Lab Assignment 3
OpenCL Implementation of an Algorithm
EE4C07 Advanced Computing Systems
(2017/18)

Smith-Waterman Protein Alignment Algorithm on GPU using OpenCL

Submitted By:

GROUP 15
Amitabh Yadav, Student No. 4715020
Pedro Costa, Student No. 4699424
Shardul Rautmare, Student No. 4750950
Delft University of Technology

November 30, 2017

1 Introduction

Background Alignment algorithms find application in the field of Bioinformatics to investigate the similarity between biological sequences, such as DNA and Proteins. The identified similarity of a *query sequence* to a *database sequence* can be used to find commonalities between organisms, or to infer an ancestral relation. Smith-Waterman (S-W) Algorithm, based on *dynamic programming*, is a sequence alignment method for such biological sequences. The computational complexity of the S-W Algorithm makes it slow for aligning long sequences. Moreover, the speedup does not increase linearly with number of CPUs due to workload distribution.

Given Lab Assignment 3 includes a sequential and a parallel implementation of the S-W Algorithm in C and CUDA, respectively.

Objective OpenCL (Open Computing Language) is a general-purpose parallel programming open-standard that supports multi-vendor platforms for heterogeneous systems. The objective of this assignment is to implement and optimise the performance of the Smith-Waterman Protein Alignment algorithm for GPU using the OpenCL. The tasks assigned are as follows:

1. Run and measure the performance of the Sequential C Program.
2. Run and measure the performance of CUDA Program.
3. Implement the Protein Alignment S-W Algorithm in OpenCL for GPU
4. Measure Performance of OpenCL implementation and identify the bottlenecks
5. Analyse the bottlenecks and optimise the performance of OpenCL implementation

We start with the discussion of Sequential performance of the algorithm when running on a CPU (Section 2), the encountered bottlenecks and investigated solutions. In Section 3, we discuss the CUDA performance, the improvements and profiling. Section 4 covers the design methodology and implementation of S-W Protein Alignment Algorithm in OpenCL. Finally, the OpenCL implementation time is recorded and performance comparisons against given CPU and CUDA implementations are reported.

The GPU used for this assignment and for benchmarking is an Nvidia Tesla K40c through `ce-cuda01.ewi.tudelft.nl`.

2 Sequential Performance of S-W Algorithm: SW_C

The sequential execution of the S-W Algorithms is a straightforward approach that loads the query sequence, substitution matrix and the database sequences from memory. Using this data, the kernel is invoked. The kernel is responsible for aligning and calculating the score for each sequence combination. This is done one sequence at a time from the database sequence, using a `for` loop to calculate the alignments for the length of the query.

Performance The Swiss-Prot `uniprot_sprot.fasta` database sequence contains **556008 sequences**. Running the sequential implementation (SW_C), with `Q9UKN1.fasta` as the query sequence, we obtained an average execution time of 0.48 seconds for processing each sequence. This was done by running part of the sequence for some minutes, and then cancelling execution. Extrapolating from this, we get a total execution time of **3.08 days** to complete the full database.

In order to get a parameter for comparison of performance of the sequential and parallel implementation, we extracted **762 sequences** from the `uniprot_sprot.fasta` database and executed the sequential code with `Q9UKN1.fasta` as the query sequence. The initialisation time, comprised by the load query sequence time, load substitution matrix time, create query profile time and load database time, recorded is **0.05 seconds**. The Total Execution Time was **398.45 seconds**. The final performance calculated is **0.003124 GCUPS**.

The results are shown in figures 1 and 2. From figure 1, we can interpret that the execution time of the sequences is *not* linear.

