GOBI Report

Efficient all-against-all protein similarity matrix computation using OpenCL

Uli Köhler & Anton Smirnov

February 14th, 2014

Contents

1. Introduction

Today, it is the amount of available data rather than its acquisition that poses a significant challenge to computer science. It is the issue of extracting valuable and useful information from increasing data volumes, and it has manifested itself throughout most of modern global digital frameworks, like economic analysis, weather forecast, or, indeed, national security, recently publicized in the 2013 NSA affair.

This problem remains very pronounced in the field of bioinformatics, where it is compounded by the rapid progress in the fields of DNA and protein sequencing within the last 20 years. Methods like next generation sequencing provide a low-cost experimental approach and their wide adaptation has led to an exponential growth in biological data. The frontier, so to speak, is now the attempt to understand all this data well enough to make it useful in a variety of scientific and industrial contexts, including, but not limited to, evolutionary biology, biochemistry, pharmaceutics and medicine.

To deal with this obstacle, a possible approach is to divide the computation in question into as many independent parts as possible, and then compute those parts on many machines. This idea of distributed computing may be seen in many examples, like Rosetta@home[1] or Folding@home[2], both using the *BOINC* framework (Berkeley Open Infrastructure for Network Computing) to distribute their computations.

A second approach, one that may also be used in tandem with distributed computing, is the idea to parallelize computation on multiple cores, be it the CPU or the GPU. This concept is becoming increasingly important with the growing availability of multi-core processors and the number of cores installed on common chipsets.

In the field of bioinformatics, the issue of the computational demands of analyzing tremendous amounts of data exceeding computational capacities may be found in different forms: be it the analysis of genome-wide association studies, genome-wide searches for ORFs, or in database-wide comparisons of all sorts of biological data, especially when an all-against-all comparison is required.

One such instance, requiring the comparison of all known protein sequence to each other, may be found in the *SIMAP* project.

2. *SIMAP*

SIMAP, the Similarity Matrix of Proteins (see [ART+05, RTA+08, RTG+10]), is an effort aimed at creating a comprehensive database containing a measure of similarity for all pairs of known protein sequences. This similarity is calculated by performing a pairwise alignment of all

[1] See [RSMB04]
[2] See [LSS+02]

sequences versus all sequences, yielding a similarity score for any pair of proteins. Such a similarity score is useful since it allows, for example, the functional prediction on aminoacid sequences based on homology. Since 2011, *SIMAP* has been crosslinked with the STRING database (see [VMHJ+03, JKS+09, SFK+11]).

As of now, the pairwise alignment of sequences is performed using *FASTA* (see [LP85]), with the Smith-Waterman score recomputed for sequences showing a similarity exceeding a predefined threshold[3]. As of writing this report, 80 is used as threshold – while a lower threshold could improve overall accuracy, it would require a significantly larger number of Smith-Waterman alignments to be computed.

Smith-Waterman, being an exhaustive algorithm yielding the optimal result as opposed to *FASTA* using heuristics to speed up the computation at the cost of accuracy, is computationally more demanding with an asymptotic runtime complexity of $\mathcal{O}(n^2)$ and is therefore used for high-scoring hits only. While this accuracy-vs-speed tradeoff does not pose a problem for sequences with low overall similarity, *SIMAP* requires accurate scores for high-scoring hits in order to be able to accurately predict function based on homology.

Even if the majority of the pairwise alignments is computed using *FASTA*, the current *SIMAP* database contains more than $7 \cdot 10^7$ sequences, requiring the computation of more than $4.9 \cdot 10^{15}$ alignments. In order to cope with the massive amount of alignments to be computed, *SIMAP* uses the BOINC[4] framework to distribute the work to a large number of volunteers donating their CPU time while their computer is idle. This approach not only reduces the total project costs dramatically, but also allows *SIMAP* to compute similarities of new protein sequences in only a few weeks to months because of the massive amount of computational resources available.

Although *BOINC* lets three different computers run any single workunits to be able to filter out computers producing incorrect results and therefore triples the overall amount of computations to be performed, the overall processing speed is significantly faster for projects using *BOINC*.

IO as limiting factor in *SIMAP* While one might argue that *SIMAP* processes large amounts of data, and therefore not computation but IO is the limiting resource.

However, it's clearly evident that n sequences require $\mathcal{O}(n^2)$ to be computed and therefore the input part of the IO and available storage space does not limit the number of sequences that can be used in *SIMAP*.

However, as each of said $\mathcal{O}(n^2)$ produces an output, one must also take into account the amount of output generated.

Currently, *SIMAP* uses about 60 TiB of storage space to storage the alignment scores for any protein pair, plus some metadata in compressed form. While this is outside the range of

[3] A comparison of Smith-Waterman and *FASTA* regarding selectivity & sensitivity can be found in [Pea91]
[4] Berkeley Open Infrastructure for Network Computing

standard computers, specialized storage arrays are easily capable of storing this amount of data.

The same applies for the network IO to and from the *BOINC* clients. Even in uncompressed form, each client will get less than a few million sequences and compute any pairwise alignment score for said sequences, yielding less than a few megabytes of alignment scores.

SIMAP currently computes the alignment itself (in contrast to the score) on-demand, i.e. when accessing their web interface.

Saving the alignment in addition to the score only would require a significantly larger amount of storage space. A hypothetical solution would be to only store alignments for high-scoring hits, however it is questionable if the overall computational effort of recomputing and storing the backtracked alignments can be justified by only a few usecases requiring a mass-retrieval of alignments.

Although the storage problem in *SIMAP* is still largely unresolved and we believe that it requires further exploration, it's evident that because of the massive number of computational effort (score-only Smith-Waterman being $\in \mathcal{O}(n^2)$, yielding an overall $\mathcal{O}(n^4)$ for n sequences) required for each input sequence computational speed rather than bare IO is the limiting factor for *SIMAP*.

3. Alternative computational hardware

Most software being written today is designed for CPU platforms. While CPUs are widely available and have well-understood performance characteristics (see [Fog14]), they are designed as a general computational platform and therefore don't provide sufficient computational power for some problems. *SIMAP*, for example, currently runs on CPUs exclusively – although no computational hardware can fully resolve this problem, currently adding only a few thousand protein sequences results in billions of alignments having to be computed, resulting in a week-long computation duration.

In the context of shotgun protein sequencing techniques (see [BCP07, MY02]), allowing a large amount of protein sequences to be generated in a short amount of time, *SIMAP* might not be able to keep up with the steady inflow of protein sequences.

While there are specialized hardware accelerators, most of which are based on FPGAs[5] (see [JLX+07, HJL+07, ACL+09]) or even on ASICs[6] like *SAMBA* (see [GJL97]), those methods are expensive and obsolesce quickly: While *SAMBA* provided outstanding performance in 1997, one can expect even recent mobile handheld devices to outperform its computation speed. FPGA- and ASIC-based approaches gain their speed advantage by being able to be configured on a boolean logic level instead of using instructions sets for execution, making the

[5] Floating Programmable Gate Array
[6] Application Specific Integrated Circuit

development process complex and difficult to debug – however, this approach allows nearly unlimited parallelism and minimized latency.

In the case of *SIMAP* these technologies can't be used in conjunction with *BOINC*, because none of the *BOINC* volunteers have such accelerators at their disposal. One area where ASIC and FPGA accelerators are unrivaled is in areas where realtime computation is required (see [VDRN11]).

A more commonly available device type whose use in bioinformatics has grown rapidly during the last years is the graphics card. Although it is normally used to compute large 3D environments and display those on a screen, its special architecture – allowing it to compute said 3D environments efficiently – can be leveraged by specialized GPGPU[7] applications.

While most modern CPUs have between 2 and 16 physical cores, GPUs have several hundreds to thousands. Although each single GPU core has fewer capabilities than a CPU core, the high parallelism and specialized hardware components for efficient linear algebra computations allow researchers to benefit from the massive computational power in GPUs (see [Sch11, Lan12, CTS05, VLM11, Far10]). Even if not all problems are suitable for GPUs – including for reasons like the inability of GPU cores to run recursive algorithms and dynamic memory allocation – speedups by multiple orders of magnitude are possible for some applications (see [BBH12]), although most applications will only achieve a speedup of less than one order of magnitude (see [WDJ+07]).

3.1. GPGPU programming frameworks

When porting an algorithm to GPU-based platforms, the developer has to choose an underlying framework that allows running the computation on GPUs.

Although there is a multitude of different backends optimized for specialized tasks and/or devices, three major frameworks allow generalized implementation of algorithms using GPGPU technology.

- *CUDA*[8], the first generalized GPGPU framework available and therefore widely used (see [LR09, LMS09, BBH12]), however limited to only GPUs manufactured by NVidia Corp.
- *ATI Stream*, the ATI/AMD Corp. analogon to *CUDA* – not very widely used and limited to AMD/ATI GPUs
- *OpenCL*[9], a vendor-independent standard that allows programs to run on any computational hardware providing a compiler for the *OpenCL* source code

[7] General-Purpose GPU
[8] Compute unified device architecture
[9] Open Computing Language

While *OpenCL* (see [KSA+10]) is a more general approach to CUDA and might therefore provide worse performance in some usecases, applications can run vendor-independently on CPUs, GPUs and even some FPGA-based platforms. Although CUDA is used more widely[10], the distributed-computing approach can leverage more idle resources by using a vendor-independent computing framework, because the hypothetical minor performance advantages of CUDA (see [FVS11, TNI+10, KDH10, SJL11, DMM+10, RVDDB10]) are expected to be outweighed by the large number AMD GPUs that would otherwise not be used. Although many projects remain using CUDA, some have ported their code to *OpenCL* for these reasons (see [Har09, DWL+12]).

OpenCL has a pluggable architecture: Each device that is capable of running programs written in *OpenCL* provides a so-called ICD[11] that provides a compiler to transform *OpenCL* sourcecode into a device-specific binary and the necessary APIs to transfer data to and from the device. By using the *OpenCL* library provided by the *OpenCL* consortium, a program can access any ICD installed on the system, without being limited to only one of the computational accelerators available on the specific system.

Both *OpenCL* and CUDA allow the developer to specify a multi-dimensional discrete space of tasks to be performed. The device-specific scheduler then assigns said tasks to the available hardware cores and optimizes the sourcecode according to the specific characteristics of the accelerator device in use.

Besides using distributed computing for GPGPU computation, *OpenCL* also provides the flexibility to use not only NVidia computation-specific GPU platforms like the Kepler architecture (see [TLHC14]), but also platforms like the Intel Xeon Phi (see [CSKM12, JR13]).

4. *CLSW*

A significant amount of research has been performed in creating a GPGPU application accelerating the Smith-Waterman algorithm (see [LR09, MV08, LBH09, DBL+10, RMG+10]). However, besides CUDASW++ (see [LMS09]), those projects don't seem to be ready for production use but only built as proof-of-concept for GPGPU-Smith-Waterman – although CUDASW++ would be usable, it is not only specific to NVidia GPUs, but newer versions have been optimized specifically for the Kepler architecture of NVidia computation accelerator cards (see [LWS13]) – therefore, the software is expected to exhibit worse performance when using on non-Kepler cards, such as those in use by the *BOINC* volunteers providing their GPU for research purposes.

Therefore, the goal of our lab project was to develop an *OpenCL*-based Smith-Waterman application specifically for the *SIMAP* usecase. We call our newly developed software *CLSW*[12],

[10] An important factor to explain this is a massive marketing effort from NVidia to distribute their proprietary platforms

[11] Installable client driver – for GPUs, the ICD is usually installed with the graphics card driver

[12] [Open]CL Smith-Waterman

stressing the *OpenCL*-based backend.

Although it was impossible to develop a complex application beyond a proof-of-concept state in the severely limited timeframe of two weeks, *CLSW* turned out to exhibit outstanding performance characteristics in some special cases (see section 5.5 on page 20) and can therefore be considered usable beyond *SIMAP*.

In this section, we will introduce core concepts in use by *SIMAP*, specifically our algorithms *QUANTMASS* (see section 5.3 on page 15), *GIVE-SWAP* (see section 5.7.5 on page 34), and *TESSIN* (see section 5.7.6 on page 35). Although only *QUANTMASS* is fully implemented in the *CLSW* toolkit, they represent methods of optimizing *CLSW* even further in order to compete with existing tools in terms of runtime performance.

5. Smith-Waterman parallelization strategies

As GPGPU programming requires the underlying algorithms to be parallelizable, we need to review different parallelization strategies for the Smith-Waterman algorithm.

Although any strategy has advantages and disadvantages and different approaches have been followed by previous implementations (see e.g. [RMG+10]), for the *SIMAP*-specific case we have to consider that a large amount of computational resources is required not because of a few large alignments (e.g. genome-to-genome alignments), but because of a huge number of shorter alignments.

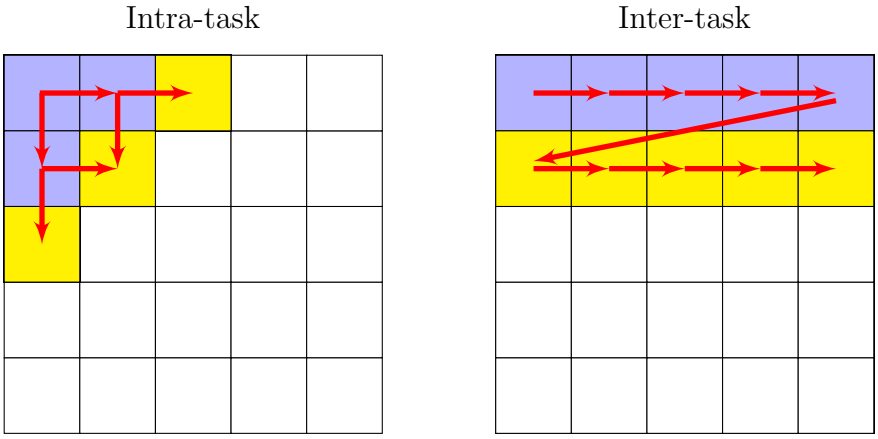

Figure 1: Smith-Waterman parallelization strategies

We can generally separate Smith-Waterman parallelization strategies into two main classes:

- intra-task parallelization and
- inter-task parallelization.

