger sequence length. The speedup statement holds true, reminding the unprecise measurement conditions for smaller pattern lengths.

It cannot be clearly resolved what causes the effect that the OpenCL implementation to run twice as fast as CUDA. By experiences, and even by using simple types only—not the BitVector classes—, the execution time on GPU itself takes the double amount of time in CUDA compared to OpenCL. This may be a reason of the compilation abilities of the NVidia compiler, translating the code into a large hardware abstracted PTX device code file, dealing with templates and the complex used structures rather unoptimized. The OpenCL code applies a rather simple code structure and complexity. Further, declarations of read and write-only memory and the ability of prefetching data that is used and only supported in the OpenCL API and may result in this better performance behavior. The driver management of asynchronous stream operations can be a reason also. For CUDA this behavior must be explicitly enabled, whereas in OpenCL it must be explicitly disabled. In addition, the internal kernel call and grid instantiation latency by the driver may be a reason here too.
7.1.2 Performance for large patterns ($w > w$)

Next, the same settings are used to test with a pattern length and resp. a bit-vector size of $m=256$ that do not fit into one machine register anymore. Here the arbitrary bit-vector classes are used. Only a comparison between the CUDA, CPU, and SeqAn runtime is performed, since the OpenCL implementation does not support more than $m=64$ bits.

The received values confirm the previously observed results. The tests with a higher pattern length scales also linear with the number of setups and the sequence length as expected. The performance gain of using GPU is obviously revealed with larger patterns length. It can be noted that the CUDA runtime is about twice times faster than the CPU emulation, and about three to four times faster than SeqAn. One could anticipate that OpenCL runs also twice as fast as CUDA, but this is an outstanding proof.

Implementational reasons causes that the CPU emulation runs two times faster than the SeqAn method, even with data and scheduling overhead. The implementation uses highly optimized bit-vectors and improvements for consecutive computations in its performance in comparison to the default SeqAn implementation with the Pattern/Finder Model with break-and-continue semantic for iterating through the setups.

In the appendix point J a notice to the expected theoretical runtime and the observed runtime is further commented.
7.1.3 CPU discharge

Besides the speedup gain by using the GPU, as the computations and data flow is designed to run asynchronous in background not blocking, the relief time for CPU can be measured. This is an important aspect in the application to operate, collect and insert data further on, not occupying the CPU power with the computation. The sequential parts of the GPU runtime are needed to give a reliable statement. This time can be quantified by the time needed only by computations, to be precise only the grid launch execution times as computation runs asynchronous, and by subtracting them from the total processing time. As reference times, the raw computing time on the CPU and with SeqAn are used. To ascertain the saved CPU processing time, the reference times used for computing on CPU or with SeqAn are than subtracted with the time needed for the sequential parts for the GPU methods.

For the pattern length of \( m = 32 \) the following discharging times can be reckon:

![Diagram 3 – CPU discharge times for a scaling sequence length with pattern length of \( m = 32 \)](image)

The sequential part is in average a value of 33% for CUDA and 43% of the OpenCL runtime. Considering the natural fluctuations and timing precisions for small length, they are nearly equal. Because of that, the discharging times of CUDA and OpenCL overlap in these diagrams in compare to CPU and SeqAn. As the runtimes already suggest, the times scales linear with the number of setups and between the CPU and SeqAn a factor of two times of CPU relief occurs.
For the pattern length of \( m=256 \) the following CPU discharge times can be observed:

Here, the percentage of the sequential portion for CUDA lays in average at only 10%.

The time of non-blocking CPU reflects the expected linear scaling behavior by the length of the sequence and the number of setups as a natural result. The more and heavier operations will be, the more time is saved in opposite while running the computations on the GPU in background. The saved time ranges from some milliseconds for a small amount of setups and small data lengths up to several minutes with large number of setups and greater data lengths. This follows the runtime characteristics of the methods on these test settings.

Against all expectations, the discharging time is not directly indicated within the measured processing runtimes. It is not true that the runtime is desired to be many times lower, in precise exactly the time saved for CPU, as effect of the asynchronous operations. The application workflow needs certain synchronization points to wait on the GPU device to finish the computation. Either during the scheduling processing, at a certain point if all buffers in the chain are fully occupied, it must be waited for results to proceed further and to gather the matching results. Moreover, at the last final launch call, that needs to compute and wait until all left distance measurements in the buffer chain were proceeded, is a synchronization point. Therefore, the discharging time is not reflected in the processing runtime, but it can be quantified as time that could be used for data collecting and filtering, otherwise instead it will be wasted during waiting at the synchronization points.
7.2 General performance behavior

For a general statement of performance, a wide range test for all applicative pattern lengths, resp. bit-vector size, from m=8 to m=1024 with 16.384 to 262.144 setups was measured. The sequence length was chosen to be twice the length of the pattern, which will satisfy common read mapper applications where verification data are usually filtered in a range of maximal twice the pattern length, to give a good valuation.

The diagrams with m<=32 are not representative, as their processing times are below 1 second and underlay natural fluctuations and the timing precision. Above m>32 interpretable results are revealed. The runtime for SeqAn is always the slowest method compared to the GPU implementations. With an increasing pattern and sequence length, the CPU emulation runtime claims up to the time of SeqAn, and grows beyond. This is a result of the overhead needed for data scheduling and management in the sequential processing part that is not done in the SeqAn test. With increasing data lengths and amount, the distance computing time of SeqAn and CPU will likely be the same, as the sequential scheduling part decreases in its ratio to the complete processing time.
The attached graphic in diagram 6 shows the average values of the sequential portions for scheduling in the complete processing runtime. It is noticeable that the severity of the scheduling and data management falls down to only some percent, if the computational part increases in its hardness.

Based on the gathered results, the factor of speedup between the GPU version, the CPU emulation, and SeqAn methodic in their runtime can be calculated:

The results with smaller pattern m<32 length are processed rather fast that no realistic interpretation can be given for this lengths. However, a general speedup of up to 1-2 times for the smaller pattern length could be observed in further experiences. The SeqAn implementation shows a big leap in the implementation for larger patterns above m>32 that jumps up from 6 to 10 times faster than the CUDA runtime. The CPU emulation with an optimized bit-vector implementation does show the expected behavior, whereas for each higher pattern length a linearize gain up to 8 times is achieved.