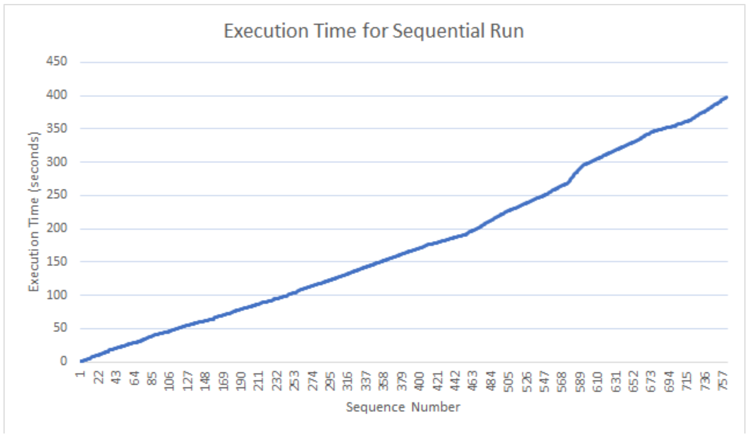

Figure 1: Execution Time for Sequential Code

Identification of Bottlenecks and Suggested Improvements

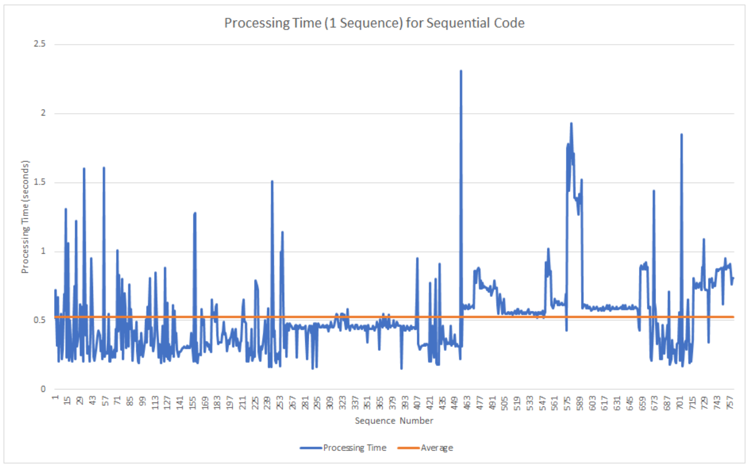

Figure 2: Processing Time for 1 Sequence (Sequential Code)

1. The Sequential Implementation calculates the scores matrix for one sequence at a time, which takes maximum time. This section of the S-W Algorithm can be implemented in parallel and accelerated on a GPU.

2. The Database Sequence `uniprot_sprot.fasta`, contains various descriptions of the names and biological information of the sequences which are not necessary in calculation of the scores using S-W Algorithm. This in turn increases the database sequence size and creates a Memory Bandwidth Bottleneck. This can overcome by removing the descriptions from the database sequence.

3. From Figure 2, we see that processing time for some of the databases are more than other, thus we can interpret that the database sequences are unsorted based on the sequence length. This increases the waiting time before each workload is finished. Thus, restructuring and reorganising the database helps in minimising the waiting time of the threads running simultaneously.

3 CUDA Performance of S-W Algorithm: SW_CUDA

The CUDA implementation for GPU (device) programming alongwith C++ for CPU (host) programming is used to implement the S-W Algorithm. The host code loads the data structures from the CPU memory to GPU memory; copies them back and presents the results. The host code passes the query sequence and other parameters as arguments to the CUDA kernel that runs on the GPU. The kernel implements the S-W Algorithm which aligns the query sequence with the database sequence. The kernel computes and returns the maximum scores. This result is then sorted and by default top 20 scores are returned. Each processing element then generates complete alignment between a query sequence and a database sequence. Therefore, we have better resource utilisation as there is no need for different processing elements to communicate with each other.

File Formats: FASTA and GPUDB It was identified by the sequential performance that the database sequence (FASTA format) `uniprot_sprot.fasta` contains sequence descriptions such as name and other biological information. This description precedes every sequence in the database. However, this information is not used during protein alignment operation of the query sequence with the database sequence. Moreover, the sequences are unsorted based on sequence length. Thus, to eliminate the memory bottleneck the implemented strategy in the CUDA implementation is restructure the database in a new format. This new database is referred to as GPUDB. This database is converted only once and takes around 1.6 seconds. This database conversion operation is implemented in `SW_CUDA/DOPA/dbconv/` and the output for the same are: `out.gpudb`, the restructured database file; and `out.gpudb.descs`, the descriptions of the sequences of database. The GPUDB file is therefore used in the protein alignment implementation in CUDA.