Intra-task parallelization, as shown on the left of figure 1, refers to the the attempt to parallelize the calculation of a single alignment matrix. This approach is made possible by

the fact that within said matrix, each cell depends solely on the cell directly above it, the cell diagonally to the upper left and the cell directly to the left of said cell.

While at the beginning one can't compute more than one cell, after each iteration, one additional cell can be computed in parallel. Without loss of generality it can be shown that for a symmetric[13] matrix the maximum parallelism is $\lfloor\sqrt{2 \cdot n^2}\rfloor$, because maximum parallelism is reached when computing the bottom-left-to-top-right diagonal in parallel. After reaching this maximum, the parallelism decreases.

In contrast to intra-task parallelization computing a single matrix in parallel, inter-task parallelization follows the simple approach of computing multiple matrices (i.e. alignment scores) concurrently – each alignment is only computed by a single core, rendering synchronization mechanisms redundant. This approach only works sufficiently well if enough alignment tasks are available to use all cores of the underlying hardware. While this might not be the case for other projects, *SIMAP* requires a large number of alignments to be computed, therefore enabling the possibility to use the simpler inter-task parallelism approach.

Inter-task-parallelism can run in linear memory in respect to the query sequence size, because when computing one row at a time, only the current row and the previous row need to be saved. See section 5.4.4 on page 18 and section 5.4.2 on page 18 for a detailed discussion regarding *CLSW* memory usage. The effective parallelism is only limited by the number of alignments available.

Although it's possible to combine multiple intra-task-parallelized computations in an inter-task-parallelized manner, we expect this concept to be difficult to implement and prone to errors, because both *OpenCL* and CUDA provide only limited control over the execution order of threads. Moreover, any control exerted is likely to reduce the overall effective parallelism due to forced serialization.

Initial development versions of *CLSW* were based on intra-task parallelization because this approach has been used in previous work relating to *OpenCL* (see [RMG+10]). However, our application initially did not compute correct results according to the validation described in section 5.4.6 on page 19, but it proved to be significantly slower than the current inter-task approach, because of computational overhead incurred by converting the coordinates in the matrix diagonal currently being computed in parallel (resembling discretized polar coordinates) to the Cartesian coordinates required in order to compute the aminoacid substitution correctly. Furthermore, most hardware cores ran in an idle state throughout a large part of the matrix, because we had to reserve $\lfloor\sqrt{2 \cdot n^2}\rfloor$ cores initially to achieve maximum hardware parallelism in the matrix diagonal.

We believe that most of these issues can be removed by further optimizations; inter-task implementation has, however, proven to be significantly easier to implement than the intra-task approach. If applicable to the specific problem, we therefore conclude that the inter-task-based

[13] i.e. a matrix having the same width as its height

	A	K	L	ε	ε
A	⋱	⋱	⋱	0	0
C	⋱	⋱	⋱	0	0
M	⋱	⋱	⋱	0	0
M	⋱	⋱	⋱	0	0
L	⋱	⋱	⋱	0	0

Table 1: Sequence extension concept. Shown is a hypothetical Smith-Waterman alignment of two sequences within a sizeclass with the length 5. Therefore, the shorter sequence of 3 aminoacids is "padded" with two ε characters, marked in yellow. Because their value is $-\infty$, the values of the cells within the respective columns become 0, thereby not altering the matrix maximum.

approach provides a simpler, yet overall more efficient approach to the problem on GPUs.

5.1. Sequence extension sizeclasses

In order to use memory efficiently – on GPUs this is even more important than on CPUs, as shown in 5.2 on page 14 – we need to make sure the input sequence arrays[14] are stored in-memory in a way that allows the start address and length of any sequence to be computing using linear addressing[15] (i.e. $offset + i \cdot n$ where n is a constant and i is the number of the sequence that needs to be accessed). In order to fulfill this condition, it is obviously required that any sequence in the sequence array is extended to the same sequence length.

In *CLSW*, we have implemented a concept called sequence extension that allows the software to extend sequences using a special character ε that can be appended to the sequence. Because we're interested in the overall alignment score, it's necessary that ε does not change the alignment score at all.

As the character in the sequence only influences the substitution part of the Smith-Waterman recursion equations (see [SW81]), it is sufficient to set the substitution score $\omega(\cdot, \varepsilon) = \omega(\varepsilon, \cdot) = -\infty$. As Smith-Waterman is maximized over zero, the cells containing ε are guaranteed to have a smaller value than the maximum value in the matrix, assuming the gap cost function has a negative sign[16]. A non-negative sign removes any significance from the score, because the Smith-Waterman algorithm will simply maximize the number of gaps instead of computing a proper alignment.

Figure 1 illustrates an exemplary query sequence extended from 3 characters to 5 characters by appending εε. In this case the values in the matrix are always zero for the extended parts,

[14] A list of sequences that can be stored in-memory sequentially

[15] Most hardware supports this addressing mode directly, making sequence access very efficient, see e.g. [HHM+02]

[16] One could also use the negated gap cost function – interpreting it as gap penalty – , in which case the sign would need to be positive

although this is not strictly required.

The *SIMAP* dataset contains sequences with lengths ranging from 20 aminoacids to more than 40,000 aminoacids If one would randomly select sequences and pad them to the maximum length in order to fulfill the linear memory addressing condition, a significant proportion of the overall memory used would only consist of ε characters. If the implementation would compute the matrix slices in the extended sequence parts (that are irrelevant to the alignment score), the computational overhead of sequence extension would be massive[17].

If, however, sequences would not be padded at all, the job generate would need to generate a single computational job even for sequence arrays containing only one sequence[18]. Although this effect might be desirable to some extent – as we will discuss in section 5.7.2 on page 31 – it generally produces a significant amount of computational overhead because of job initialization and cleanup procedures that need to be executed, plus the inability of fully leveraging parallelism using the inter-task parallelization concept, when less alignments tasks than cores are available.

As we can neither use one large sequence array for all lengths because of massive overhead, nor one sequence array per length (because that would lead to too many jobs being generated), we need to group sequence lengths together. Because grouping random, non-adjacent lengths would only result in a significant amount of overhead, in order to achieve maximum efficiency, we need to put only adjacent lengths into one group, i.e. we're classifying sequence length into bins or *sizeclasses* – hence the name *sequence extension sizeclasses*.

Appropriate selection of those sizeclass bins will minimize the overall overhead. Obviously, one sequence length must be assigned to exactly one sizeclass using the surjective transformation $SequenceLength \rightarrow Sizeclass$.

Figure 2 on the following page shows the *SIMAP* sequence length distribution in green, whereas an exemplary distribution of sizeclass boundaries is shown in red. Each red dashed line separates two sizeclasses. Even if the image only shows an example, it is clearly visible that in the area with higher density of the function $\frac{\text{number of sequences}}{\text{sequence length}}$, the average density of sizeclass boundaries is high, because placing them further apart would mean that a large number of sequences would need to be extended, yielding high overhead.

In section 5.3 on page 15 we will present *QUANTMASS*, a linear-time algorithm to heuristically determine optimal sizeclass boundaries on a sequence length population like shown in green in figure 2 on the following page using equivalent-distance assignments in the discrete population quantile domain.

Initially, we expected *CLSW* to exhibit significantly better performance when computing the full matrix, including extension characters. We expected this characteristic due to the requirement that GPU cores execute the same instruction concurrently under some conditions

[17] As discussed in section 5.4 on page 17 regarding the branch-free design principle, it's not possible for performance reason to include `if` statements that stop processing once an extension region is reached

[18] While in the range of 20-1000 aminoacids there are enough sequences for any length to do this, most sequence lengths > 10,000 occur only once

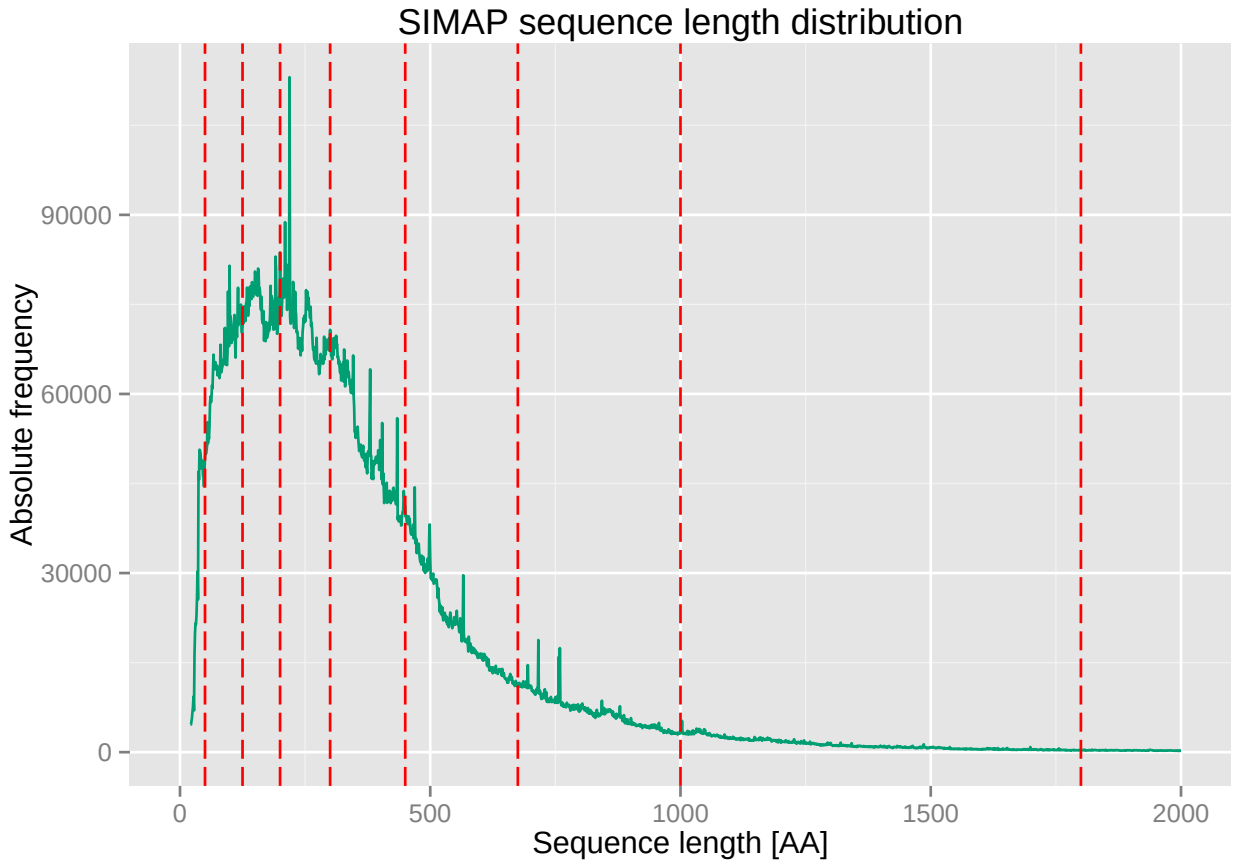

Figure 2: Amino acid sequence length distribution within *SIMAP*. For demonstration purposes, the concept of sizeclasses is illustrated by dividing this distribution into putative "bins", or sizeclasses (dashed red lines).

(as shown in [NVi12] – although the cited reference is NVidia-specific, we assumed that the microarchitectures of GPUs are similar in this respect) which we were unable to define reasonably in the *OpenCL* context due to the high-level-abstractions imposed by the language.

However, our tests showed that a version of the *CLSW* kernel that requires information on how long the non-extended part of a sequence is (a *length-aware* kernel), is within 0.3% of the runtime of a kernel that actually computes the extensions (a *length-unaware* kernel). Moreover, for sequence extension sizeclasses containing a large number of extensions, we expect the length-unaware kernel to be significantly slower than the length-aware kernel, because submatrices generated by the extensions need to be fully computed.

These results show that – in contrast to our expectations – sequence extension sizeclass don't provide better performance than length-aware kernels. However, they may be highly device-dependent, so other *OpenCL*-capable devices may show different performance characteristics.

Although we are not able to deduce the exact causes of aforementioned effects, we suspect that the phenomenon is related to the AMD *OpenCL* ICD using LLVM (see [LA04, Lat02, LA02]).

LLVM is a partially architecture-agnostic intermediate compiler representation format and framework which is known to be highly-optimizable, using advanced techniques like polyhedral optimization (see [GZA+11]). It is possible that these optimizations yield a more efficient binary representation of length-aware kernels.

Independently of the results on other devices, sequence sizeclasses are still required in order to use memory efficiently and to reduce memory overhead, as discussed in section 5.2 and section 5.7.6 on page 35.

5.2. Kernel sizeclasses

Based on previous experiences with GPU programming, we expect the overall performance of the application to depend largely on memory allocation and usage behavior

GPUs have an even more complex memory architecture than CPUs: Besides multiple levels of caches, with their exact characteristics depending on the GPU model, most GPUs are able to allocate memory not only in the main DRAM, but also in an on-chip SRAM.

Said SRAM area is generally faster than the DRAM because it interfaces with the GPU's cores directly, however it is significantly smaller in size than the DRAM. By optimizing memory allocation behavior, we can increase the likelihood of the compiler allocating an array in the most efficient memory area available.

As the kernel source code is compiled by the *OpenCL* ICD and we expect this compiler to know details about the memory areas available on the specific *OpenCL* device the kernel shall be compiled for, we can safely assume the compiler places the buffers used in the kernel into the most appropriate memory area.

Based on these assumptions, we introduce *kernel sizeclasses*, a concept of compiling a single kernel for multiple query sequence sizes, in order to reduce overhead and – at the same time – maximize computation efficiency by increasing the likelihood of buffers being allocated in fast RAM areas.

If one single kernel would be compiled for all sequence sizes, it would lead to considerable inefficiency: The maximum sequence size is about 40 000 aminoacids for the 2014-02 *SIMAP* data dump; a kernel allocating more than 480 kilobytes of memory (sequence size, multiplied by 3 buffers for affine gap costs, multiplied by 4 byte, because buffers consist of 32-bit floating-point variables) for each of several hundreds to thousands of jobs running in parallel would only fit in the slowest but largest memory accessible to the GPU[19].