This point clearly shows how the GPU affects the performance advantages. The full exhaustion of the graphical device power is most effective with larger pattern lengths, above the machine word size, and data sizes. For lower bit-vector sizes, which fit into one register,
and lower data amount, the runtime is mostly consumed by the sequential parts for controlling and data buffering, by transferring the memory data to the device, and by the grid launching latency.

For completeness, the CPU discharge time of the measurements reveals this behavior also:

Two facts are clearly observable by the diagrams for pattern length of $m > w$. Using more setups to compute on GPU will save more time on CPU. Using heavier operational bit-vectors will discharge the CPU even more. Both effects scale linearly as expected.

### 7.3 High throughput performance

For the applicative usage, the behavior with a high data throughput was tested to give an impression of what can be reached, even for smaller pattern lengths that are not reliably measureable with only a relatively few number of setups.

To express the idea of what amount of data is thinkable as high data throughput is described in the following table, pointing out the amount of memory that is processed for different sequence lengths and amount of distance measurements.
### 7 Empirical Results

#### Table 19 – Data throughput amount for different number of measurements and sequence length

<table>
<thead>
<tr>
<th>N</th>
<th>65.536</th>
<th>131.072</th>
<th>262.144</th>
<th>524.288</th>
<th>1.048.576</th>
<th>2.097.152</th>
<th>4.194.304</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.5 MB</td>
<td>1 MB</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
</tr>
<tr>
<td>16</td>
<td>1 MB</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
</tr>
<tr>
<td>32</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
<td>128 MB</td>
</tr>
<tr>
<td>64</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
</tr>
<tr>
<td>128</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
<td>512 MB</td>
</tr>
<tr>
<td>256</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
<td>512 MB</td>
<td>1 GB</td>
</tr>
<tr>
<td>512</td>
<td>32 MB</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
<td>512 MB</td>
<td>1 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>1024</td>
<td>64 MB</td>
<td>128 MB</td>
<td>256 MB</td>
<td>512 MB</td>
<td>1 GB</td>
<td>2 GB</td>
<td>4 GB</td>
</tr>
</tbody>
</table>

The following graphical evaluations show the runtime and speedup factor for about one hundred thousand up to 4 million setups.

**Diagram 9 – Runtime and speedup for scaling pattern length for large number of setups**

![Graphs showing runtime and speedup for different setups and sequence lengths.](image-url)

- **CUDA**
- **OpenCL**
- **CPU**
- **SeqAn**
Empirical Results

524,288 setups

1,048,576 setups

2,097,152 setups

4,194,304 setups

speedup factor

m=8  m=16  m=32  m=64  m=128  m=256  m=512  m=1024

0,000  2,000  4,000  6,000  8,000  10,000

0,125  0,250  0,500  1,000  2,000  4,000

0,125  0,250  0,500  1,000  2,000  4,000
The characteristics were already recognized in previous testing results and could be observed here too. In general a speedup for pattern length of m<=32 up to two times is observable. Compared to the SeqAn method a factor of eight times is achieved until m<=512, and above this decreases to six times on the GPU. The speedup to the CPU method increases near linear by the pattern length up to a factor of seven.

A good reliable statement of the average speedup factor can be exposed, based on the measured results:

<table>
<thead>
<tr>
<th>m</th>
<th>GPU vs. CPU</th>
<th>GPU vs. SeqAn</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1,276x</td>
<td>1,192x</td>
</tr>
<tr>
<td>16</td>
<td>1,479x</td>
<td>1,611x</td>
</tr>
<tr>
<td>32</td>
<td>1,592x</td>
<td>1,946x</td>
</tr>
<tr>
<td>64</td>
<td>1,593x</td>
<td>6,825x</td>
</tr>
<tr>
<td>128</td>
<td>2,545x</td>
<td>8,344x</td>
</tr>
<tr>
<td>256</td>
<td>4,376x</td>
<td>8,553x</td>
</tr>
<tr>
<td>512</td>
<td>6,904x</td>
<td>8,592x</td>
</tr>
<tr>
<td>1024</td>
<td>7,334x</td>
<td>5,773x</td>
</tr>
</tbody>
</table>

Table 20 – Determined speedup factors using GPU for Myers algorithm

It must be stated that these values cannot be taken as a general outcome. They are applied to a certain underlying hardware and will only hold true for that configuration with the used test settings.

It is reasonable that the speedup factor up to m=32, or especially m=64 for CPU, only shows a small increase of performance. These cases uses bit-vectors that always fits into on machine register, resulting in the same runtime and computing characteristics of them. Additionally, it must be considers that the sequential part of the runtime is about 40%-50%, and the runtime includes the waiting time at the synchronization points of as described in 7.1. Moreover, memory transferrings and execution latency has a relative great amount in the total runtime. In total at these cases results in a rather small gain between 1.5 and 2 times using GPU.

Above m>w the sequential parts falls stepwise down to 10% with m=1024 and the memory transfers and execution latency time became a small part in the total runtime. Instead of obtaining an expected speedup of near twice the time when doubling the pattern length, a far lower factor is reached. This is reasonable as the level of parallelization vastly decreases with a growing pattern length caused by needing more resources for a kernel. Exemplary listed in the appendix point I, showing the assignable threads per grid block for different pattern length and alphabets. This is also the reason why for m=1024 a breakdown of speedup is noticeable as the achievable concurrent executed threads falls down to a minimum.
7.4 Performance influencing parameters

The settings in the test before were fixed to a specific parameter space to get comparable results. Therefore, the influences of the different parameters mentioned in the beginning of the empirical analyses are now being examined.

7.4.1 Double vs. Triple vs. Quad buffering

To research the different buffering variants only the sequential runtime of the CUDA implementation for 261.144 setups is tested. This is accounted because in general at 65,536 setups a lunch is executed, which is configuration hardware dependent. Therefore, this number of setups has been chosen to simulate a full buffering pipeline for all variants.

Diagram 10 – Sequential runtime of CUDA for different buffering methods with scaling pattern length

The double buffering variation is blocking the CPU at most by design. Copying the host front buffers into the device back buffers is not asynchronous, because this process must be finished before resuming to the application, to fill the front buffer further on. The triple and quad buffering fully use the possibilities of asynchronous transfers, not blocking the CPU. This results in much better, lower sequential behavior. The quad buffering method delays the grid launch while computations are not finished. This achieves a slightly better performance for computational harder operations where more processing time is needed. Notice that the measured times includes only adding and handling of the data, not the computations.
7.4.2 Influence of hit rate

The next tests were executed to get to know how the hit rate and number of matchings affect the runtime. A rate of matchings from 0%, 25%, 50%, 75%, and 100% was tested. The tests measured only the computation time needed to process the setups and to collect the matchings. The diagram displays the increasing factor of runtime compared to the time when no hits occur.