Performance Optimisation in the CUDA Algorithm .
Thread Occupancy: Workload is evenly distributed between the processing units. This achieved by computing various sequences in a half warp consisting of 16 threads. With workload evenly distributed, the threads within the same half warp do not have to wait for long before the next sequences are available for processing.

Memory Bandwidth Optimisation: To optimise the use of memory bandwidth as efficiently as possible, some of the interesting optimisations done in the CUDA algorithm are as follows: Memory Coalescing: Each thread of a half warp access a 4 byte value. These values are stored in unordered different addresses; and to access these, each thread will execute a 32 byte memory access sequentially (which is the least memory access size for the GPU), thus wasting 28 bytes of bandwidth per access. Memory Coalescing fetching of these values from neighbouring memory addresses. Thus, when a 64-byte memory access is done by the GPU for 16 threads, no memory bandwidth is wasted.

Textured Memory: The protein alignment method makes use of substitution matrix. This substitution matrix is loaded in a `map` data structure which is converted to array based format or *query profile*. Textures are cached 'windows' into global memory, optimized for spatially local reads that offers lower latency and is well suited for random access. It allows fetching four values at a time and therefore offers yet more speed up.

Results We tested the CUDA implementation on the GPU by using the `Q9UKN1.fasta` as the query sequence and the gpudb file, `out.gpudb` which was converted using `uniprot_sprot.fasta`. The total execution time of the same was **24.21 seconds**. And the performance recorded was **45.08 GCUPS**.

Profiling The profiling of the CUDA program of the S-W Algorithm was done using the NVIDIA Visual Profiler, which helped analyse the code with the perspective of hardware resource utilisations. The results of the same are summarised as follows:

1. The kernel performance bounded by computation, bandwidth and instruction/memory latency is shown. It is observed that there is a good utilisation of processing power; with low memory utilisation. (Figure 3)

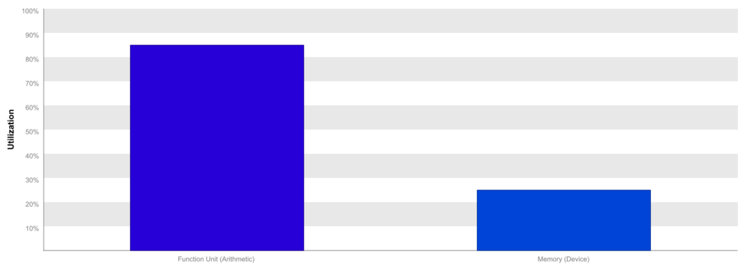

Figure 3: Kernel Performance by Compute Resource Utilisation

2. The execution instructions are not branched and there is high warp execution efficiency.

3. The S-W Algorithm is compute intensive. This is observed from the utilisation of functional units (Figure 4). Whereas, resource utilisation of other functional units is low.

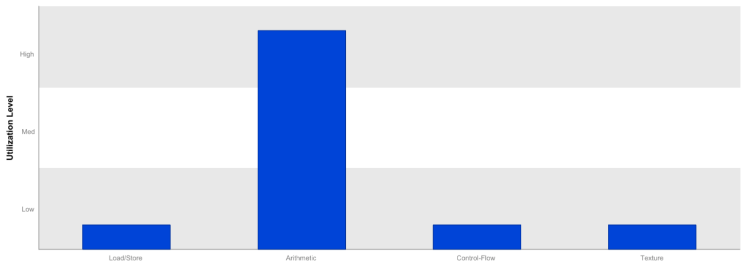

Figure 4: Kernel Performance by Resource Utilisation of Functional Units

4. Memory Bandwidth creates a bottleneck when the GPU is not able to provide data at the rate requested by the kernel. This bottleneck is apt overcome by wise bandwidth optimisation by memory coalescing. The results of profiling clearly show low memory bandwidth utilisation.

5. The occupancy of the Streaming Multiprocessor is high and workload is almost equally distributed. This is clearly observed in Figure 5.