On the other hand, if a kernel would be compiled for each sequence size, then the process of kernel compilation – each individual compilation taking about one to two seconds for recent GPU ICDs – would require a large proportion of the overall computation time.

[19] One might argue that such a kernel would not fit into memory at all for low-end graphics cards – however, properly programmed *OpenCL* drivers will reduce parallelism to reduce the overall memory usage in the case mentioned

If it is detrimental to the performance to use a single kernel, but it's also hardly possible to use tens of thousands of kernels, we need to group adjacent sequence sizes together, i.e. assign said sizes to classes – therefore we have the same situation as with sequence extension sizeclasses, yielding the concept of kernel sizeclasses.

We expect the effect of kernel sizeclasses on the performance to depend significantly on both the device and the ICD version.

5.3. QUANTMASS - A metric for sizeclass selection

A question that is unanswered so far is how to optimally select both kernel sizeclasses and sequence extension sizeclasses. As this is obviously an optimization problem, we first need to define an error function to be minimized, which needs to incorporate the parameters:

$$Penalty(sc) := \sum_{s \in sc} length(s) - length(sc)$$
$$SizeScore(sc) := maxlength(sc) - minlength(sc)$$

where $maxlength(sc)$ is the maximum length of any sequence (with $minlength(sc)$ defined accordingly) in sc and $length(s)$ is the length of sequence s in the sizeclass sc.

While $Penalty(sc)$ is equivalent to the negated total number of sequence extensions in said sizeclass and therefore ensures the optimizer does not choose a single sizeclass for all sequences, $SizeScore(sc)$ ensures sizeclasses get a positive score proportional to the number of lengths included in them. These functions assume that a size is assigned to exactly one sizeclass.

Based on these functions we can define a basic error function, with $\overrightarrow{sc}$ being the vector containing all sizeclasses

$$err(\overrightarrow{sc}) = \sum_{sc \in \overrightarrow{sc}} \alpha * Penalty(sc) + \beta * SizeScore(sc)$$

When choosing α and β appropriately, we can therefore run a standard optimization algorithm to find an optimal set of sizeclasses. However, there is no inherently correct way to choose those parameters. Therefore we require a simple algorithm to choose sizeclasses heuristically, without the requirement of choosing such parameters.

It is possible to simply add sizeclass boundaries on each nth sequence length. This approach does not require extensive computation, but obviously does not minimize aforementioned error function, because its result does not depend on the underlying sequence length distribution, therefore not being an adaptive binning algorithm.

In this section, we will present $QUANTMASS$[20], a simple algorithm that allows penalty-

[20] QUANTile Metrics for Adaptive Sizeclass Selection

bounded selection.

QUANTMASS requires three simple steps:

1. Select n initial sizeclass boundaries: For each $1 \leq i < n$, place a sizeclass boundary at the sequence length boundary nearest to the quantile $q_{i/n}$
2. Merge sizeclass boundaries that have been placed on the same sequence length boundary
3. Compute $Penalty(sc)$ for each sizeclass sc. For each sizeclass larger than a given penalty threshold, recursively subdivide said sizeclass until no sizeclasses with penalties larger than the given threshold are left.

In step 1, we can easily compute the discrete population quantiles for any sequence length boundary by using:

$$q_l := \frac{\sum_{i=0}^{i<=l} NumSequences(i)}{TotalSequences}$$

$$TotalSequences := \sum_{i=0}^{i<=MaxSeqLen} NumSequences(i)$$

which computes the discrete population quantile q_l directly after the sequence length where $NumSequences(l)$ denotes the number of sequences having a length of exactly l and $MaxSeqLen$ is the maximum sequence length occurring in the dataset[21].

After computing q_l for any sequence length l in the underlying dataset, we can assign the sizeclass boundaries with equivalent distances in the quantile domain instead of the sequence length domain.

Because of the non-continuous nature of the underlying datasets, multiple quantiles may be assigned to a single sequence length boundary, therefore we need to merge those which would create empty or congruent sizeclasses.

As we observed for the *SIMAP* dataset, the approach only consisting of steps 1 and 2 generated good results for sequence lengths shorter than about $10,000$ – sizeclasses including larger sequences showed significantly higher penalty values: While sizeclasses for shorter sequences had penalties < 100, the penalty values of larger sizeclasses exceeded 10^9 for some sizeclasses.

We can resolve this issue by selecting a penalty threshold – for the *SIMAP* dataset, we chose 2000 – and recursively subdividing this sizeclass, reducing the penalty in the process. For this step it is obviously desirable to minimize the number of subdivisions as this approach will maximize $SizeScore$ for the sizeclass to be divided.

[21] Simply speaking, this formula computes what fraction of the total sequences occur before (including) the length l

For the *SIMAP* dataset, subdividing the sizeclasses at the 73% quantile proved to yield the minimal number of sizeclasses; however, this selection might depend not only on the dataset but also on other parameters of *QUANTMASS*.

Although *QUANTMASS* does not necessarily provide an optimal solution to the sizeclass selection problem, it can compute a heuristic in $\mathcal{O}(n)$ time, with n being the number of sequence lengths in the dataset, and is therefore suitable for large datasets like *SIMAP*.

5.4. Design

CLSW is implemented in C++11 using only the *OpenCL* library and SeqAn (see section 5.4.6 on page 19) and therefore has no external dependencies on libraries beyond *OpenCL*.

Because *OpenCL* ICDs are present on most systems anyway – assuming the graphics card driver is installed – this results in high portability and maintainability while providing high modularity. These aspects of *CLSW* are most important if *CLSW* is used as a *BOINC* GPU application in the future, as any external dependency might introduce problems on the different target system configurations the *BOINC* client will run on.

CLSW uses a branch[22]-minimized implementation approach that is especially important on GPUs because any of the GPU cores that run an *OpenCL* kernel need to execute the same instruction at the same point in time – if a kernel fails to fulfill this condition, the GPU scheduler is forced to serialize the execution until common instructions are reached, effectively eliminating any performance advantage gained by core parallelism.

Additionally, *CLSW* does not use ASCII-encoded aminoacid sequences but rather encodes any character in the sequence with the 0-based index in the substitution matrix, allowing the application to compute the substitution score in only three memory fetches[23].

5.4.1. IO formats

CLSW can read *MATBLAS*-formatted substitution matrices and uncompressed *FASTA* file directly and is therefore compatible with the input formats of similar programs.

Because of the sequence extension sizeclass implementation, the *MATBLAS* matrix read by the IO module is automatically extended by an ε character (as discussed in section 5.1 on page 11), with the substitution score set to $-10,000$. Although technically only $-\infty$ would be correct, it's obvious that because of the low negative substitution score, the (invalid) substitution will not influence the overall alignment score as the cell score is maximized over 0 in Smith-Waterman.

[22] A branch is any point in the program flow where the program may take two different approaches, with the probability of each branch being data-dependent and/or randomized

[23] While there are approaches to reduce the number of fetches to only two, these algorithms require precomputation and have significantly increased memory consumption – as we show in section 5.5 on page 20, memory optimization is specially important for GPU-based implementations. We will therefore not discuss those methods in detail

Low-complexity regions in sequence inputs are automatically uppercased, because handling those characters in the *OpenCL* code without strict necessity would result in severe performance issues.

5.4.2. Affine gap costs

Although explicitly not required in the task description, we implemented Gotoh's algorithm (see [Got82]) to compute alignment scores with affine gap costs. *CLSW* allows selection of either non-affine or affine gap costs to demonstrate performance differences. We present benchmarks comparing those implementations in section 5.5.4 on page 25.

For comparisons to other tools, we exclusively use the affine implementation, because all comparable tools have been built to achieve high throughput with affine gap costs only – using the same value for the gap-open and gap-extend penalties would therefore massively bias the results as *CLSW* implements optimized methods for this case whereas *SSEARCH* and *SWIPE* will not significantly benefit (in term of computation duration) from non-affine gap costs.

5.4.3. Kernel sizeclass implementation

CLSW uses a single *OpenCL* source code file containing custom macro templates. In order to achieve maximum performance, it's important to aid the compiler in optimizing by using as many constants as possible.

In the case of *CLSW*, while sequences and their lengths need to be transferred at runtime, parameters like the gap-open and gap-extension penalties are inserted by the template system during compilation. Additionally, *CLSW* needs to store a macro for the address computation inside the substitution matrix, as said address depends on the number of characters in the substitution matrix, which might vary[24].

The most important macro is the maximum row buffer size, directly implementing the kernel sizeclass concept as introduced in section 5.2 on page 14: With this parameter set to – for example – 50, query sequences up to 49 characters can be processed. Even if *OpenCL* does not guarantee the compiler allocates this constant-sized area in an efficient memory area, we increase the likelihood of performance increase dramatically.

5.4.4. Memory usage

Fast memory is a scarce resource on any computation platform. Our results (see section 5.5 on page 20, especially the results in section 5.2 on page 14) show that GPU-based computation is even more sensitive to suboptimal memory usage patterns than normal CPU-based software, especially for memory that is accessed frequently (see [Fag10]).

[24] Although the NCBI matrices contain 22 characters, some matrices and sequence sets might for example not contain a `X` character

As aforementioned, *CLSW* uses the inter-alignment parallelism approach. Assuming we only require the alignment score and not the alignment (i.e. the capability of backtracking) itself, the implementation only requires three row buffers (two for the non-affine case), each of which has the size $n + 1$ where n is the number of characters in the query sequence. As shown in section 5.2 on page 14, the buffer size has significant performance impact – therefore users of *CLSW* should take care that the shorter sequence is always used as query sequence (as the buffer size does not depend on the length of the target sequence).

In this report, we will show that it's possible to replace a large buffer (which is frequently accessed) by a target-sequence-size-dependent, less frequently accessed vector buffer and a smaller buffer using the *GIVE-SWAP* algorithm (see section 5.7.5 on page 34). Although *CLSW* does not currently use that algorithm, we expect this construct to exhibit significantly better speed characteristics especially for longer query sequences.

Aspects of memory usage related to limitation of job execution time are discussed in section 5.7.2 on page 31.

5.4.5. Instrumentation

Furthermore, *CLSW* contains detailed instrumentation code that measures the runtime of different parts of the program. Although we used the Unix command `time` to measure the overall runtime of the program in order to avoid biasing the results in comparison with other software, this code prove to be valuable when estimating the overhead of subtasks like kernel compilation, as discussed in section 5.7.1 on page 30.

5.4.6. Validation

In order to ensure *CLSW* computes their results correctly and therefore it is comparable to existing tools, we developed a validation component that is tightly integrated into *CLSW* and can be switched off using command-line flags.

This validator uses the SeqAn library (see [DWRR08]), which is a library of common bioinformatical tools provided by the University of Berlin. While SeqAn provides a multitude of algorithms, Smith-Waterman is one of them. We assume a well-known library like SeqAn has very few bugs and therefore provides correct results which we can use to validate *CLSW*. We checked this assumption by manually comparing results for about 50 alignments with *EMBOSS*.

Assuming the command-line flags enable validation, each alignment score that has been computed using *OpenCL* is subsequently recomputed using SeqAn. These reference scores are then compared to the GPU-computed scores using a preconfigured difference threshold of $10^{-}6$ to account for floating point rounding errors.

Although the SeqAn reference computation takes about three to four orders of magnitude longer than *CLSW* itself (our validation code has neither been parallelized nor otherwise

optimized for performance), we ran the current version of *CLSW* with about $25 \cdot 10^6$ alignments of different query and target sizes randomly selected from the *SIMAP* sequence dataset – none of the validation passes failed.

By using this validation approach we found multiple critical bugs in our source code that would have slipped through the debugging process otherwise, because they were related to binary sequence serialization and only occurred for a subset of those sequences that contained the special aminoacid code *Z*.

CLSW also implements *OpenCL* functions that compute and store the entire Smith-Waterman matrix instead of computing the score for a two-dimensional space of alignments. These functions can be used to debug differences in the resulting output to locate the areas inside the matrix where the scores are different to the reference. We implemented scripts to visualize differences in various forms, including as ASCII graphic or as grayscale image. Latter approach has been used successfully to find aforementioned bug relating to the *Z* aminoacid code.

In previous versions of *CLSW*, we used our own implementations of Smith-Waterman and Gotoh for validation. Even if we had fewer difficulties interconnecting the components using our own software (for example it was difficult to use the heavily templated code from SeqAn to encode and use the same substitution matrix read from the *MATBLAS* file), there were minor differences in our understanding and interpretation of the algorithms and the implementation of other tools like *EMBOSS*, leading to difficult-to-debug issues – for example, some implementations assume that the gap open cost already includes one gap extension costs, while other sources (e.g. [Hay09]) assume the cost of a one-size gap is $gapopen + gapextend$. Our implementation is modeled after the *EMBOSS* interpretation, because we consider *EMBOSS* a source that is more suitable as reference than Haynberg's lecture notes, although the parallel GOBI lab course explicitly had to use the Haynberg interpretation in their Gotoh implementations).

5.5. Benchmarks

5.5.1. Methodology

After successfully validating *CLSW* using the approach described in section 5.4.6 on the preceding page, we benchmarked our application (using the length-aware kernel, see section 5.1 on page 11) by comparing it to *SSEARCH* 3.6 (see [GML+10]) and *SWIPE* 2.0.9 (see [Rog11]) modified by Mathias Walter (see `https://github.com/tolot27/swipe`) to support an output format comparable to the other tools, both of which have been evaluated for the *SIMAP* project.

The *SIMAP* sequence dataset contains more than 70 million sequences – a full computation would take several years, independently of how efficient the implementation is. As no other tool we benchmarked supports sequence extensions, we needed to make sure the input sequences to all tools have equivalent length while avoiding any bias that might be generated by

Therefore, we implemented a filtering tool we call "`SIMAPTool`" that reads a *SIMAP*-formatted

`sequences.gz` and writes out a set of sequences to a *FASTA* file, filtered by length, with a list of

We used this approach to generate *FASTA* files containing at least thousand sequences of different lengths and measured using the UNIX tool `time` how long the execution of the whole program took, including all initialization passes. We did not use the instrumentation implemented in *CLSW* (see section 5.4.5 on page 19), because the other tools we benchmarked didn't support detailed measurements and using only specific timings from *CLSW* would artificially introduce bias to the benchmark.