![Diagram 11 – Increasing factor using different rates of hits with scaling pattern length](image)

The matchings are collected by iterating over all distance scores with the CPU method. Gathering on GPU is processed parallelized with a dedicated kernel function. At the hit rate of 50% the GPU method is used. The factor of runtime increase is not more than twice, and is realistically below a factor of 1.5x. Measuring with small patterns length of m<=32 needs only some milliseconds and are subject of imprecisions. However, collecting more matchings needs more time naturally, but this will not affect the runtime dramatically at all.

7.4.3 Influence of GPU architecture

For CUDA devices, the NVidia compiler can be introduced to compile the code for different architectures. The influences are proofed against the different architecture code compilations, and measured by their computational time for executing the kernel launches.

![Diagram 12– Runtime for different CUDA code architecture with scaling pattern length](image)
Beside natural fluctuations, not influences on the runtime by the different architecture versions can be observed. This may be a result of testing on the same device with support of 2.1 capabilities, where always the same runtime characteristic is achieved, even if the code is differently compiled especially between version 1.x and 2.x. This might change if the tests are made on a system with some comparable devices of different architectures, as these cards will have different hardware capacities of memory and cores.

### 7.4.4 Local vs. shared vs. global memory variant

The different implemented memory variants used by a kernel shows the following characteristic:

![Diagram 13– Runtime for diff. kernel memory usage with scaling pattern length](image)

No direct performance differences can be observed between the local and the shared variant. The shared variant is used to copy the alphabet into the shared memory and use it from that memory for computations. The desired faster access of this memory is obviously compensated by the copy process compared to the local variant where the alphabet remains in the global memory. In the global mode, the pattern bitmasks are placed into global memory, which has a clearly higher access time. Even if more threads can be assigned to each SM and increases the level of parallelism, as no shared memory is needed to store the bitmask, the longer access time degrades this effect. Furthermore, a large global memory part will be reserved for storage, lowering the achievable total number of setups processing at once.
7.4.5 **BitvectorArray vs. BitvectorList**

For dealing with large bit-vectors (m>128) two classes have be designed with different conceptions. Either with a fixed array of vector items to store the bits – the BitvectorArray class- or by using meta-programmed linked list – the BitvectorList class. The factor of speedup or runtime decrease of using the list implementation on CUDA or CPU has been observed and is displayed in the following diagram.

![Diagram 14– Diff. runtime factor using the array or list implementation for bit-vectors](image)

The effect is a slower executional runtime on GPU and faster runtime on CPU. This is expected. The concepts underlay different vector item accessing mechanisms. As for an array, an item is located by its offset from the starting array location using indirect addressing. The linked list variation accesses an item through recursive searches along the linked list with a direct addressing mechanism. The list mechanism increases the branch granularity on GPU as if conditions are needed to locate one certain item in the item chain, but on CPU, this mechanism results in a better performance.

**7.4.6 Iterative loops vs. unrolled loops**

When using the BitvectorArray class, the operations over the single vector items can be executed iterative over loops, or using meta-function for unrolling these loops as the array has fixed lengths. The following diagram shows the runtime factor when using normal iterative loops instead of unrolled loops:

![Diagram 14– Factor of runtime increase using unrolled loops with scaling pattern length](image)
A great performance improvement with over twice the time can be noticed when using unrolled loops, which therefore can be recommended to enable. An additionally bad effect when using iterative loops is that the number of used registers is exceeded for $m=1024$ at CUDA runtime. This results to fail and crash the kernel execution. The denoted value is only estimated, but revealed that not unrolling loops have greater performance loss on GPU also.

### 7.4.7 Cut-Off vs. no Cut-Off mechanism

Using a cut-off mechanism will break the computation of one distance measurement early, if the score cannot fall below the $k$-value anymore. It is expected that the GPU runtime will increase due to the branch granularity, and will slightly be faster for CPU. The diagram below shows the runtime factor for CUDA and CPU when enabling the cut-off mechanism.

![Diagram 15– Factor of runtime using cut-off mechanisms with scaling pattern length](image)

The observed results show with an increasing pattern length a runtime enhancement on CPU of up to about 10% and for GPU runtime of about 5%. This is reasonable because for larger data lengths the distance computation may stops earlier. This effect varies depending on the number of expected hits in the setup - here 25%. If more hits are expected, the cut-off beneficitation will decrease as the cut-off point will be reached less frequently.

### 7.4.8 Alphabet influence

The length of the alphabet influences the array used to store the pattern bitmasks in the algorithm. A bit-vector for each alphabet symbol exists that represents the pattern states in the length of $m$ in this array. Therefore this depend the used memory for one kernel, resulting in a larger consumption of memory using larger alphabet. The memory usage always directly affects the number of assignable threads per SM as only limited memory resource are available, decreasing the level of parallelism, depicted in the appendix point I.
It is expected that using a greater alphabet will lower the runtime. The diagram 16 shows the gathered results. This influence on GPU is directly observable in the results as performance loss. Depending on the used alphabet a less parallelism and runtime is achieved. This is not the case on CPU since it does not need to deal with that limitation factor of hardware.

Furthermore, the kernel failed to execute with $m=1024$ for compiling with CUDA 1.0. By compiling and testing for architecture 2.0 this does not occurs, deductive that 1.0 has some problems dealing with overflowing register usages, like this happened already before in 7.4.6 comparing the used bit-vector classes.

![Diagram 16– Runtime and factor of speedup alphabet depended with scaling pattern length](image)
7.5 Performance of banded algorithm

The banded algorithms are independent to the used pattern length. They are dependent on the ε environment, the range of the band. The ε range appoints the number of bits used for the bit-vectors in the computations. The k-value for the error factor is used to determine this ε-band range by the next logarithmic size of k+1 as listed in the table aside.

| k+1 | ε/|B| |
|-----|---|--|
| <8  | 8  |
| <16 | 16 |
| <32 | 32 |
| <64 | 64 |

The first tests are executed to proof the independence of the pattern length in the runtime. They are applied with the default settings, a fixed number of setups of about \(\frac{1}{4}\) million, a fixed sequence length of 1024 and an accepted rate differences of k=6 that ε=8. The matchings are always only counted and not collected.