6. Figure 6 shows different stall reasons for the kernel execution. It also points at high utilisation of texture memory, which is good for the

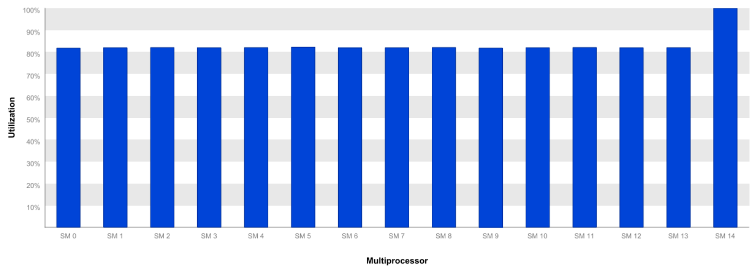

Figure 5: Workload distribution between processors

overall performance. The major contributors to stall reasons other than Instruction Dependency, are Pipeline Busy and Memory Dependency. The GPU is limited by compute resources as it is already performing with high compute resource utilisation. However, improvements in the memory access can be suggested to reduce the stall time during processing.

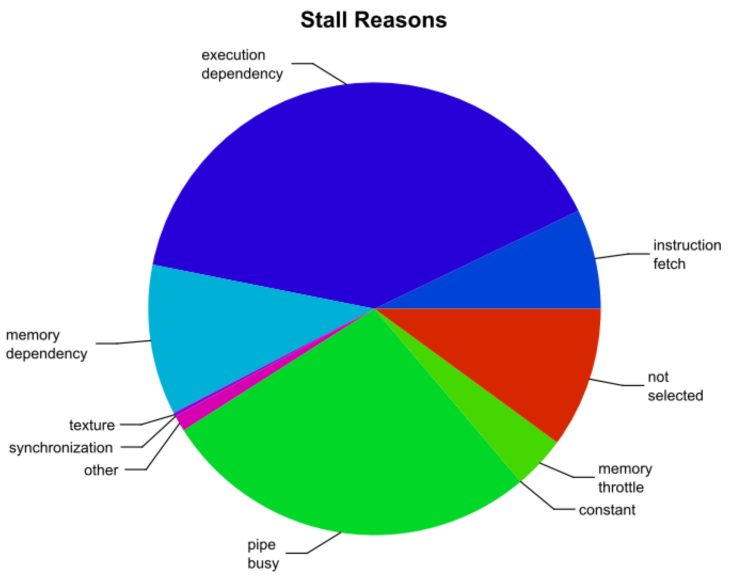

Figure 6: Stall Reasons during Kernel Execution

Identification of Bottlenecks The implementation of CUDA S-W Algorithm is done on NVIDIA Tesla K40c GPU, which produced a high performance result of 45.08 GCUPS. This implementation already gives a speed-up of **11023.7**, which is again very good.

However, looking at the CUDA Profiling of the kernel, we observe the following bottlenecks that can be considered to improve the performance further:

1. The GPU implementation of the CUDA Protein Alignment Algorithm is a standard for a current generation of GPUs and it would require to

optimisations for newer GPUs. For example, it uses one half-warp at a time to compute threads whereas newer GPUs can work with two half warps simultaneously. This calls in for redefining memory layout.

2. The number of blocks depends on the length of the longest sequence, therefore there will be less sequence groups; which will result in low compute resource utilisation.

3. Memory Utilisation is very low on the GPU hardware and is one of the reasons for instruction stalling. Therefore, data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

4 OpenCL Implementation of S-W Algorithm: SW_OpenCL

OpenCL is a general-purpose multi-vendor open-standard parallel programming API for heterogeneous systems. This implementation is open to different heterogeneous platforms such as CPU, GPU, FPGA etc. unlike CUDA, which is specific only for NVIDIA GPU. This section discusses the OpenCL implementation of the S-W Algorithm for Protein Alignment on GPU.

Given A C++ code format for host code of the Protein Alignment Problem is provided. The code has implemented the task of loading query sequence (`fasta`), database sequence (`gpudb`)and the substitution matrix (`blosum62.mat`). Further, the processing of the records, creating query profile from substitution matrix and loading database and database description has been done and provided. The result sorting and return output has also been implemented and provided. Various header file variables and functions that were defined in `gpudb.h` have been defined in `main.h`.