In contrast to *CLSW*, both *SSEARCH* and *SWIPE* support computing the alignments itself. We ensured that command line options to disable those calculations were active in both tools. By comparing the program executing time with and without these options, we furthermore ensured that the alignments are not only suppressed but not computed at all.

Special care needs to be taken in the IO parts of all given programs. As the program will print $n \cdot m$ scores for input sequence set sizes n and m, the output contributes significantly more to the program execution than reading the input.

While IO is an integral part of the program execution time in practice, our intention was to benchmark the core algorithm implementations. Therefore we redirected the both *stdout* and *stderr* to `/dev/null`. While this approach does not remove IO performance issues directly (because the IO is still a system call), it speeds it up significantly by avoiding to print the data to a pseudoterminal or a file.

Because *SWIPE* supports multi-threading and enabling this option significantly increased its runtime performance, we ran *SWIPE* both single-threaded (*SWIPE*) and multi-threaded (*SWIPE-MT*), where we ran it with any multithreading setting from 1 to 16 threads (inclusive) and selected the best result.

Any benchmark was run four times with the same exact setting successively, discarding the first value (in order to decrease bias introduced by uncached IO) and using the arithmetic average of the remaining three as benchmark value.

Our benchmarks exclusively show runtime values, therefore lower is better for any benchmark graph.

As we had only AMD graphics cards at our disposal and using the GPU nodes kindly provided by the University of Vienna raised severe technical difficulties, we could not get CUDASW++ to work in the limited lab timeframe. Although we have no comparable GPU benchmarks (also see *OCLSW* in section 5.6.1 on page 27, which we also couldn't benchmark for other reason), these difficulties prove the most major issue of CUDA-based approaches: They are not as portable as required.

The test platform – as described in section A on page 40 – is an approximately 3 year old midrange desktop computer. Because in the years since it was manufactured the GPU speed increased proportionally more than the CPU speed and a quad-core[25] is considered not

[25] with Hyperthreading, virtual octocore

ubiquitous for desktop computers even for today's standard, our test method tends to favor CPU-based approaches.

Although we were unable to benchmark on different graphics cards and/or CPU platforms due to the severely limited lab time, we believe our results are – while not fully representative for the potential of GPU platforms – significant and show some effects that can be predicted for the specific platforms and software versions in use.

5.5.2. Kernel sizeclass benchmark

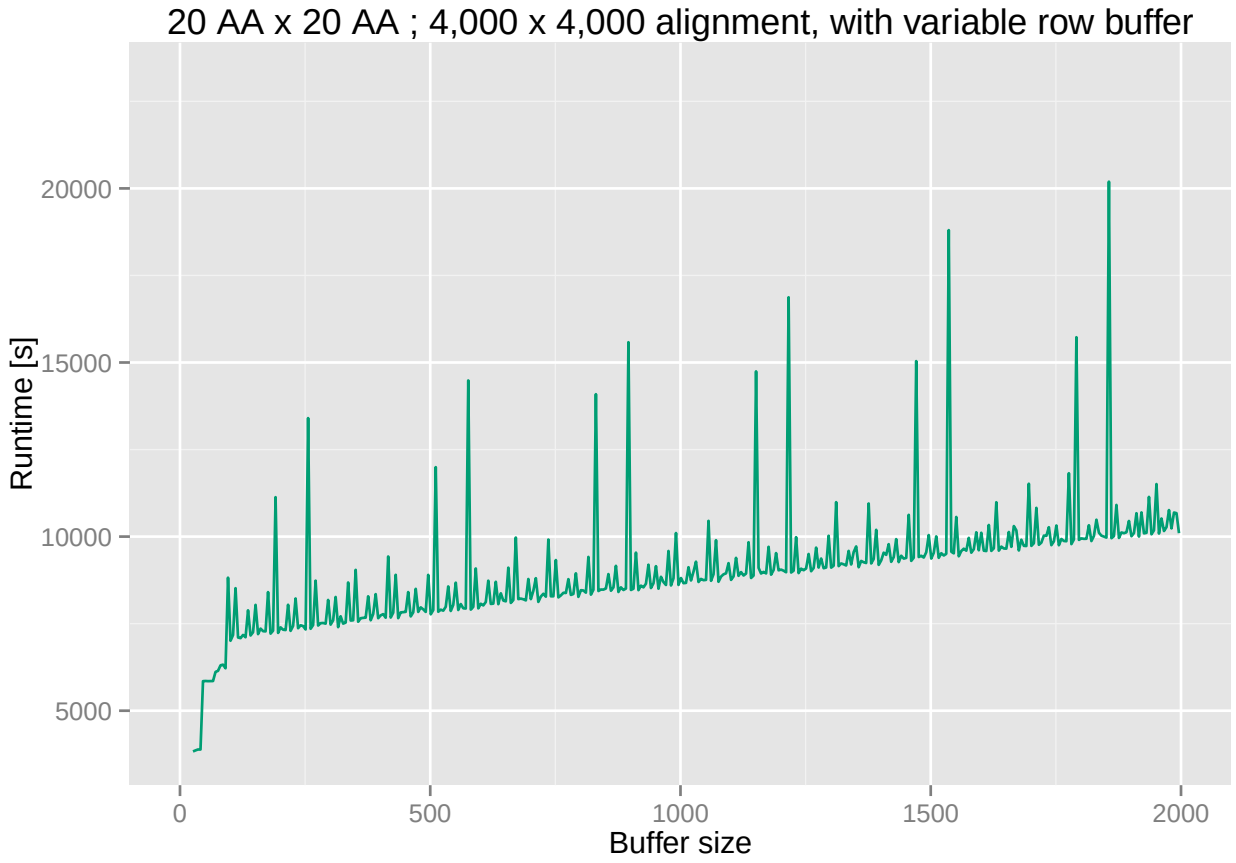

Figure 3: Kernel sizeclasses

In figure 3 we benchmark kernel sizeclasses (and therefore check our assumptions about the performance on the system described in section A on page 40) using the following methodology:

We selected a random set of 4,000 aminoacids with length 2000 from the *SIMAP* dataset. This sequence set was subsequently used as input (as both query and target sequence) for *CLSW*. By leveraging the buffer-size-templating mechanism discussed in 5.4.3 on page 18, we computed the exact same set of $16 \cdot 10^6$ alignments on the same hardware, only varying the buffer size. Most notably, the *CLSW* architecture ensures that the amount of computations (i.e. arithmetical

operations) performed is exactly the same and invariant in respect to the buffer size.s

As it's clearly visible in figure 3 on the preceding page, our assumptions made in section 5.2 on page 14, that larger buffer size are detrimental to the overall performance (and therefore kernel sizeclasses are necessary) clearly applies.

Although we don't know details about the memory architecture of the GPU in use, the massive runtime increase at buffer size = 46 seems to fit the point where to compiler switches to a slower memory area. A similar phenomenon occurs at buffer size = 96, seemingly indicating that there are more than two classes of memories.

The rest of the graph generally shows a linear ascent. We assume this is caused by reduced parallelism: If the compiler can't fit buffers for all available cores into memory, it seems to reduce the parallelism.

Besides the linear ascent, a repeating pattern of peaks is clearly visible, most notably including two large peaks that seem to have approximately the same distance. Although we don't know the exact source of this phenomenon, we've seen similar results in [Fog14, Table 9.1], where choosing a buffer size that is a integral multiple of the cache line size is massively detrimental to performance.

These results can be leveraged by using the *GIVE-SWAP* algorithm introduced in section 5.7.5 on page 34.

5.5.3. Performance comparison benchmarks

Figure 4 on the next page shows the runtime of compute $1 \cdot 10^6$ Smith-Waterman alignment of 1,000 × 1,000 sequences of query and target sequences of 1,000 aminoacids length is shown, comparing the tools listed in section 5.5.1 on page 20.

As it can be clearly seen, *CLSW* is significantly slower that the other tools used. Multithreaded *SWIPE* is the fastest of all compared tools and almost ten times as fast as *CLSW*.

In order to discuss possible causes for the bad performance shown in figure 4 on the next page, we need to compare it with figure 5 on page 25. Said figure uses the same set of length-1,000 aminoacid sequences, but uses length-20 query sequences. The absolute runtime of computing $1 \cdot 10^6$ alignments of length-20-query to length-1,000-target sequences, however, would be less that ten seconds and therefore over-emphasize the IO and initialization overhead in all compared tools. Therefore we used 4,000 query sequences, therefore computing overall $4 \cdot 10^6$ alignment scores.

In contrast to figure 4 on the next page, *CLSW* exhibits better performance than any of the other tools in figure 5 on page 25 – it's about 2x faster than multithreaded *SWIPE*.

Although there is a multitude of potential causes for this effect and we expect it to be highly-device-dependent, our results in section 5.5.2 on the previous page show that memory usage is a critical performance factor in our test environment. As the buffer size solely depends

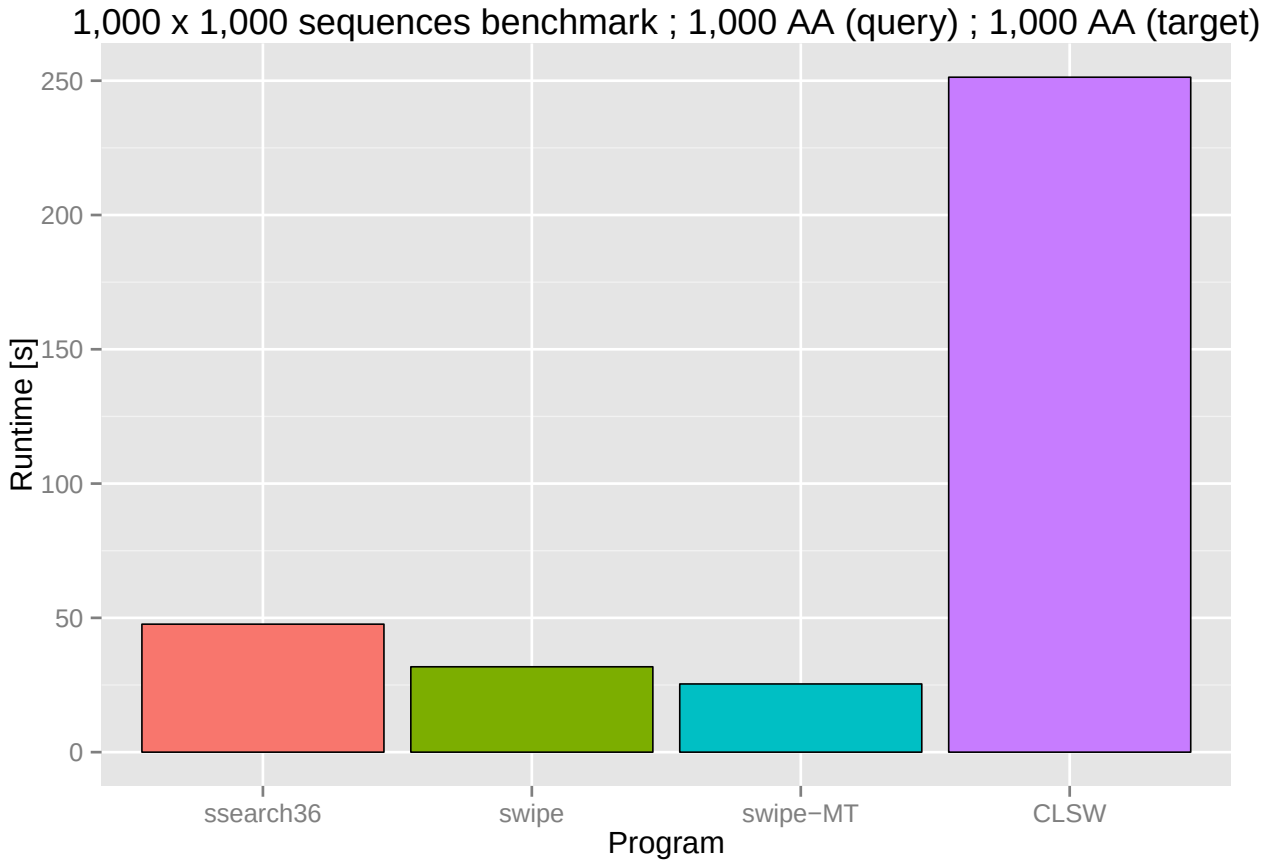

Figure 4: Benchmarks of *SSEARCH36*, *SWIPE*, *SWIPE-MT* (multithreaded *SWIPE*) and *CLSW* performing a 1,000 × 1,000 sequences all-against-all alignment of query and target sequences that are 1,000 aa long.

on the query size (and, most notably, not on the target sequence size), figure 4 requires three buffers[26] of size 1,001 whereas figure 5 on the next page requires three buffers of size 21.

Figure 3 on page 22, as discussed in section 5.5.2 on page 22, clearly shows that buffer sizes of ≥ 50 have a massive on the program runtime.

However, it's also evident that currently using multithreaded *SWIPE* is faster than using *CLSW* for the majority of the *SIMAP* alignments.

As *CLSW* has been developed in less than two weeks, it was impossible to implement all optimization approaches discussed in section 5.7 on page 30. Approaches like SIMD vectorization (see section 5.7.7 on page 39) can however provide a significant speedup. We expect that, by optimizing *CLSW* using these optimization methods, it will reach at least comparable performance to comparable tools, without loosing any of its inherent advantages as discussed in 5.6 on page 27.

Furthermore, with *GIVE-SWAP* in section 5.7.5 on page 34 we introduce an algorithm to

[26] using the affine kernel

Figure 5: Benchmarks of *SSEARCH36*, *SWIPE*, *SWIPE-MT* (multithreaded *SWIPE*) and *CLSW* performing a 1,000 × 4,000 × 1,000 sequences all-against-all alignment of query sequences that are 20 aa long with target sequences that are 1,000 aa long.

leverage the high speed characteristics of *CLSW* for small query sequences for sequences of arbitrary size.

5.5.4. Affine vs. non-affine gap costs

As discussed in section 5.4 on page 17, *CLSW* implements optimized algorithms for both affine and non-affine gap costs Although we did not explore the factors involved in this in detail, we assume that both the increased usage of memory (leading to a higher probability of the buffers being placed into slower memory, as discussed in section 5.7.6 on page 35 and section 5.2 on page 14) and the computational overhead of computing three matrices instead of one leads to the worse performance of the affine kernel.