![Diagram 17– Runtime of unbanded and banded algorithm with scaling pattern length](image)

Obviously, the pattern length has no direct effect of the runtime any more for the banded method, especially observable for the CPU. Mainly the runtime stays constant for different pattern lengths compared to the linear increase with unbanded algorithm as stated in section 7.2 and 7.3. The slightly increases of the runtimes with larger pattern lengths are caused by the need to handling and transferring more data for the tests. Additionally remarkable, the current OpenCL implementation can be used for all use cases of the banded method, even with a pattern length m>64, because the banded algorithms depends the needed bit-vector size only on ε. In assumption that ε <= 32 or ε < = 64, only the simple typed bit-vectors are needed and used, which can therefore be applied to the current OpenCL method.

The next tests were executed with an accepted rate of k=5, 10 and 20 differences, so that ε=8, 16 and 32 to measure the influence in to runtime depending on k. They are applied with a fixed sequence length of n=256, a pattern length of m=128 a varying number of setups. It is expected that runtime of the banded method should linear increase by the number of setups, that has been observed in section 7.3, and increase for each higher ε range.
The diagrams show that the banded algorithm scales linear by the amount of setups. Since this method also iterates over the sequence length, it will follow the previously observed results that the runtime will scale linear by the sequence length too. It is recognizable that only a minimal increase of runtime for the different k reps. ε occurs. This is reasonable since ε <= 32, and therefore the bit-vectors sizes are below 32, are small enough to be computed always within one register, that not remarkable performance loss or gain can be noticed.

The direct comparison between the unbanded and banded method in diagram 18 does not reveal differences in the runtime on the GPU, but for the CPU emulation that is about 4 times faster. To confirm that behavior the same tests setup used in section 7.3 was measured with k=0 for the banded algorithm and compared to each other in the runtime. The following diagrams shows the speedup factor of using the banded method compared the unbanded for a scaling pattern length and number of setups differed for CUDA and the CPU emulation.
The characteristics show a different behavior for GPU and CPU using the banded method. Notice that up to 131.072 and \( m \leq 32 \) the measuring times are below one second and underlays the timing imprecision and natural fluctuations to interpret the results.

For the CPU, the advancement scales linear with the pattern length. This is an expected behavior because the size of the bit-vectors for the unbanded algorithm is directly bounded to the pattern length. The banded method depends the size only on \( \epsilon \) that is below 32. For \( m = 128 \), on a CPU with 64-bit support, needs two vector items to represent a bit-vector with the unbanded method. Therefore, twice the time is needed to compute them compared to the banded method where only one vector item is needed since \( \epsilon \leq 32 \). Further, a factor of eight times and more on a 64-bit CPU for \( m = 512 \) is expected using banded algorithm, but it is even lower. The banded method uses no pattern preprocessing, but this operational lightweight step does not decrease the runtime in total, as this part is indirect executed too. In general, the operational hardness of both methods is near equal. Additionally hyper-threading abilities on the CPU can advance here the operational processing with the larger bit-vectors. In total this result in a lower expected performance gain, but nevertheless instead of having a runtime of \( O\left(n \left\lfloor \frac{m}{w} \right\rfloor \right) \) the banded algorithm shows to have only \( O(n) \).

For CUDA a speedup of up to 1.5 times is only noticeable. This is a direct effect of the used asynchronicity, since the complete processing time is measured instead of the raw computation time. Measuring the computations time only without asynchrony and blocking the CPU will reveal the same behavior observable for CPU. In fact, the algorithm is proceeded faster in a direct comparison of both methods for different pattern length with equal \( k \). However, the effect in runtime is hidden in the background because the sequential processes of the application cover them. This is not the case on CPU with no asynchronous
execution is done. Further, no advantage of CPU discharging is offered. The overall total time of the banded method achieve an improvement, but not even twice the time.

7.6 Performance of Hamming distance computing

The performance behaviour for computing the Hamming distance on GPU has been measured briefly at last. The test was designed to compute 1 million setups with an increasing length of the texts, both the same lengths, to compare with each other.

The diagram 20 of the runtimes indicates that using GPU does not always result in a good performance gain. The complexity and computational time for the Hamming distance itself is rather low that no remarkable better speedup can be achieved. Here, the runtime of the used SeqAn method is even faster compared to CUDA and the CPU emulation. It must be noticed that these is mainly caused by the overhead of data scheduling and management what is not done in SeqAn.

The CUDA implementaion is observable the slowest of all methods. Using the OpenCL implementation is twice and more times faster as CUDA. Only with OpenCL the time used to compute the hamming distances compared to SeqAn is slower with an increasing text length. Nevertheless, the sequential time for scheduling and data management is dominating the total processing time as this could be compensated by the OpenCL GPU implementation.

The profit of using GPU can only be classified by the CPU discharing time as the computation is executed in the background.
7 Empirical Results

The diagrams above indicates that the saved CPU time only takes effects on GPU if directly compared to the emulation on CPU. Compared to the SeqAn method even more CPU time is needed for processing, as this is a direct result of the overhead in the sequential part of scheduling and data management, and of course the memory transferring an kernel execution latency time. The CUDA runtime is even slower than the CPU emulation at about 10%-20%, compared to some performance gain between the OpenCL implementation and CPU emulation in a factor up to 1.5x. This confirms the results that OpenCL has a better GPU execution behavior then CUDA, which was already notices in the beginning of the empirical results.
8 Summary and Perspective

One major empirical finding is that Myers algorithm on GPU gets the most acceleration in cases when the bit-vectors are using a number of bits above the machine word size. That means the offered performance of the GPU shows best results if the bit-vectors cannot be computed within one machine register in $O(1)$ time anymore, and operations need $O\left(\frac{b}{w}\right)$ time, where $b$ is the number of used bits and $w$ the machine word size. The original Myers algorithm obtains here a high performance in cases with larger pattern lengths $m > w$. This includes also the number of distance measurements and the length of the sequence. Using more data will always gain more performance using GPGPU. In addition, the method enables a vast release of CPU power due to the computation in the background on the graphic card as shown in empirical result in chapter 7.