File Formats: FASTA and GPUDB The database sequences are available as `fasta` format. For reasons, discussed in the CUDA implementation, this format is converted to more convenient format, named `gpudb`. This format is a structured format that is created using the `SW_CUDA/DOPA/dbconv/` program. The input to this program is a `fasta` file which is processed into an optimised format and the program returns: 1] `out.gpudb` (the structured database sequence) and the 2] `out.gpudb.descs` (the descriptions of the database sequences).

In OpenCL implementation, parameters are specified to work using the `gpudb` format for database sequence. Therefore,this format is used.

Methodology The OpenCL implementation of the Protein Alignment Algorithm makes use of the functions built in the OpenCL API. This `main.cpp` code is the host code for the implementation and runs on the CPU. This host code is responsible for accepting input arguments, initialising the OpenCL for GPU implementation using API functions, loading the substitution matrix, query and sequence database , creating the query profile, implement the kernel and retrieve the results.

We follow the methodology of porting the CUDA code to OpenCL. This has to be done in the following steps: 1. Initialising Platform, 2. Initialising Devices, 3. Creating Context, 4. Creating Command Queue, 5. Creating Buffers and copying data to GPU, 6. Creating and Compiling the program, 7. Defining Kernel Arguments, 8. Creating Kernel, 9. Configuring Work-item structure, 10. Reading the Result, and 11. Releasing the OpenCL resource such as kernel, memory etc.

The kernel of the S-W Algorithm is implemented in the `kernel.cl` file. We start the kernel implementation by passing on the following arguments: 1. numGroups; 2. scores array; 3.blockOffsets; 4. seqNums; 5. sequences; 6. tempColumns; 7. queryLength; 8. gapPenalty and 9. gapPenaltyTotal. The kernel uses the get_global_id(0) function to access groups in the compute unit. These groups correspond to the number of multiprocessors in the GPU. Each group calls the align function which implements the alignment method on the query and database sequences using the query profile. One implementation that we have performed differently from the CUDA code is not using the texture memory to store the query profile elements. These are stored in the global memory and values are fetched randomly. Also, the kernel uses the vector variables for memory coalescing.

Results and Performance **Following the model of kernel programming from CUDA, we were able to successfully implement the protein alignment S-W Algorithm in OpenCL.**

The performance of the same was measured by the initialisation time, host to device copy time and kernel execution time. The initialisation time of the openCL host code recorded was **0.224628 seconds**, the host to device memory copy time recorded was **0.030447 seconds** and the kernel execution time recorded was **61.21 seconds**. Using the kernel execution time we can calculate the performance of the OpenCL application in GCUPS, as follows:
$GCUPS = numCells/seconds/(10^9)$
where, $numCells = QueryLength * NumberOfSymbols$.
Using the above formula, we have
$seconds = KernelExecutionTime = 61.21$,
$QueryLength = 5478$, and
$NumberOfSymbols = 199234755$ (see stdout.log file for OpenCL).
Thus, performance of OpenCL in GCUPS = 17.83 GCUPS.

The obtained result, in contrast with the sequential execution show performance speedup however, it performs slow in comparison to the CUDA implementation of the same. We identify some of the bottlenecks of the implementation in the section `Identification of Bottlenecks and Suggested`

`Improvements`.

Verification of Correctness We made the use of diff command to compare the output results of CUDA and OpenCL implementation. The OpenCL output file was retrieved from the run folder for verifying the correctness of the results. In order to compare the alignment results of all the sequences, we removed the result sorting step from the host code for one run to get all the results. All the sequence score were compared and verified for the correctness of the result. The stdout.log file for both CUDA and OpenCL implementation has been provided.

Identification of Bottlenecks and Suggested Improvements The OpenCL code for the S-W Algorithm has been implemented using the CUDA kernel implementation and therefore address all the bottlenecks identified in the sequential implementation. However, the CUDA implementation uses the texture memory to store the substitution matrix elements which is used in aligning the protein sequences. The memory access to this memory type is random and very frequent. Since the texture memory is faster than global memory, therfore, by using texture memory, the CUDA implementation improves the performance of the kernel. By not using the texture memory in the OpenCL implementation we are creating the memory bandwidth bottleneck, which causes the performance loss compared to the CUDA implementation.