Because of the results discussed in section 5.5.2 on page 22, we furthermore assume the increased memory usage (even though it only increases by about $\frac{1}{3}$) accounts for the majority of the runtime performance detriments.

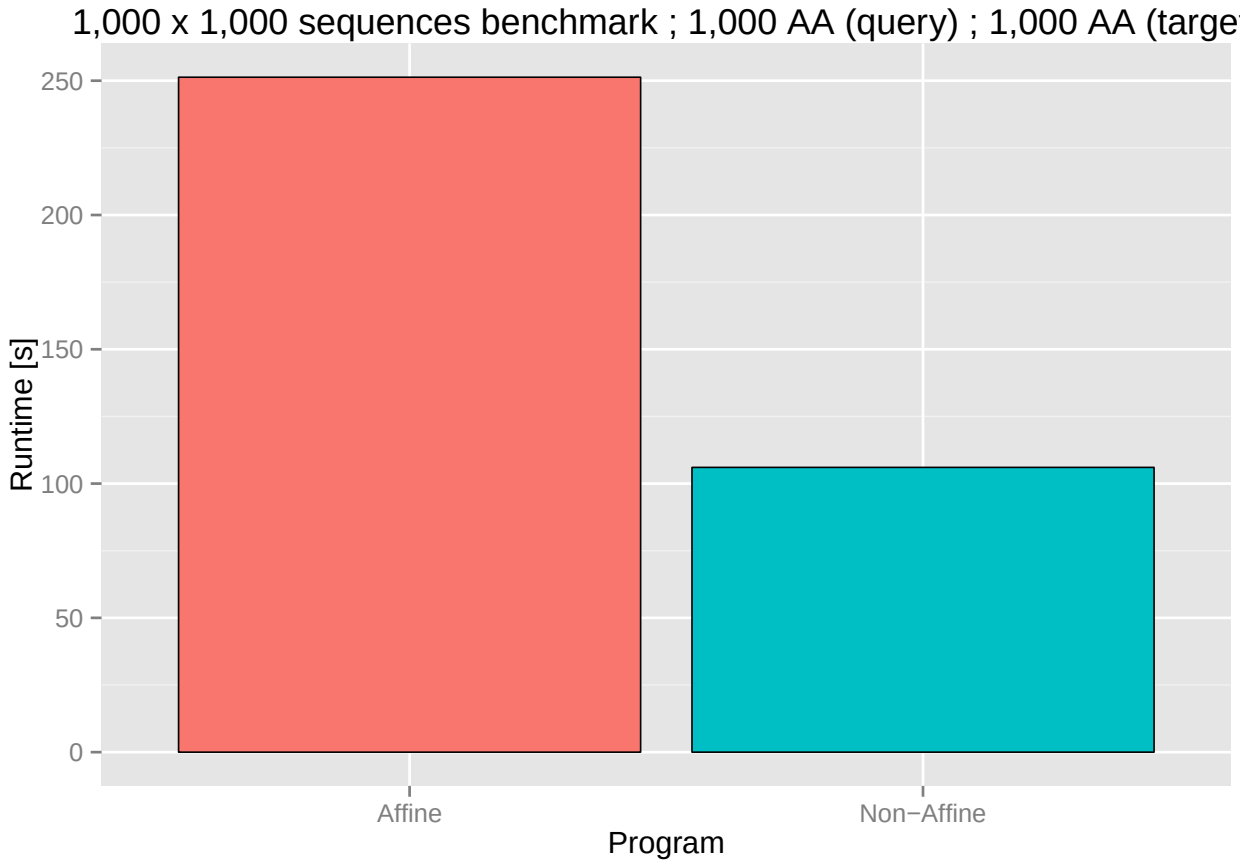

Figure 6: Benchmark of affine vs. non-affine gap cost implementation

In figure 6 a comparison of the runtime of both implementations is shown, being used with the same parameters and sequence sets as figure 4 on page 24 is shown. All other tests we conducted show nearly identical results, with smaller overall runtimes showing significantly less absolute performance advantage of the non-affine implementation. By using the instrumentation discussed in section 5.4.5 on page 19, we were able to attribute about 80% of the difference between long-duration and short-duration tasks to kernel initialization – we assume the remaining 20% are caused by initialization and cleanup overhead that can't be measured individually.

Although these results might significantly depend on various parameters, our results show a speedup of 2 to 3 when using the non-affine approach. For practical use, however, we think the higher resemblance of affine gap costs with the biological reality significantly outweighs the performance increase. Should any application require only non-affine gap costs for any reason, these results show that it is possible to gain significant performance advantage by removing any affine-specific code.

5.6. Advantages over existing solutions

5.6.1. Comparison to *OCLSW*

Another *OpenCL* Smith-Waterman implementation, namely *OCLSW*, is introduced in [RMG+10]. The approach of the authors generally follows the intra-alignment parallelism approach as outlined in 5 on page 9, but implement advanced mechanisms to precompute parts of the matrix in parallel for si

Although a performance comparison would be interesting, we found a multitude of complex bugs in the implementation that led to the conclusion that the authors did not develop their application to a generally usable state.

One example for a critical bug in the implementation is the code line

```
char * const g_stAminoAcids = "ATCG"; //"ARNDCEQGHILKMFPSTWYVBZX";
```

Obviously, the authors only tested the publicly released version with DNA sequences – trying to use aminoacid sequences as input for *OCLSW* led to an immediate program crash with the message that *M* is not an aminoacid

After resolving three critical bugs without success, we abandoned all attempts to get *OCLSW* to work properly.

As our primary goal was to develop a GPU-based Smith-Waterman application for *SIMAP*, we can't see *OCLSW* as viable alternative to other applications: It is severely bugged to a point where it's utterly useless for the *SIMAP* usecase and there is no indication the authors continue developing and maintaining it.

We suspect the authors only developed *OCLSW* as a proof of concept application to prove their theses for their paper. Even if *CLSW* is – in its current state – also a proof-of-concept-only implementation and one might raise similar criticism about *CLSW* for other usecases, we deem *CLSW* as significantly more stable and maintainable (partly because of its simpler inter-alignment parallelism approach) at least for the *SIMAP* usecase.

5.6.2. Platform-agnosticism

In contrast to other GPU-based solutions like CUDASW++ (see [LMS09, LSM10, HCO+11]), *CLSW* not only runs on NVidia-based platforms (CUDA is proprietary NVidia technology that does not run on hardware made by other vendors), but also on AMD GPUs and other highly parallel accelerators like Intel Xeon Phi (see e.g. [CSKM12]).

Assuming appropriate ICDs exist, *CLSW* also runs any type of CPU and even FPGA devices (see section 3.1 on page 7 for a detailed discussion). Therefore it is truly platform-agnostic, without requiring any change of the source code, although specific platform might require specific optimizations and the parallel programming model might also be suited better for platforms with higher hardware parallelism than for classical general-purpose platforms like CPUs.

Besides being a CPU-only solution, *SWIPE* is specific to CPUs supporting the SSE2 (and optionally SSSE3) instruction set extensions. While all modern x86_64 CPUs support those instruction set extensions, *SWIPE* can neither leverage the full power of wider SIMD instructions (e.g. AVX) in recent CPUs (see section 5.7.7 on page 39 for a detailed description) nor run on any non-x86_64 CPU, even if alternative architectures like ARM are getting more and more relevant – *SIMAP* even recently released a *BOINC* client that runs on Android smartphones, the vast majority of which are ARM-based.

5.6.3. Compact codebase

While comparable solutions like *SWIPE* have a large (almost 10,000 SLOC[27] codebase *CLSW* achieves comparable or better results (see section 5.5 on page 20) with less that 1,000 overall lines (including validation code), while the core *OpenCL* Smith-Waterman implementation consists of less that 50 SLOC.

SWIPE requires complex SSE-specific inline assembler code which is complex and hardly documented, therefore increasing the probability of rarely occurring bugs (for example, the authors of *SIMAP* recently discovered a bug relating to alignment scores $> 2^{32} - 1$) and dramatically decreasing the maintainability of the software. It's almost impossible to port *SWIPE* to another platform, as even small architectural changes require rewrites of large amount of complex core code.

5.6.4. Floating-point computation

Existing tools for both CPU and CPU platforms (most notably *SWIPE* and *CUDASW++*) exclusively use integral computations for both the substitution and the Smith-Waterman matrix itself. While for many cases this method is sufficient, we believe that (although it has proven difficult to find hard evidence for this theory) this has mainly historic reasons:

When substitution matrices and their underlying statistical theorems were discovered, floating point computation was still a difficult task for computers. While the CPU itself can run fast integer computations, floating point computations could only be handled by specialized FPU[28] ICs like the x87 family. Later, these ICs were integrated into the main processor, but floating point computations were still significantly slower than integer operations.

Based on this phenomenon, computational biologists started to truncate[29] the logarithms in the substitution matrices to be able to compute using fast integer operations exclusively.

While there is a scaling factor λ^{-1} applied (see [YA05]) before the truncation and this factor could lead to the same results as using true floating-point computation if chosen appropriately

[27] Source lines of code, does not include empty lines and lines only containing comments or brackets
[28] Floating point unit
[29] Rounding is also an option, but expresses the same issues

large[30], during our research regarding this topic we have not found a single substitution matrix that uses large numbers – most notably, the *MATBLAS* matrices published by the NCBI rarely exceed single-digit numbers.

We believe that further research is required to determine how large the effect is

In contrast to the aforementioned other implementations, *CLSW* uses single-precision (32-bit) floating point numbers exclusively. Not only can floating point numbers be excepted to be as fast as integers on modern platforms, there is even the possibility of GPUs being faster in the floating point domain because the majority of the 3D computation requires floating-point accuracy.

By using floating points, the results when using integral substitution matrices are equivalent to using existing methods – however, if using more accurate input, one can leverage that characteristic without changing *CLSW* itself to improve the output score accuracy.

Composition-based score adjustment While one application of floating-point-capable algorithms is to use more accurate substitution matrices as parameter, a significant amount of research has been performed on *Composition-based score adjustment* (see [AWG$^+$05, YA05]).

The method is based on the observation that using one substitution matrix for all sequences assumes a random protein model, "in which the aminoacids occur independently with background probabilities $\overrightarrow{p}$" ([YA05, p. 1]). Composition based score adjustment resolves this problem by modifying $\overrightarrow{p}$ according to the protein sequence. As $\overrightarrow{p}$, i.e. the aminoacid propensity which significantly depends on the propensity of secondary structures like β-sheets or α-helices).

Although a detailed discussion would break the boundaries of this report and we therefore refer to [AWG$^+$05, YA05] for the construction algorithm, we observe that in [AWG$^+$05, p. 6] Stephen Altschul (which we consider to be of significant importance for the mathematics underlying substitution matrices – most notably Karlin-Altschul-statistics) et al. use a floating-point-based BLOSUM62 as base for the adjustment. Although [AWG$^+$05] does not clearly indicate why they're not using integral truncated matrices (notably, Altschul works for the NCBI who publishes said truncated matrices), we assume that him using a floating-point matrix is supportive of our theory in section 5.6.4 on the previous page where we assume that floating-point matrices yield more accurate results.

For composition-based adjusted substitution matrices however, we assume that the changes to the original matrix will be so small that truncation will in most cases truncate away the information gained by composition-based score adjustment. Therefore we assume that floating-point accuracy is an absolute requirement for composition-based score adjustment.

The authors of *SIMAP* intend to integrate composition based score adjustment into new versions of their application – for the Smith-Waterman algorithm (in contrast to using *FASTA*

[30]The gap penalties require adjustment accordingly

or *BLAST*) doing that is only possible using *CLSW*, because other applications like *SWIPE* or *CUDASW++* do not support the required floating-point accuracy for the adjustment

PSSM-based scoring Although we did not explore this in detail, we believe the even more general approach of *sequence-specific and position-specific substitution matrices* outlined in [AH11] could result in even more accurate scores than composition-based-adjusted substitution matrices and it would be possible to integrate this approach into *CLSW*.

5.6.5. Performance advantages for short queries

As discussed in section 5.5 on page 20, *CLSW* has a significant advantage in performance for short query sequences. By using the *GIVE-SWAP* algorithm introduced in section 5.7.5 on page 34, we expect that this advantage can be expanded to arbitrarily-long query sequences.

However, there are usecases where very short aminoacid sequences need to be aligned to larger sequences: While there has been significant progress in Next-Generation genomic sequencing (*CLSW* is generally capable of DNA/RNA alignments, however some optimization might be necessary to achieve competitive speeds), progress has also been made in shotgun protein sequencing allowing high-throughput aminoacid sequencing (see [BCP07, MY02]). The high performance of *CLSW* even for larger target sequences – as long as short fragments are used as queries – seems to perfectly fit the task of reassembly, possibly allowing to use optimal alignment scores instead of heuristic approaches, increasing the overall accuracy.

5.7. Methods for further optimization

In this section we will introduce several methods that have not been implemented in *CLSW* so far, but could have a high chance of increasing its overall performance. because we can not possibly include all optimization methods, we focused on algorithms that can be implemented with realistic effort without negatively affecting any of the advantages discussed in section 5.6 on page 27

5.7.1. Reduce kernel compilation overhead

In the current version of *CLSW*, any time the executable is run on a sequence dataset, the *OpenCL* sourcecode is compiled by the *OpenCL* ICD.

This ensures maximum compatibility and reduces the complexity of the *CLSW* source code – however the optimization performed by the *OpenCL* compiler introduces a significant amount of overhead. While for the tested GPUs the kernel compilation reproducibly took between 1-2 seconds (depending on the driver version and GPU hardware), FPGA-based devices are expected to be significantly slower. In order to explain this phenomenon, we first need to review FPGA-based computational hardware design.

There are two general ways to compile *OpenCL* kernels for reconfigurable FPGAs:

1. Use a highly-parallel softcore on the FPGA that is independent of the *OpenCL* kernel executed, and run *OpenCL* as software on it
2. Synthesize[31] a separate FPGA bitstream[32] for each[33] kernel

While option (1) also works for one-time-programmable FPGA, the FPGA fabric is used to mimic a GPU-like architecture – although it may be specialized for a class of computational tasks, most of the FPGA fabric space is wasted for logic required for instruction interpretation and synchronization.

Option (2) uses the full potential of the FPGA fabric capable of logic-level optimizations. However, the process of building a FPGA bitstream – mainly consisting of *Synthesis* and *Place&Route* – is NP-complete[34] (see [GJS74]). In reality – even though commercially-available tools have been optimized significantly – it takes several minutes to compile programs equivalent to less than 10 lines of CPU code. Compiling larger programs – like those required for highly-parallel *OpenCL* computation – is expected to take multiple hours to days.

OpenCL provides a facility to retrieve the compiled binary program from the ICD. A possible optimization of *CLSW* would involve retrieving the binary, saving said binary and re-using it if an appropriate binary is available. However, the developer would have to take special care that no incompatible kernels[35] are used in order to avoid errors.

This optimization would therefore reduce overhead for CPU and GPU computations, while it is absolutely essential for FPGA-based devices, assuming option (2) is chosen by the vendor. In the context of the benchmarks outlined in section 5.5 on page 20, the speedup for short query sequences could be increased further from $3.0\times$ (*CLSW* $\leftrightarrow$ single-threaded *SWIPE*) to $3.4\times$[36]

5.7.2. Compute multiple sequence sets in one program run

The current *CLSW* implementation reads two sequence sets S_a and S_b and computes alignment scores for any sequence tuple (s_a, s_b) where $s_a \in S_a \wedge s_b \in S_b$. For efficiency reasons, *CLSW* allocates a chunk of memory for both S_a and S_b. Because these sequences are extended to a sizeclass with a size equivalent to the maximum sequence length of the respective sequence set,

[31] Synthetization is the process of turning an abstract algorithm implementation into a boolean-logic-level construct that can be programmed into an FPGA

[32] A *bitstream* is the logic-level configuration of the FPGA fabric and connected peripherals and therefore roughly equivalent to a software on GPUs or CPUs

[33] While it seems logically possible to compile a whole set of kernels into a bitstream, analysis of this possibility is outside the scope of this report

[34] While synthesis is only considered NP-hard, because of the NP-completeness of Place&Route the whole process needs to be considered NP-complete

[35] for example, if the user changed his graphics card or driver version

[36] Assuming a normally distributed kernel compilation duration with $\mu = 1.7$

therefore yielding a $\mathcal{O}(n+m)$ space complexity, with n and m being the sizeclass lengths of S_a and S_b.

However, as the *OpenCL* kernel needs to store a 32-bit floating point score for any sequence pair, the space complexity of the output needs to be added to the space occupied by the sequence arrays: $\mathcal{O}(|S_a| \cdot |S_b|)$.

Especially for desktop computers it's not possible to use all of the memory the graphics card has available[37], because both the management part of *OpenCL* and – in case of desktop systems – the whole graphics system, including display buffers, require large amounts of DRAM themselves.

In our tests with desktop systems, we discovered that using more than 50% of the graphics card DRAM (with 512/1024 MB DRAM on AMD cards) increased the probability of system crashes and freezes significantly during computation, while the system seemed perfectly stable with less DRAM being used.

Therefore, it's necessary to limit $|S_a| \cdot |S_b|$ to be able to reserve no more than 40% of the available graphics card DRAM[38]. This consideration is specially important in a distributed computing environment where different generations of high-end and low-end graphics cards need to be supported.

Assuming a minimum of 256 MB for supported graphics cards, we need to ensure that $|S|^2 \leq 0.4 \cdot 256 \cdot 10^6 \Rightarrow |S| \leq \sqrt{0.4 \cdot 256 \cdot 10^6} \cong 10119$ where we assume without loss of generality that $|S| := |S_a| = |S_b|$.

Another important consideration is that *BOINC* users might expect their desktops not to freeze during computation. Although the exact freezing mechanics seems to depend on the driver and graphics card in use, based on previous experiences we know that freezing can only be eliminated when not using one large minute-duration job, but thousands of smaller jobs. Although an analysis of the exact cause of this phenomenon is outside the scope of this report, we attribute it to the missing capability of GPU drivers (or even GPUs itself) to preempt running jobs, together with the incapability of running several jobs in parallel (although the workspace inside one task is computed in parallel). Therefore, the scheduler needs small tasks in order to interleave both display rendering and the GPGPU task.

While this is not relevant for servers and unused desktop devices, that is not an assumption we can generally make when building a system that is capable of being used as a distributed computing GPU application. Therefore we conclude that for server-only installations the DRAM is the limiting factor, while for applications where freeze prevention is required, a job duration limit several orders of magnitude lower needs to be imposed upon the *CLSW* job splitter.

Even though both the memory limitation and the freeze prevention methodology seem to

[37] This is no different in CPU-only platforms: A server with e.g. 32 GB of DRAM can't run an application using 32 GB of RAM without using techniques like swapping

[38] Determining more accurate numbers require testing on a multitude of platforms, which we don't had to our disposal during this project

indicate smaller jobs are always better, one also needs to take into account the overhead introduced by starting more than one job. Not only will multiple jobs result into non-continuous computation (which is desirable to some extent in order to prevent freezes), but also in increased overhead to setup the jobs on the device.

In the current state of the application, it's impossible to compute large sets of alignment without calling *CLSW* multiple times, because it would fail trying to allocate a too large amount of DRAM. Therefore, it's not only necessary to implement a job splitter in the application itself for performance and compatibility reasons[39], but also to be able to run it standalone with longer-duration *BOINC* jobs[40].

5.7.3. DMA-based overhead reduction

Especially in the context of small jobs required for freeze prevention and memory limits, it is important to reduce the overhead induced by data transfers from and to the device. While transferring the sequence arrays to the device can introduce a non-negligible amount of latency in the overall process, we have shown that the resulting score array asymptotically occupies quadratic space (see 5.7.2 on page 31) while the sequence arrays only occupy linear space in respect to the number of sequences in both sequence sets.

As shown in figure 7 on the next page, we can use a technology known as DMA to transfer data to and from the device while a computation is in progress. Most modern computational hardware, including CPUs and GPUs include a DMA controller that enables applications[41]. DMA requires device support for mapping device memory into the CPU's virtual address space, however there are no modern graphics cards that do not support memory mapping because of increasing performance demands even for simple desktop applications and other compatibility reasons.

While DMA might still stall the computation for a small period of time because of arbiting[42] conflicts, it can still be expected to be faster than interleaved transfer/computation scheduling as outlined in figure 7 on the following page.

Currently, *CLSW* uses DMA-based host-to-device-transfers (i.e. sequence array transfers), but (mainly because of lacking support for the optimization outlined in section 5.7.2 on page 31) it doesn't support data transfer concurrent to the computation.

[39] For example, *CLSW* could adaptively adjust the job size based on several BOINC-supplied parameters

[40] While shorter *BOINC* jobs loose fewer computation results in case of failure, it's preferable to use at least 10-minute jobs in order to reduce the massive server load incurred by requesting, transmitting and checking second-duration jobs

[41] On most operating systems, only the kernel uses DMA directly, e.g. to transfer Ethernet packets from the RAM to an Ethernet card's internal buffer

[42] Arbiter: Logic that controls and serializes memory accesses

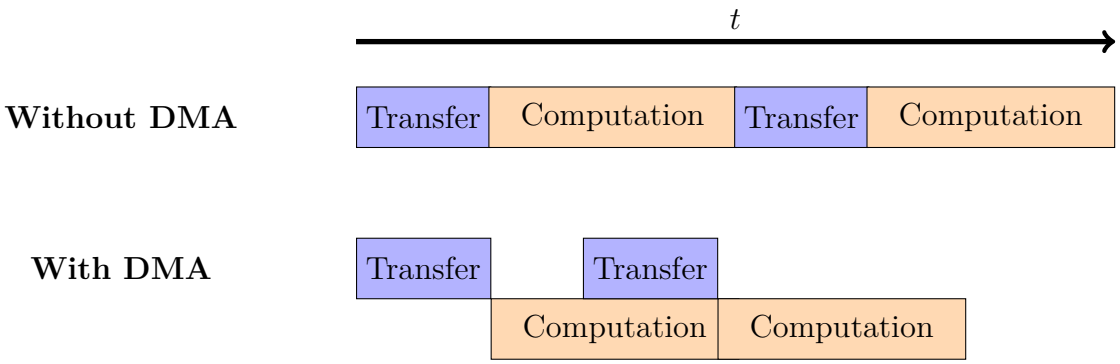

Figure 7: DMA-based transfer concept

5.7.4. Sequence array reuse

Should host-to-device-transfers of sequence arrays proof to be a significant bottleneck, it's possible to re-use either the query sequence or the target array by intelligently scheduling the job order so that e.g. one set of query sequence can be aligned with multiple target sequence arrays without the need to re-upload the query sequence array.

While this might be relevant in conjunction with small job sizes outlined in 5.7.2 on page 31 because of the requirement for internal splitting into sub-jobs, we expect that this method won't provide any noticeable performance increase because

5.7.5. The *GIVE-SWAP* subdivision algorithm

In section 5.5 on page 20 we observed that *CLSW* is faster for very short query sequences.

In this section, we will present the *GIVE-SWAP*[43] algorithm that allows us to leverage this phenomenon to increase processing speed for larger query sequences. *GIVE-SWAP* is an extension to Smith-Waterman that allows us to partially calculate a slice of the matrix, using a numeric vector as intermediate state representation. While it could also be applied to target sequences, Smith-Waterman is inherently invariant to sequence swapping – therefore creating an isomorphism between both calculations.

For our example, we'll use a 5×5 Smith-Waterman matrix. If we would compute the matrix using the unmodified Smith-Waterman algorithm using inter-task parallelism (therefore computing one row before the next, storing only the current and the previous row), we would need to allocate two row buffers that can store at least 5 bytes[44].

By using *GIVE-SWAP*, we can reduce that to 2 bytes[45]. In order to comprehend the underlying

[43] Generalized initialization vector extensions to Smith-Waterman alignment problems

[44] For performance reasons, 6 bytes need to be stored in the buffer, including the leading zero to allow branch-free direct array indexing

[45] Any subdivision is possible, down to a row buffer size of 2, independent of the query sequence size

concept[46], we first introduce the column vector $\overrightarrow{IV}$[47] where $\overrightarrow{IV} \in \mathbb{R}^n$ where n is the number of characters in the target sequence.

For a given (sub)matrix that is calculated atomically, we define $\overrightarrow{IV}$ as the column that is directly left of the first column to be computed. This column is not assigned to any query sequence character[48]. Therefore, the classical Smith-Waterman algorithm always uses $\overrightarrow{IV} := 0$, while unmodified Needleman-Wunsch without affine gap costs uses $\overrightarrow{IV} := [0 \cdot G, 1 \cdot G, 2 \cdot G, 3 \cdot G, ...]$ where G is the configured gap penalty. The initialization vector is – together with the first row which can be computed trivially assuming one knows the absolute Cartesian coordinates in the matrix[49] – sufficient for computing any remaining cells because each cell depends only on the cell left, above and above-left of the cell to be computed. The cells left and above-left of said cell to be computed are either normal cells or elements of either $\overrightarrow{IV}$ or the first row vector.

Furthermore, we observe that by extending the query sequence, the matrix is only extended horizontally by adding new columns. This new sub-matrix can however be calculated by knowing only the values of the last column vector of the already calculated sub-matrix. Said column vector is however the column vector directly left to the first column to be computed in the new sub-matrix – therefore it is $\overrightarrow{IV}$ in respect to said new sub-matrix.

Based on these observations, we conclude that we can extend the query sequence of any Smith-Waterman-Problem while knowing only the last computed column vector.

Given this knowledge, we can easily use it to subdivide an existing query sequence by considering the first n characters of said query sequence and extending appropriately to compute the rest of the matrix. Between extensions, only $\overrightarrow{IV}$ for the next computation, being equivalent to the last computed column vector, needs to be stored in-memory. As the memory area storing $\overrightarrow{IV}$ is only accessed $\mathcal{O}(n)$ times, there is less performance impact if this memory is slow and we expect an implementation using *GIVE-SWAP* to be faster regardless of the target sequence size.

In figure 8 on the following page a basic example is shown - the subdivision occurs at the column boundary marked in red, i.e. after two query sequence characters. The column marked in blue represents $\overrightarrow{IV}$ for the sub-matrix after the subdivision.

5.7.6. Memory access optimization (*TESSIN*)

As we observed in section 5.5 on page 20, memory speed is a critical factor in optimizing *CLSW*.

A discussion of GPU-memory access optimization in relation to alignment algorithms can

[46] Without loss of generality, we explain *GIVE-SWAP* only in the non-affine-gap cost domain By using a vector of tuples instead of a scalar vector for $\overrightarrow{IV}$, this can be generalized

[47] We use the notation $\overrightarrow{IV}$ to express the vector character of IV, without expressing any further meaning about IV, as in [YA05]

[48] As aforementioned, we assume without loss of generality that query sequence characters are always assigned to columns while target sequence characters are always assigned to rows

[49] In the special case of Smith-Waterman it is set to zero, therefore knowledge of the Cartesian coordinates is not required

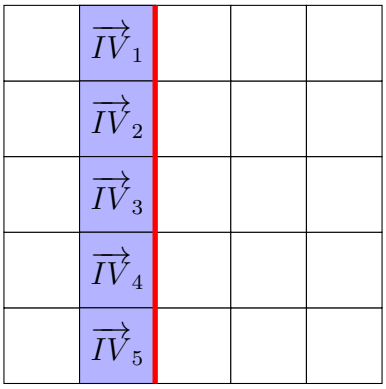

Figure 8: GIVE-SWAP subdivision example

be found in [ZMA+12]. While the assumptions they make for their particular algorithm only partially apply to the concepts in use by *CLSW*, their basic idea can be re-used to create a more efficient memory layout.

In this section, we will present *TESSIN*[50], an algorithm that allows us to store a set of sequences with equivalent length[51]

Although a detailed discussion would break the boundaries of this report, we will present a generalization of in [ZMA+12] that in a special case theoretically provides better performance than both the linear sequence array storage currently in use by *CLSW*.

In order to explain our method, we require some definitions: Let S be the sequence set we intend to store efficiently. Let s_n the nth sequence in said sequence set. Furthermore, let $s_{i,j}$ be the jth character in the ith sequence in S, with $s_{i,n-m}$ denoting the nth to mth character inside s_i. In our examples, we will use 1-based indices exclusively.

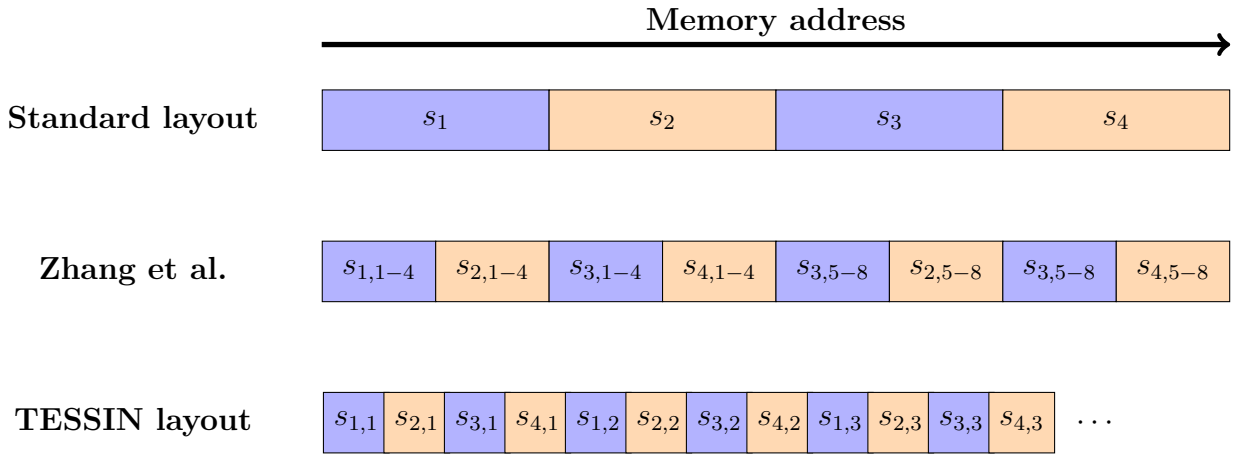

Figure 9: Memory layouts in comparison with TESSIN

In the standard linear memory layout, any sequence is stored in an non-fragmented manner,

[50] Transposed Equi-length Storage of Sequence INformation
[51] Shorter sequences can be extended up to the desired length using the method introduced in section 5.1 on page 11

whereas both Zhang et al. and *TESSIN* store sequences in multiple fragments.

In order to explain why sequence fragmentation can – depending on the memory access pattern and device in use – improve computational speed, we need to have a look into the internals of caching on modern devices:

If a task requires a single byte of memory from the system's RAM – assuming said byte is not yet in the cache – the memory controller will load at least[52] an entire cache line from memory. This pattern not only limits cache management overhead but also ensures the high hardware parallelism of modern DRAM ICs is fully leveraged.

Although the cache line size depends on the device in use and varies by several orders of magnitudes, one can generally assume that at least 32 consecutive bytes are loaded for each memory fetch on both CPUs and GPUs. While the *OpenCL* optimizer is able to improve many performance-relevant parts of the program, memory access optimization can usually not be optimized automatically, because the compiler often does not have sufficient information on the exact memory access pattern and most ICDs can't re-order memory on the fly while uploading to the device[53]

Because of the inter-task parallelism used in *CLSW* and its deterministic cell computation order, it can easily be proven that while the target sequence character only changes after finishing to compute an entire row, the query sequence character required for the substitution computation changes for every cell.

Therefore, it's evident that optimizing cache hit rates for query sequence character access will improve overall program performance.

[ZMA+12] subdivide sequences into 4-character-long fragments and store the nth fragment for any sequence in S consecutively. Therefore, a cache line might contain the nth to $n+3$th character of multiple sequences. The layout stores multiple lines of fragments where a line is a single fragment for each sequence in S stored consecutively (see [ZMA+12, Fig. 2]).

If the memory access pattern matches this assumption, the number of cache hits are maximized. Zhang et al's method seems to be optimized for a SIMD-like approach where 4 cells are computed at a time. Obviously, storing fewer characters for each sequence in a cache line means that character for proportionally more sequences can be stored in said cache line.

In the case of *CLSW*, only one character is required at a time. Though the exact task executing order is neither deterministic nor controllable, we assume that caching the nth character of consecutive sequences will improve the overall executing speed, even if not all fetches will yield cache hits. Especially for long sequences[55], there will be virtually no cache hits, especially if the cache is not large enough to store an entire line for any sequence.

However, the method introduced in [ZMA+12] can be generalized to load not exactly 4 but m

[52] On CPUs with virtual memory support, the kernel mostly loads entire pages, but in faster caches like L1 there is limited space so the page is only partially loaded into these caches

[53] With DMA, this task is even harder as COTS[54] DMA controllers can't do any processing at all

[55] with a length greater than the cache line size

characters for each sequence at a time. Using powers of 2 for m improves the performance by avoiding division-like[56].

For the general case, calculating the memory offset (relative to the start of the sequence memory region) yields (with $|S|$ denoting the number of sequences in S):

$$s_{i,j} = |S| \cdot m \cdot \lfloor \frac{j}{m} \rfloor + m \cdot i + (j \ mod \ m)$$

In this computation the computationally expensive instructions are the division in $\lfloor \frac{j}{m} \rfloor$ (see [Fog14]), the modulus in $(j \ mod \ m)$. By using powers of 2, the division and Gauß bracket application can be reduced to a bit shift while the modulus application can be reduced to XOR by using $\oplus(n+1)$.

While standard linear sequence storage is a space case of this generalized storage layout (with m being equal to the sequence length), *TESSIN* uses $m := 1$. This reduces the aforementioned formula to $s_{i,j} = |S| \cdot j + i$. Even when used with a SIMD-enabled application as shown in section 5.7.7 on the next page, we expect the easier address calculation consisting of only a multiplication with a constant and an offset-addition is overall faster[57] because complex address computations for each cells are avoided.

TESSIN (see figure 9 on page 36 for a visual comparison with [ZMA+12] and the linear layout) can also be seen as a simple transposed storage layout: A single sequence s_i can be seen not only as a string, but also as a vector[58] of characters:

$$s_i = \begin{pmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,n} \end{pmatrix}$$

However, we can also see this as matrix (with one of the dimensions being 1) and transpose it:

$$(s_i)^\top = \begin{pmatrix} s_{i,1} \\ s_{i,2} \\ \cdots \\ s_{i,n} \end{pmatrix}$$

If we consider $(s_i)^\top$ as a column vector of a matrix and successively add any other sequence column vector from S, we'll get a sequence set matrix, for example with $|S| = 3$ and a sequence length of 3:

$$S^\top = \begin{pmatrix} s_{1,1} & s_{2,1} & s_{3,1} \\ s_{1,2} & s_{2,2} & s_{3,2} \\ s_{1,3} & s_{2,3} & s_{3,3} \end{pmatrix}$$

[56] Division-like: Modulus or divisions, many architectures generally compute both at once

[57] This operation is supported as indirect addressing by most hardware

[58] Although vectors are often noted as columns, i.e. vertically, we note s_i as horizontal/row vector because of its horizontal notation as a string

Storing $S^\top$ in row-major order directly yields the *TESSIN* representation.

Although *CLSW* currently does not currently implement *TESSIN*, we expect it to improve performance by a factor of between two to ten, being highly dependent on device and ICD characteristics like cache size.

5.7.7. SIMD-Vectorization

While *CLSW* gains most of its performance from the high inter-task parallelism, alternatives like *SWIPE* use SIMD[59] instructions to speedup their computation.

SIMD hardware – including any modern CPU and GPU – allows the programmer to execute the same instruction not on a single scalar but on a vector of scalars with a defined size.

For example, the SSE[60] of the x86 platform allows computations (mainly simple arithmetic operations like addition, multiplication or division) to four 32-bit floating point numbers instead of one, while not taking more time than a single computation[61]. Newer instruction set extensions like AVX[62] in x86_64 processors offer an even-wider parallelism of for example eight 32-bit-floating point numbers, although neither *SWIPE* nor comparable tools support them.

GPUs usually compute large-scale 3D environments and apply visual effects to them – in order to do this efficiently, they not only require a large number of parallel computation units but also SIMD parallelism in order to compute 3D coordinates (which are represented by 4×4 matrices) efficiently.

OpenCL supports using 4-dimensional vectors of floats instead of single floats currently in use by *CLSW*. While it would be possible to combine GPU-core inter-task parallelism with SIMD intra-task parallelism, it seems to be significantly easier and less overhead-prone to simply compute 4 Smith-Waterman alignment scores in parallel on a single GPU core.

In order to efficiently do this, we require not only basic arithmetics on vectors but also a vector *max* operation, being defined as:

$$max(\begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{pmatrix}, \begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{pmatrix}) := \begin{pmatrix} max(a_1, a_2) \\ max(b_1, b_2) \\ max(c_1, c_2) \\ max(d_1, d_2) \end{pmatrix}$$

In *OpenCL*, support for this operation is already present, however as it is not generally important for efficient 3D calculation, some *OpenCL* devices might calculated the resulting vector serially.

[59] Single instruction – multiple data

[60] Streaming SIMD Extension

[61] It should be noted that this performance gain comes with a significant number of problems, including overhead for transfer to/from the SSE registers and using 32 or 64 bit floating-point precising instead of 80-bit precising as in use by the x87 FPU integrated in any x86 processor

[62] Advanced Vector Extensions

While implementation of this feature in *CLSW* was not possible, we consider SIMD optimization a method that easily allows the programmer to achieve three-to-four-fold speedup.

6. Conclusion & Outlook

In this report we introduced *CLSW*, a fast GPU-based Smith-Waterman score-only-alignment calculator. While generally applicable for any protein alignment problem, it was designed specifically as a proof-of-concept application for *SIMAP*.

Even if we had only two weeks to develop a fully functional, validated and optimized implementation and all related concepts, our results show that in some special cases *CLSW* is significantly faster, generally more portable and overall simpler than competing solutions like *SWIPE*.

By using *OpenCL* as a backend computation framework, *CLSW* is not only GPU-vendor-agnostic, but achieves true platform agnosticism by running on an ever-increasing multitude of computational hardware, without requiring any modification.

Three new algorithms were introduced that can be used to optimize *CLSW* even further, especially in those cases where competing solutions are currently faster:

- *QUANTMASS*, an adaptive heuristic metric to efficiently compute kernel- and sizeclass boundaries
- *GIVE-SWAP*, an extension to the score-only Smith-Waterman algorithm that allows imposing arbitrary limits on computation buffer sizes while still only using linear memory
- *TESSIN*, a memory layout scheme for generalized sequence storage that takes advantage of cache-line characteristics to improve cache hit rates for inter-task Smith-Waterman parallelization

By implementing these optimizations, we believe that *CLSW* could prove advantageous not only to *SIMAP*, but to many projects that today can only use heuristic approaches to alignment scoring. *CLSW* could be used to improve the data quality of those databases, using only off-the-shelf hardware instead of requiring expensive and specialized accelerators.

A. Test platform

All tests were conduct on an Acer Aspire M5811, equipped with a *Samsung Series 830* 128 GiB SATA 3 SSD in order to decrease the biasing effects of IO performance on programs.

The test computer contains 6 GiB of Non-ECC DDR3 RAM, clocked with 1333 MHz and uses a Core i7 860 CPU, clocked at 2.80 GHz.

As graphics card for GPGPU applications we used a Cypress PRO Radeon HD 580 AMD card, clocked at 725 MHz, with the 1 GiB GDDR5 memory clocked at 1,000 MHz. Although we could have overclocked the GPU to improve our results, we intended to benchmarks with minimized bias. Furthermore, we have the experience that overclocking often leads to tasks failing or providing incorrect results.

As software backend, we used Ubuntu 13.10 with a 3.13.0 vanilla kernel optimized for Core2+ x86_64 with default settings and AMD CPU microcode removed . We used both `g++` 4.8.1 from the distribution package sources and `clang++` 3.4 FINAL vanilla as compiler for the *CLSW* C++ code (with negligible runtime differences). As graphics card driver, we used vanilla AMD catalyst 12.9 and AMD APP SDK version 2.9 as *OpenCL* provider.

B. Bibliography

References

[ACL+09] Allred, Jeffrey ; Coyne, Jack ; Lynch, William ; Natoli, Vincent ; Grecco, Joseph ; Morrissette, Joel: Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* IEEE, 2009, S. 1–4

[AH11] Agrawal, Ankit ; Huang, Xiaoqiu: Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 8 (2011), Nr. 1, S. 194–205

[And04] Anderson, David P.: Boinc: A system for public-resource computing and storage. In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on* IEEE, 2004, S. 4–10

[ART+05] Arnold, Roland ; Rattei, Thomas ; Tischler, Patrick ; Truong, Minh-Duc ; Stümpflen, Volker ; Mewes, Werner: SIMAP—the similarity matrix of proteins. In: *Bioinformatics* 21 (2005), Nr. suppl 2, S. ii42–ii46

[AWG+05] Altschul, Stephen F. ; Wootton, John C. ; Gertz, E M. ; Agarwala, Richa ; Morgulis, Aleksandr ; Schäffer, Alejandro A. ; Yu, Yi-Kuo: Protein database searches using compositionally adjusted substitution matrices. In: *Febs Journal* 272 (2005), Nr. 20, S. 5101–5109

[BBH12] Bustamam, Alhadi ; Burrage, Kevin ; Hamilton, Nicholas A.: Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 9 (2012), Nr. 3, S. 679–692

[BCP07] Bandeira, Nuno ; Clauser, Karl R. ; Pevzner, Pavel A.: Shotgun Protein Sequencing Assembly of Peptide Tandem Mass Spectra from Mixtures of Modified Proteins. In: *Molecular & Cellular Proteomics* 6 (2007), Nr. 7, S. 1123–1134

[CAD+12] Czajkowski, Tomasz S. ; Aydonat, Utku ; Denisenko, Dmitry ; Freeman, John ; Kinsner, Michael ; Neto, David ; Wong, Jason ; Yiannacouras, Peter ; Singh, Deshanand P.: From OpenCL to high-performance hardware on FPGAs. In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on* IEEE, 2012, S. 531–534

[CSKM12] Cramer, Tim ; Schmidl, Dirk ; Klemm, Michael ; Mey, Dieter an: OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison. In: *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, 2012

[CTS05] Charalambous, Maria ; Trancoso, Pedro ; Stamatakis, Alexandros: Initial experiences porting a bioinformatics application to a graphics processor. In: *Advances in Informatics*. Springer, 2005, S. 415–425

[DBL+10] Dohi, Keisuke ; Benkrid, Khaled ; Ling, Cheng ; Hamada, Tsuyoshi ; Shibata, Yuichiro: Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs. In: *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on* IEEE, 2010, S. 29–36

[DMM+10] Danalis, Anthony ; Marin, Gabriel ; McCurdy, Collin ; Meredith, Jeremy S. ; Roth, Philip C. ; Spafford, Kyle ; Tipparaju, Vinod ; Vetter, Jeffrey S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* ACM, 2010, S. 63–74

[DWL+12] Du, Peng ; Weber, Rick ; Luszczek, Piotr ; Tomov, Stanimire ; Peterson, Gregory ; Dongarra, Jack: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. In: *Parallel Computing* 38 (2012), Nr. 8, S. 391–407

[DWRR08] Döring, Andreas ; Weese, David ; Rausch, Tobias ; Reinert, Knut: SeqAn an efficient, generic C++ library for sequence analysis. In: *BMC bioinformatics* 9 (2008), Nr. 1, S. 11

[Fag10] Fagerlund, Olav A.: Multi-core programming with OpenCL: performance and portability: OpenCL in a memory bound scenario. (2010)

[Far10] Farber, R: Introduction to gpgpus and massively threaded programming. In: *Bioinformatics High Performance Parallel Computer Architectures* (2010), S. 29–48

[Fog14] Fog, Agner: *Optimizing Software in C++*. `http://agner.org/optimize/optimizing_cpp.pdf`. Version: 02 2014. – Accessed 2014-02-13

[FVS11] Fang, Jianbin ; Varbanescu, Ana L. ; Sips, Henk: A comprehensive performance comparison of CUDA and OpenCL. In: *Parallel Processing (ICPP), 2011 International Conference on* IEEE, 2011, S. 216–225

[GJL97] Guerdoux-Jamet, Pascale ; Lavenier, Dominique: SAMBA: hardware accelerator for biological sequence comparison. In: *Computer applications in the biosciences: CABIOS* 13 (1997), Nr. 6, S. 609–615

[GJS74] Garey, Michael R. ; Johnson, David S. ; Stockmeyer, Larry: Some simplified NP-complete problems. In: *Proceedings of the sixth annual ACM symposium on Theory of computing* ACM, 1974, S. 47–63

[GML+10] Goujon, Mickael ; McWilliam, Hamish ; Li, Weizhong ; Valentin, Franck ; Squizzato, Silvano ; Paern, Juri ; Lopez, Rodrigo: A new bioinformatics analysis tools framework at EMBL–EBI. In: *Nucleic acids research* 38 (2010), Nr. suppl 2, S. W695–W699

[Got82] Gotoh, Osamu: An improved algorithm for matching biological sequences. In: *Journal of molecular biology* 162 (1982), Nr. 3, S. 705–708

[GZA+11] Grosser, Tobias ; Zheng, Hongbin ; Aloor, Raghesh ; Simbürger, Andreas ; Grösslinger, Armin ; Pouchet, Louis-Noël: Polly-Polyhedral optimization in LLVM. In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)* Bd. 2011, 2011

[Har09] Harvey, Matt: Experiences porting from CUDA to OpenCL. In: *Imperial College London CBBL IMIM* (2009)

[Hay09] Haynberg, Rolf: *Der Gotoh-Algorithmus*. Electronic. `http://blog.haynberg.de/wp-content/uploads/2009/11/gotoh.pdf`. Version: 2009. – Accessed 2014-03-12

[HCO+11] Hains, Doug ; Cashero, Zach ; Ottenberg, Mark ; Bohm, Wim ; Rajopadhye, Sanjay: Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* IEEE, 2011, S. 490–501

[HHM+02] Herz, Michael ; Hartenstein, Reiner ; Miranda, Miguel ; Brockmeyer, Erik ; Catthoor, Francky: Memory addressing organization for stream-based reconfigurable computing. In: *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems*, 2002

[HJL+07] Harris, Brandon ; Jacob, Arpith C. ; Lancaster, Joseph M. ; Buhler, Jeremy ; Chamberlain, Roger D.: A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on* IEEE, 2007, S. 765–769

[JKS+09] Jensen, Lars J. ; Kuhn, Michael ; Stark, Manuel ; Chaffron, Samuel ; Creevey, Chris ; Muller, Jean ; Doerks, Tobias ; Julien, Philippe ; Roth, Alexander ; Simonovic, Milan u. a.: STRING 8—a global view on proteins and their functional interactions in 630 organisms. In: *Nucleic acids research* 37 (2009), Nr. suppl 1, S. D412–D416

[JLX+07] Jiang, Xianyang ; Liu, Xinchun ; Xu, Lin ; Zhang, Peiheng ; Sun, Ninghui: A Reconfigurable Accelerator for Smith–Waterman Algorithm. In: *Circuits and Systems II: Express Briefs, IEEE Transactions on* 54 (2007), Nr. 12, S. 1077–1081

[JR13] Jeffers, James ; Reinders, James: *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013

[KDH10] Karimi, Kamran ; Dickson, Neil G. ; Hamze, Firas: A performance comparison of CUDA and OpenCL. In: *arXiv preprint arXiv:1005.2581* (2010)

[KSA+10] Komatsu, Kazuhiko ; Sato, Katsuto ; Arai, Yusuke ; Koyama, Kentaro ; Takizawa, Hiroyuki ; Kobayashi, Hiroaki: Evaluating performance and portability of OpenCL programs. In: *The fifth international workshop on automatic performance tuning*, 2010

[LA02] Lattner, Chris ; Adve, Vikram: The LLVM instruction set and compilation strategy. In: *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS* (2002)

[LA04] Lattner, Chris ; Adve, Vikram: LLVM: A compilation framework for lifelong program analysis & transformation. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* IEEE, 2004, S. 75–86

[Lan12] Langdon, WB: Initial experiences of the emerald: e-infrastructure south GPU supercomputer. In: *RN* 12 (2012), Nr. 08, S. 08

[Lat02] Lattner, Chris A.: *LLVM: An infrastructure for multi-stage optimization*, University of Illinois, Diss., 2002

[LBH09] Ling, Cheng ; Benkrid, Khaled ; Hamada, Tsuyoshi: A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In: *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on* IEEE, 2009, S. 94–100

[LMS09] Liu, Yongchao ; Maskell, Douglas L. ; Schmidt, Bertil: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. In: *BMC research notes* 2 (2009), Nr. 1, S. 73

[LP85] Lipman, David J. ; Pearson, William R.: Rapid and sensitive protein similarity searches. In: *Science* 227 (1985), Nr. 4693, S. 1435–1441

[LR09] Ligowski, Lukasz ; Rudnicki, Witold: An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* IEEE, 2009, S. 1–8

[LSM10] Liu, Yongchao ; Schmidt, Bertil ; Maskell, Douglas L.: CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. In: *BMC research notes* 3 (2010), Nr. 1, S. 93

[LSS+02] Larson, Stefan M. ; Snow, Christopher D. ; Shirts, Michael u. a.: Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology. (2002)

[LWS13] Liu, Yongchao ; Wirawan, Adrianto ; Schmidt, Bertil: CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. In: *BMC bioinformatics* 14 (2013), Nr. 1, S. 117

[MV08] Manavski, Svetlin A. ; Valle, Giorgio: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. In: *BMC bioinformatics* 9 (2008), Nr. Suppl 2, S. S10

[MY02] McDonald, W H. ; Yates, John R.: Shotgun proteomics and biomarker discovery. In: *Disease markers* 18 (2002), Nr. 2, S. 99–105

[Ni09] Ni, Jun: CUDA and OpenCL—Development Interfaces for Multicore Programming. In: *Consortium of College of Computer Science & Technology, Harbin Engineering University, China* (2009)

[NVi12] NVidia: *NVIDIA CUDA C Programming Guide*. http://www.cs.unc.edu/~prins/Classes/633/Readings/CUDA_C_Programming_Guide_4.2.pdf. Version: 2012. – Accessed 2014-03-13, Version 4.2

[Pea91] Pearson, William R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. In: *Genomics* 11 (1991), Nr. 3, S. 635–650

[RMG+10] Razmyslovich, Dzmitry ; Marcus, Guillermo ; Gipp, Markus ; Zapatka, Marc ; Szillus, Andreas: Implementation of Smith-Waterman algorithm in OpenCL for GPUs. In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on* IEEE, 2010, S. 48–56

[Rog11] Rognes, Torbjørn: Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. In: *BMC bioinformatics* 12 (2011), Nr. 1, S. 221

[RSMB04] Rohl, Carol A. ; Strauss, Charlie E. ; Misura, Kira M. ; Baker, David: Protein structure prediction using Rosetta. In: *Methods in enzymology* 383 (2004), S. 66–93

[RTA+08] Rattei, Thomas ; Tischler, Patrick ; Arnold, Roland ; Hamberger, Franz ; Krebs, Jörg ; Krumsiek, Jan ; Wachinger, Benedikt ; Stümpflen, Volker ; Mewes, Werner: SIMAP—structuring the network of protein similarities. In: *Nucleic acids research* 36 (2008), Nr. suppl 1, S. D289–D292

[RTG+10] Rattei, Thomas ; Tischler, Patrick ; Götz, Stefan ; Jehl, Marc-Andre ; Hoser, Jonathan ; Arnold, Roland ; Conesa, Ana ; Mewes, Hans-Werner: SIMAP—a comprehensive database of pre-calculated protein sequence similarities, domains, annotations and clusters. In: *Nucleic acids research* 38 (2010), Nr. suppl 1, S. D223–D226

[RVDDB10] Rul, Sean ; Vandierendonck, Hans ; D'Haene, Joris ; De Bosschere, Koen: An experimental study on performance portability of OpenCL

kernels. In: *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010

[SAM+01] Schäffer, Alejandro A. ; Aravind, L ; Madden, Thomas L. ; Shavirin, Sergei ; Spouge, John L. ; Wolf, Yuri I. ; Koonin, Eugene V. ; Altschul, Stephen F.: Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements. In: *Nucleic acids research* 29 (2001), Nr. 14, S. 2994–3005

[Sch11] Schmidt, Bertil: *Bioinformatics: High Performance Parallel Computer Architectures*. CRC Press, 2011

[SFK+11] Szklarczyk, Damian ; Franceschini, Andrea ; Kuhn, Michael ; Simonovic, Milan ; Roth, Alexander ; Minguez, Pablo ; Doerks, Tobias ; Stark, Manuel ; Muller, Jean ; Bork, Peer u. a.: The STRING database in 2011: functional interaction networks of proteins, globally integrated and scored. In: *Nucleic acids research* 39 (2011), Nr. suppl 1, S. D561–D568

[SJL11] Seo, Sangmin ; Jo, Gangwon ; Lee, Jaejin: Performance characterization of the NAS Parallel Benchmarks in OpenCL. In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on* IEEE, 2011, S. 137–148

[SW81] Smith, Temple F. ; Waterman, Michael S.: Identification of common molecular subsequences. In: *Journal of molecular biology* 147 (1981), Nr. 1, S. 195–197

[TLHC14] Tran, Nhat-Phuong ; Lee, Myungho ; Hong, Sugwon ; Choi, Dong H.: Multi-stream Parallel String Matching on Kepler Architecture. In: *Mobile, Ubiquitous, and Intelligent Computing*. Springer, 2014, S. 307–313

[TNI+10] Tsuchiyama, Ryoji ; Nakamura, Takashi ; Iizuka, Takuro ; Asahara, Akihiro ; Miki, Satoshi ; Tagawa, Satoru: The OpenCL Programming Book. In: *Fixstars Corporation* 63 (2010)

[VDRN11] Vidanagamachchi, SM ; Dewasurendra, SD ; Ragel, RG ; Niranjan, M: Tile optimization for area in FPGA based hardware acceleration of peptide identification. In: *Industrial and Information Systems (ICIIS), 2011 6th IEEE International Conference on* IEEE, 2011, S. 140–145

[VLM11] Vigelius, Matthias ; Lane, Aidan ; Meyer, Bernd: Accelerating reaction–diffusion simulations with general-purpose graphics processing units. In: *Bioinformatics* 27 (2011), Nr. 2, S. 288–290

[VMHJ+03] Von Mering, Christian ; Huynen, Martijn ; Jaeggi, Daniel ; Schmidt, Steffen ; Bork, Peer ; Snel, Berend: STRING: a database of predicted functional associations between proteins. In: *Nucleic acids research* 31 (2003), Nr. 1, S. 258–261

[WDJ+07] Wilson, Justin ; Dai, Manhong ; Jakupovic, Elvis ; Watson, Stanley ; Meng, Fan: Supercomputing with toys: harnessing the power of NVIDIA 8800GTX and playstation 3 for bioinformatics problem. In: *Comput Syst Bioinformatics Conf* Bd. 6, 2007, S. 387–390

[YA05] Yu, Yi-Kuo ; Altschul, Stephen F.: The construction of amino acid substitution matrices for the comparison of proteins with non-standard compositions. In: *Bioinformatics* 21 (2005), Nr. 7, S. 902–911

[ZMA+12] Zhang, Yuhong ; Misra, Sanchit ; Agrawal, Ankit ; Patwary, Md Mostofa A. ; Liao, Wei-keng ; Qin, Zhiguang ; Choudhary, Alok: Accelerating pairwise statistical significance estimation for local alignment by harvesting GPU's power. In: *BMC bioinformatics* 13 (2012), Nr. Suppl 5, S. S3

C. Acknowledgment

We would like to thank Mathias Walter[63] and Thomas Rattei[64] who made this project possible with their continuous and outstanding support.

[63]Helmholtz-Zentrum München - Deutsches Forschungszentrum für Gesundheit und Umwelt
[64]CUBE - University of Vienna