For the unbanded algorithms, a runtime improvement of six to eight times has been achieved in the tests with pattern lengths of $m > w$ compared to SeqAn. Even for smaller patterns with $m \leq w$ the tests shows an increase of runtime of up to two times for CUDA and four times using OpenCL. The banded algorithms perform good results also on CPU while not being dependent on the pattern length and fulfill the expectation of an $O(n)$ algorithm in general. Compared to the runtime on GPU, the performance effect is not remarkable increasing like the unbanded methods. Nevertheless, it runs faster up two times in CUDA and up to four times for OpenCL, addressing the runtimes of the unbanded algorithm with $m \leq w$. However, these results must certainly respect the underlying used GPU hardware. Further noticeable is that the times that relief the CPU while computing on GPU and the applied buffering mechanism can now be used to supply data continuously without great interruptions. The sequential time for scheduling need about 40-50% of the total processing time for the unbanded algorithm with $m \leq w$, meaning to safe more than 50% of time for computing with the CPU. For larger pattern length of $m > w$ only up 10%-20%, depending on $m$, of sequential time is needed, implying that 80% of the time is relieved for CPU usage.

This is a good result for the practical usage of the Myers adaption, as the pattern lengths will usually not have a small length of only 32 or 64. It is expected to be used within a length of several hundred or a length of thousands, driven by further technical achievements too. The implementation could be advanced easily to support even more than a length of 1024, which has mainly been specified by technical reasons of graphic device hardware limitations than by implementational reasons.

This diploma thesis was initiated to develop for the CUDA API. Later the request of an OpenCL implementation was demanded to support the major graphic card conductors with GPGPU abilities. The main programming part of about 75% was needed to build up the GPU infrastructure for SeqAn and to provide a unified way for both CUDA and OpenCL and a CPU emulation mode, and combine their different design philosophies. This includes the basic
common GPU functions and classes, bit-vectors, buffer and array classes, the scheduler interface class, the GPU execution controlling, and further specialized methods. About 15% of the implementation time was used for fine-tuning, improvements, and attentions to find the most performant way of usage. Only 5% of the time was needed to assemble the different variants of Myers algorithm and distance methods by Hyyrö. The rest of the time was used for documentations.

The architectural code was committed to give a maintainable and extensible state for other implementations with GPGPU in SeqAn. The left open OpenCL implementation for larger bit-vectors requires some further reflections about a good realization. In CUDA designated classes with templating mechanism was favorable to realize a general usage without specializing the Myers and Hyyrös algorithms for the different types. For OpenCL a different conception had to be considered, enabling a variation for arbitrary fixed bit-vectors sizes dealing with the programming functionality of OpenCL. This could be done by using the supported #define directives with constants, but would need to compile the code multiple times for different sizes on demand. This was not the scope in this diploma thesis, but the main part of the underlying architecture for that has already been built up for that.

Further, the functional power of the hardware and underlying API makes progress. Recently, in March 2011, NVidia released the CUDA Toolkit 4.0 for developers. Two new important features have been evolved besides others improvements. First, Multi-GPU support is added for a single host program that was already been available in OpenCL. Unfortunately, as this was not possible before, the implementation did not made use of it, as this would contradict the unified infrastructure built up for CUDA and OpenCL in SeqAn. Secondly, a concept for dynamic memory allocation in kernels with the default new() and delete() constructions is introduced. This could have bypassed the problem of selecting and instancing a kernel each for a bit-vector size in the best effort way. Now this could be done dynamically, not wasting memory, bounding each computation on their requirements, and not any more by the largest bit-vector size at the launch for a collected block of distance measurements. However, this functionality has not been included in OpenCL by now.

On top of this, the GPU hardware is expected to increase its capacities and power steadily. Newer generations have already up to 512 cores instead of only 56 used in the tests. The core frequency rises up to over 1.5 GHz each. The memory limitation for each kernel, beginning with 16 KB in CUDA 1.x and with now 48 KB of shared memory or 512 KB for local memory by the CUDA 2.x specification, will increase in the next generations, enabling even higher applicable bit-vector sizes. This will not only affect a speedup of runtime, also the achievable parallelism will increase. Therefore, the available hardware will raise the factor of speedup and performance gain multiple times of the measured results. In future, and already now the power of graphic devices are grown as expected by the Moore Law, even for the CPU, make it difficult to ascertain a general behavior except that using GPGPU for Myers algorithm achieves always a better performance with saving operational time on CPU.
9 Literature and references

9.1 Register of references

<table>
<thead>
<tr>
<th>Reference</th>
<th>Source</th>
</tr>
</thead>
</table>
9 Literature and references

<table>
<thead>
<tr>
<th>Reference</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kna08]</td>
<td>Course Script Computer graphics, C. Knauer(2008), FU-Berlin</td>
</tr>
<tr>
<td>[PTX10]</td>
<td>NVIDIA® PTX: Parallel Thread execution ISA Version 2.1, April 2010</td>
</tr>
</tbody>
</table>

9.2 Register of citations

<table>
<thead>
<tr>
<th>Reference</th>
<th>Citation source</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>[MYE]</td>
<td>From reference [Mye99], Chap. 1, pg.2</td>
<td>8</td>
</tr>
<tr>
<td>[SEQ]</td>
<td><a href="http://www.seqan.de">www.seqan.de</a> - Homepage, Paragraph “What is SeqAn?” access-date: 28-02-2011</td>
<td>41</td>
</tr>
</tbody>
</table>
### 9.3 Register of illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Source Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Needleman-Wunsch Alignment <a href="http://www.ibm.com/developerworks/java/library/j-seqalign/NWTab2.gif">http://www.ibm.com/developerworks/java/library/j-seqalign/NWTab2.gif</a> access-date: 05-02-2011</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Sellers Algorithm Scheme - from reference [GKR09], chapter 3.9</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Banded Alignment Scheme - modified Figure 1 by author</td>
<td>7</td>
</tr>
</tbody>
</table>
| Figure 4 | a Scoring Matrix - by author  
  b Vertical Δ-values - by author  
  c horizontal Δ-values - by author | 9    |
| Figure 5 | Cell Structure - from reference [Mye99], chapter 3 | 9    |
| Figure 6 | Myers Computing Matrix - by author | 9    |
| Figure 7 | Banded algorithm scheme - by author | 14   |
| Figure 8 | Wireframe Sphere [http://de.wikipedia.org/wiki/Datei:Sphere-wireframe.png](http://de.wikipedia.org/wiki/Datei:Sphere-wireframe.png) access-date: 01-23-2011 | 17   |
| Figure 9 | Rasterized fish graphic [http://commons.wikimedia.org/wiki/File:Raster_graphic_fish_40x46_20x23_overlay_hdtv-sdtv-example.png](http://commons.wikimedia.org/wiki/File:Raster_graphic_fish_40x46_20x23_overlay_hdtv-sdtv-example.png) access-date: 01-23-2011 | 18   |
| Figure 10 | Gouraud shaded sphere [http://de.wikipedia.org/w/index.php?title=Datei:Gouraud_high.gif](http://de.wikipedia.org/w/index.php?title=Datei:Gouraud_high.gif) access-date: 01-23-2011 | 18   |
| Figure 11 | Shaded Utah Teapot [http://commons.wikimedia.org/wiki/File:Utah_teapot_simple.png](http://commons.wikimedia.org/wiki/File:Utah_teapot_simple.png) access-date: 01-23-2011 | 18   |
| Figure 12 | Lone House from Blender 3D [http://en.wikipedia.org/wiki/File:Lone_House.jpg](http://en.wikipedia.org/wiki/File:Lone_House.jpg) access-date: 01-26-2011 | 19   |
| Figure 13 | Comparison of the GFlops of GPUs and CPUs - from reference [KHW10] ; Figure 1.1 | 20   |
| Figure 14 | Graphic Pipeline Scheme - from reference [BC09], Figure 2.3 | 21   |
| Figure 15 | Graphic Processors Scheme - from reference [BC09], Figure 4.1 | 22   |
| Figure 16 | Architecture Scheme of CPU vs. GPU - from reference [KHW10] ; Figure 1.2 | 23   |
| Figure 17 | Unified Device Architecture Scheme of a NVidia graphic card - from reference [KHW10] ; Figure 1.3 | 24   |
| Figure 18 | OpenCL Architecture Scheme - from reference [OCL10] | 26   |
| Figure 19 | ATI Stream Device Architecture - from reference [OCL10] | 26   |
| Figure 20 | The "peach" of a parallel Program - by author | 27   |
| Figure 21 | Computing Grid scheme in 2D - by author | 28   |
| Figure 22 | Thread assignment scheme in CUDA - from reference [KHW10]; Figure 3.7 | 30   |
| Figure 23 | GPU Buffer Structure Scheme - by author | 47   |
| Figure 24 | Scheduler processing scheme - by author | 63   |
| Figure 25 | Scheduler Launch Process Scheme - by author | 64   |
| Figure 26 | Matching Detection Scheme - by author | 65   |
| Figure 27 | Grid execution flow - by author | 66   |
Appendix

A. Different technological terms used by CUDA and OpenCL

<table>
<thead>
<tr>
<th>NVidia CUDA Term</th>
<th>ATI Stream Term</th>
<th>Description / Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device</td>
<td>GPGPU capable graphic card</td>
</tr>
<tr>
<td>Streaming Multiprocessor (SM)</td>
<td>Compute Unit (CU)</td>
<td>A block of multiple cores</td>
</tr>
<tr>
<td>Streaming Processor (SP)</td>
<td>Stream Core (SC)</td>
<td>One computing core</td>
</tr>
<tr>
<td>Global Memory</td>
<td>Global Memory</td>
<td>Installed graphic card memory</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Local Memory</td>
<td>Installed local memory for a SM, usable by all SP in a SM</td>
</tr>
<tr>
<td>Local Memory</td>
<td>Private Memory</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td>Program</td>
<td>Executed program at each core</td>
</tr>
<tr>
<td>Block</td>
<td>Work-group</td>
<td>Group of threads</td>
</tr>
<tr>
<td>Thread</td>
<td>Work-item</td>
<td>Executed program in a core</td>
</tr>
<tr>
<td>Warp</td>
<td>Wavefront</td>
<td>Concurrent threads inside an SM</td>
</tr>
</tbody>
</table>

Table 21 – Technology terms of CUDA and OpenCL

B. CUDA and OpenCL SDK

The CUDA SDK is available from NVidia homepage for GPU computing [http://developer.nvidia.com/object/gpucomputing.html](http://developer.nvidia.com/object/gpucomputing.html). The wide functional range of the CUDA API is shipped with NVidia's own compiler, runtime libraries, development tools and examples. The SDK and drivers are available for Windows, Linux, and Mac platforms in 32 and 64-bit versions. Furthermore, NVidia technologies like PhysiX™, OptiX™, Complex™, GPPUDirect™ and other force the scientific visualization and academicals computations with GPGPU. It is shipped with build rules and settings for automatic compilation of CUDA programs for certain file extensions like .cu with simple changeable configuration settings. For example, Visual Studio can use the “CUDA VS Wizard” for an easy integration.

OpenCL is an open source library created and maintained by the Khronos group. The API files are available from [http://www.khronos.org/opencl/](http://www.khronos.org/opencl/). OpenCL is also compatible to the widely used platforms Windows, Unix, and Mac. OpenCL has participations of many industry-leading companies and institutions including 3DLABS, Activision Blizzard, AMD, Apple, ARM, Broadcom, Codeplay, Electronic Arts, Ericsson, Freescale, Fujitsu, GE, Graphic Remedy, HI, IBM, Intel, Imagination Technologies, Los Alamos National Laboratory, Motorola, Movidius, Nokia, NVIDIA, Petapath, QNX, Qualcomm, RapidMind, Samsung, Seaweed, S3, ST Microelectronics, Takumi, Texas Instruments, Toshiba and Vivante. ([Citation: [KHO]])

Both APIs are written mainly for C and C++ but are already adapted to other programming platforms like Java and .NET. Other programming software libraries like Microsoft® DirectX® with DirectCompute™ offers supports for GPGPU programming also.
C. Fundamental GPU independent types

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar</td>
<td>unsigned char</td>
<td>Shortcut for 8 bit unsigned type</td>
</tr>
<tr>
<td>ushort</td>
<td>unsigned short</td>
<td>Shortcut for 16 bit unsigned type</td>
</tr>
<tr>
<td>uint</td>
<td>unsigned int</td>
<td>Shortcut for 32 bit unsigned type</td>
</tr>
<tr>
<td>ulong</td>
<td>unsigned long</td>
<td>Shortcut for 64 bit unsigned type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Type</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>8-bit value</td>
<td>uchar</td>
<td>uchar</td>
<td>__uint8</td>
</tr>
<tr>
<td>uint16</td>
<td>16-bit value</td>
<td>ushort</td>
<td>uchar</td>
<td>__uint16</td>
</tr>
<tr>
<td>uint32</td>
<td>32-bit values</td>
<td>__uint32</td>
<td>uint</td>
<td>__uint32</td>
</tr>
<tr>
<td>uint64</td>
<td>64-bit value</td>
<td>__uint64</td>
<td>ulong</td>
<td>__uint64</td>
</tr>
</tbody>
</table>

Table 22 – Common unified GPGPU types

D. Common GPU memory status functions

<table>
<thead>
<tr>
<th>Memory function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_is_host_memory</td>
<td>Test if memory is allocated on host memory</td>
</tr>
<tr>
<td>gpu_is_device_memory</td>
<td>Test if memory is allocated on device memory</td>
</tr>
<tr>
<td>gpu_is_read_only_memory</td>
<td>Test if memory has “read-only” state</td>
</tr>
<tr>
<td>gpu_is_write_only_memory</td>
<td>Test if memory has “write-only” state</td>
</tr>
<tr>
<td>gpu_is_reserve_only_memory</td>
<td>Test if memory has “reserve-only” state</td>
</tr>
</tbody>
</table>

Table 23 – Common memory functions

E. General CUDA and OpenCL methods for detecting and initializing devices, to receive informational about the graphic card, and for kernels

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaHasDevice</td>
<td>clDeviceExists</td>
<td>Detect if a GPU device is installed or for a given architecture</td>
</tr>
<tr>
<td>cudaDetectDevice</td>
<td>clDeviceExists</td>
<td></td>
</tr>
<tr>
<td>cudalInitContext</td>
<td>clInitContext</td>
<td>Initialise a GPU context</td>
</tr>
<tr>
<td>cudaClearContext</td>
<td>clClearContext</td>
<td>Clear a GPU context</td>
</tr>
<tr>
<td>cudaGetKernelUsage</td>
<td>clGetKernelUsage</td>
<td>Get kernel usage parameters</td>
</tr>
<tr>
<td>cudaPrintDeviceProperties</td>
<td>clPrintDeviceProperty</td>
<td>Print GPU device properties</td>
</tr>
<tr>
<td>cudaGetDeviceProperties</td>
<td>clGetDeviceProps</td>
<td>Get GPU device properties</td>
</tr>
</tbody>
</table>

Table 24 – General CUDA and OpenCL functions
F. General purpose functions and structures for OpenCL

<table>
<thead>
<tr>
<th>OpenCL methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clGetPlatform</td>
<td>Get OpenCL Platform ID</td>
</tr>
<tr>
<td>clGetDevice</td>
<td>Get a OpenCL Device</td>
</tr>
<tr>
<td>clGetContext</td>
<td>Get a OpenCL context</td>
</tr>
<tr>
<td>clGetCommandQueue</td>
<td>Get a OpenCL command queue</td>
</tr>
<tr>
<td>clLoadProgramFromBinary/Source</td>
<td>Load a OpenCL binary/source file</td>
</tr>
<tr>
<td>clWaitForProgramBuild</td>
<td>Wait for OpenCL compiler finish</td>
</tr>
<tr>
<td>clPrintBuildInfo</td>
<td>Print OpenCL compile infos</td>
</tr>
<tr>
<td>clReadKernelInfo</td>
<td>Get kernel informations</td>
</tr>
<tr>
<td>clReadKernelWorkGroupInfo</td>
<td>Get kernel workgroup informations</td>
</tr>
<tr>
<td>clGetErrString</td>
<td>Get human readable error string</td>
</tr>
</tbody>
</table>

Table 24 – Additional OpenCL functions

<table>
<thead>
<tr>
<th>OpenCL structures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_device_properties</td>
<td>OpenCL device properties structure</td>
</tr>
<tr>
<td>cl_kernel_info</td>
<td>OpenCL kernel informations</td>
</tr>
<tr>
<td>cl_workgroup_info</td>
<td>OpenCL workgroup informations</td>
</tr>
</tbody>
</table>

Table 25 – Additional OpenCL structures

G. GPGPU library dependencies

<table>
<thead>
<tr>
<th>CUDA dependencies</th>
<th>OpenCL dependencies</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuda.lib</td>
<td>opencl.lib</td>
<td>API library</td>
</tr>
<tr>
<td>cudart.lib</td>
<td>-</td>
<td>Runtime library</td>
</tr>
<tr>
<td>cutill32.lib or cutil64.lib</td>
<td>oclUtil32.lib or oclUtil64.lib</td>
<td>Utils library</td>
</tr>
</tbody>
</table>

Table 26 – GPGPU libraries

These libraries are located in the SDK of the APIs, available in 32 or 64-bit versions.

H. Common types for the Myers implementation

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyersBool</td>
<td>char</td>
<td>1 bytes</td>
<td>Boolean representation</td>
</tr>
<tr>
<td>MyersID</td>
<td>int</td>
<td>4 bytes</td>
<td>Identifier representation</td>
</tr>
<tr>
<td>MyersChar</td>
<td>char</td>
<td>1 bytes</td>
<td>String character type represenations</td>
</tr>
<tr>
<td>MyersString</td>
<td>char*</td>
<td>-</td>
<td>String pointer reference</td>
</tr>
<tr>
<td>MyersSymbol</td>
<td>uchar</td>
<td>1 bytes</td>
<td>Alphabet symbol representations</td>
</tr>
<tr>
<td>MyersLen</td>
<td>uint</td>
<td>4 bytes</td>
<td>Type for length of strings or sizes</td>
</tr>
<tr>
<td>MyersSize</td>
<td>size_t</td>
<td>8 bytes</td>
<td>Size declarations for memory or offsets</td>
</tr>
<tr>
<td>MyersScore</td>
<td>ushort</td>
<td>2 bytes</td>
<td>Score type representation</td>
</tr>
</tbody>
</table>

Table 27 – Common types for Myers implementation
I. Notes to assignable threads per block, based on the local memory usage

The following table demonstrates the maximal assignable threads per block with certain local memory usage. It lists for different alphabets and pattern lengths, resp. bit-vector sizes, on a CUDA 1.0 architecture with 16 KB of shared memory per block, the number of threads that not exceed the hardware limitation of one SM.

<table>
<thead>
<tr>
<th>Pattern length</th>
<th>DNA (Length 4+1)</th>
<th>Iupac (Length 16+1)</th>
<th>AminoAcid (Length 24+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory</td>
<td># threads</td>
<td>Memory</td>
</tr>
<tr>
<td>m=8</td>
<td>5 bytes</td>
<td>3276</td>
<td>17 bytes</td>
</tr>
<tr>
<td>m=16</td>
<td>10 bytes</td>
<td>1638</td>
<td>34 bytes</td>
</tr>
<tr>
<td>m=32</td>
<td>20 bytes</td>
<td>819</td>
<td>68 bytes</td>
</tr>
<tr>
<td>m=64</td>
<td>40 bytes</td>
<td>409</td>
<td>136 bytes</td>
</tr>
<tr>
<td>m=128</td>
<td>80 bytes</td>
<td>204</td>
<td>272 bytes</td>
</tr>
<tr>
<td>m=256</td>
<td>224 bytes</td>
<td>73</td>
<td>608 bytes</td>
</tr>
<tr>
<td>m=512</td>
<td>448 bytes</td>
<td>36</td>
<td>1216 bytes</td>
</tr>
<tr>
<td>m=1024</td>
<td>896 bytes</td>
<td>18</td>
<td>2432 bytes</td>
</tr>
</tbody>
</table>

Table 28 – Assignable threads per block bases on local memory usage

Note that other parameters like the amount of register used by a kernel program influence the real applicative number of threads as well.

The kernel memory usage scales linearly for small bit-vector sizes. With larger pattern lengths, some overhead in memory is required for operational usage and controlling in the bit-vector classes. The number of assignable threads decreases vastly with larger memory usage and therefore decreases the achievable level of parallelism on GPU. Additionally, using even larger bit-vectors or alphabets will soon exceed the available memory and the kernel will not be compiled any more. The hardware limitation is clearly a negative influencing factor when implementing an algorithm on GPU, but these hardware abilities increase by the time as the CUDA architecture 2.0 support already 48 KB per SM.

J. Theoretical runtime vs. real runtime

The theoretical runtime of Myers algorithm is $O(n [m/w])$. Therefore factor $[m/w]$ should be observable in the performance results. In practice, the effecting factor is justified to a lower level. Hardware and compiler optimizations influence the behavior. For the most bit-vector operations, each vector item can independently be computed to each other. This is able to benefit from a kind of improved code pipelining with hardware parallelism on the cores, also known as hyper-threading abilities for example. In addition, the percentage of the sequential parts in the runtime decreases with higher sequence and pattern lengths and by more distance measurements, as the heaviness of computing parts in assumption parallel executed increases the runtime. In total this effects an overall lower runtime than expected by the factor $[m/w]$. 
K. CUDA kernel naming scheme

The following example shows how the NVidia compiler assembles the complete internal kernel function name for one Myers distance computing instance:

<table>
<thead>
<tr>
<th>Internal Kernel Name</th>
<th>Label descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ZN 5seqan 4cuda</td>
<td>Starting prefix</td>
</tr>
<tr>
<td>12myers_kernel ILt256E</td>
<td>Namespace “seqan”</td>
</tr>
<tr>
<td>NS_10SimpleTypeIh NS_4Dna_EEEK NS_3TagI NS_18Hyyro_Levenshtein_EEEK NS5_INS_17Myers_Local_Mode_EEEEEvPhjPcP NS_7Index_tEjSD_SF_JP NS_14MyersScoringEjPtsF_PjsD_cy</td>
<td>Namespace “cuda”</td>
</tr>
<tr>
<td></td>
<td>Kernel function name “myers_kernel”</td>
</tr>
<tr>
<td></td>
<td>Bitvector size “256”</td>
</tr>
<tr>
<td></td>
<td>SeqAn structure “SimpleType”</td>
</tr>
<tr>
<td></td>
<td>Alphabet tag name “Dna_”</td>
</tr>
<tr>
<td></td>
<td>SeqAn declaration “Tag”</td>
</tr>
<tr>
<td></td>
<td>Scoring scheme tag name</td>
</tr>
<tr>
<td></td>
<td>Kernel variant tag name</td>
</tr>
<tr>
<td></td>
<td>Buffer index struct “index_t”</td>
</tr>
<tr>
<td></td>
<td>Scoring function name</td>
</tr>
</tbody>
</table>

The marked text segments are composed by the length of the label first and then the label. Changing the internal program structure and names sometimes requires to adapt the determination function accordingly to operate in a defined state.

L. OpenCL source enhancements by #defines

The following #defines are created and inserted on top of the OpenCL source code in advance to specialize, commonly aligned to the application internal used declarations:

<table>
<thead>
<tr>
<th>Constants for OpenCL</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHABET_SIZE</td>
<td>256</td>
<td>Used alphabet size, for example 5 for DNA5</td>
</tr>
<tr>
<td>ALPHABET_BUFFER_SIZE</td>
<td>256</td>
<td>Used buffer size for storing the alphabet map</td>
</tr>
<tr>
<td>MYERS_SCORING_SCHEME</td>
<td>Hyyro_Levenshtein_Distance</td>
<td>Used scoring scheme, based on the enum Myers_Scoring_Scheme</td>
</tr>
<tr>
<td>MYERS_KERNEL_VARIANT</td>
<td>Myers_Local_Memory_Mode</td>
<td>Used kernel memory variant, based on the enum Myers_Memory_Mode</td>
</tr>
<tr>
<td>MYERS_USE_CUTOFF</td>
<td>false</td>
<td>Enable using cut off mechanism, based on the MYERS_USE_CUTOFF constant</td>
</tr>
<tr>
<td>MYERS_BITVECTOR_TYPE</td>
<td>ulong</td>
<td>Used type for bit-vector representation</td>
</tr>
</tbody>
</table>

Table 29 – OpenCL constants
**Eidesstattliche Erklärung zur Diplomarbeit**

Hiermit versichere ich, dass ich die Diplomarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst habe, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

Ich bin damit einverstanden, dass ein Exemplar meiner Diplomarbeit in der Bibliothek ausgeliehen werden kann.

Berlin, den

Unterschrift des Autors

Diese Diplomarbeit wurde erstellt von:

Name : Lars Langner  
Matrikelnummer : 3905118  
Studiengang : Informatik mit Nebenfach Bioinformatik  
Universität : Freie Universität Berlin  
Fachbereich : Informatik/Mathematik ; Bioinformatik  
Arbeitszeitraum : 01.10.2010 bis 01.04.2011