However, in OpenCL implementation we move the storage of substitution matrix to global memory. This memory is conveniently accessible to the kernel for performing alignment. However, in this implementation we encounter texture memory as a bottleneck. This can be overcome by looking into texture memory implementation in OpenCL, such as `clCreateImage`.

5 Conclusion and Further Improvements

This report documents the task of implementation of S-W Algorithm for Protein Alignment using OpenCL.

The algorithm is initially implemented sequentially in C++. The S-W Algorithm has high computational complexity and therefore the sequential implementation performs poorly. The results of the running the sequence alignment on 762 sequences are shown in Figure 1 and 2. The sequential implementation shows that the task of calculating the scores is done individually for each element takes maximum time; and therefore indicates that the program can be implemented in parallel. We also identify the memory bandwidth bottleneck as the database sequence contains various description parameters for each sequence which has no role to play in the S-W Algorithm, which can be optimised.

The parallel implementation of the protein alignment problem is done in CUDA. CUDA implementation has been highly optimised to provide maximum performance. The CUDA implementation yielded a performance of 45.08 GCUPS on NVIDIA Tesla K40c GPU. The main bottlenecks of sequential implementation was addressed by using *gpudb* format for database sequences which structures the database. This structured database contains *sequence groups* of equal sequence set. This format therefore allows equal distribution of workload to different processing elements of the GPU (Streaming Multiprocessors). The CUDA implementation further makes use of memory coalescing for fetching values per half-warp which massively decreases the memory bandwidth utilisation. Further, the use of texture memory, which is faster than global memory is used to fetch 4 values from the queryprofile at once instead of 1 value. This again adds up to improve the overall performance. The profiling of the CUDA kernel is done using the NVIDIA Visual Profiler which shows the equal distribution of workload between different multiprocessors and a high utilisation of the compute capacity of the GPU. By studying the instruction stall reasons, we find that pipeline busy and memory dependency are the main contributer to stalls, which indicates for improvements in design of memory access.

The parallel programming approach is then implemented in OpenCL. In this section, all the initializations for platform and devices was implemented from scratch, followed by identification of the kernel arguments for implementation of S-W Algorithm and writing the kernel using the approach in CUDA implementation. One of the major changes that was implemented in the OpenCL kernel was not using the textured memory for queryProfile access. Using textured memory for performing the alignment operation has speed up advantage in CUDA implementation for three reasons: it is faster compared to global memory, it is suitable for random accesses, and it allows accessing 4 elements at a time. However, the implementation of the same was not feasible with porting the CUDA code, so instead the global memory was used. Various openCL functions were used to debug the code and different instances to investigate the bottlenecks and record the execution time. The Final OpenCL implementation fetched a performance of *17.83 GCUPS* with the Kernel Execution Time being *61.21 seconds.* This mitigations to identified bottlenecks were use of texture memory and the possibility of improving memory access through design.

A comparison of the results of sequential and parallel execution of S-W Algorithm for Protein Alignment is presented in the following table:

Parameter	Sequential Execution	CUDA Execution	OpenCL Execution
Initialisation Time	0.05 sec	0.46 sec	0.224628 sec
HosttoDevice Memory Copy Time	N/A	0.03 sec	0.030447 sec
Kernel Execution Time	3.08 days (approx.)	24.21 sec	61.21 sec
GCUPS	0.003124 GCUPS	45.09 GCUPS	17.83 GCUPS

The Protein Alignment S-W Algorithm was, thus, successfully implemented in OpenCL and results reported and analysed.

References

[1] Hasan, L., Kentie, M. and Al-Ars, Z., 2011. DOPA: GPU-based protein alignment using database and memory access optimizations. BMC research notes, 4(1), p.261.

[2] Kentie, M. A. 2010, 'Biological Sequence Alignment Using Graphics Processing Units', Master Of Science Thesis, Delft University of Technology, `http://www.kentie.net/article/thesis/thesis.pdf`.

[3] OpenCL API 1.2 Reference Card, Khronos Group. `https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf`