Delft University of Technology

Advanced Computing System EE4C07

LAB REPORT

OpenCL : Implementation of Protein Alignment

Group 22

Author: *Email:*

Ade Setyawan Sajim (4608232) A.Sajim@student.tudelft.nl

Priadi Teguh Wibowo (4625862) PriadiTeguhWibowo@student.tudelft.nl

1 February 2017

Contents

1 Introduction

Protein sequence alignment is a method to arrange the sequences of protein to identify regions of similarity which might be an effect of functional, structural, or evolutionary relationships between the sequences [5]. Similarities between sequences can be utilized to identify to find commonalities between organisms, or to infer an ancestral relation [4]. There are many approaches to perform protein sequence alignment, and two of the most common is dynamic programming and heuristic methods. Most common characteristic of using heuristic methods is faster computation but with a drawback, which is less accurate. On the other side, dynamic programming methods offers a guaranteed optimal alignments, with a relatively slow performance. Dynamic programming itself is a way of breaking down a problem into smaller subproblems [4].

An alignment algorithm also able to be divided based on the result. There are 2 types of alignment based on result orientation; global alignment and local alignment. In a global alignment, a comparison is performed on end to end alignment with the subject and therefore it may end up with a lot of gaps in global alignment if the sizes of query and subject are dissimilar. In the other hand, in a local alignment, an observation is done by trying to match query with a sub-string of subject[7].

1.1 Smith-Waterman Protein Alignment

An example of sequence alignment algorithm is Smith-Waterman. Smith-Waterman algorithm uses dynamic programming method on protein alignment and has an ability to produce local alignment.

1.1.1 BLOSUM

During alignment procedures, Smith-Waterman algorithm uses scoring system, implemented in matrices. In the matrices, positive or higher value is assigned for a match and a negative or a lower value is assigned for mismatch. These assumption based scores can be used for scoring the matrices. One of the most famous scoring matrix type is BLOSUM. BLOSUM stands for Blocks Substitution Matrix. The value inside matrices are actual percentage of similarity on identity values. In this lab assignment, BLOSUM version that was used for testing is blosum62.

1.1.2 Gap Penalties

Another feature that exist in Smith-Waterman algorithm is gap penalties. Gap penalties is used to maximize the biological meaning and defines a penalty given to alignment when there is insertion or deletion[7]. For each gap that has been introduced, gap penalty is subtracted. Most common examples of gap penalty are gap open and gap extension. Gap open and gap extension is introduced when there are continuous gaps along the sequence since using linear gap penalty is not accurate. The open penalty is always applied at the start of the gap, and the other gaps follow are given with a gap extension penalty which will be less compared to the open penalty. In this lab assignment, gap open penalty is set to -10 and gap extension penalty is set to -2.

1.1.3 Smith-Waterman Algorithm

In Smith-Waterman, there are 3 phase required, which are :

- Initialization
  Two sequences (e.g. sequence A and sequence B) involve in comparison are arranged in a matrix form with A+1 columns and B+1 rows. The values in the first row and first column are set to zero.
- Matrix Fill
  To fill the value inside the matrix for linear gap penalties, equation 1 is utilized. In the equation, i and j describes row and columns, F shows the matrix value of the required cell, s is the score of the required cell (i, j), gap alignment is shown on d.

$$F_{i,j} = Maximum[F_{i-1,j-1} + s_{i,j}, F_{i,j-1} + d, F_{i-1,j} + d, 0] \quad (1)$$

  Meantime, To fill the value inside the matrix for affine gap penalties, equation 2, 3, and 4 are utilized. In those equation, i and j describes row and columns, F shows the matrix value of the required cell, s is the score of the required cell (i, j), gap opening penalty is shown as d, and gap extension penalty is shown as e.

$$F_{i,j} = Maximum[F_{i-1,j-1} + s_{i,j}, Ix_{i,j}, Iy_{i,j}, 0] \quad (2)$$

$$Ix_{i,j} = Maximum[F_{i-1,j} + d, F_{i-1,j} + e] \quad (3)$$

$$Iy_{i,j} = Maximum[F_{i,j-1} + d, F_{i,j-1} + e] \quad (4)$$

- Traceback
  Traceback is useful to find out the maximum score obtained in the entire matrix for the local alignment of the sequences. Traceback start at a position which has the highest value, pointing back with the pointers, thus find out the possible predecessor, then move to next predecessor and continue until score 0 is reached. On this lab assignment, traceback is not implemented since the purpose of built program is just to have the search tool return maximum Smith-Waterman scores, not the actual alignments[4].

1.1.4 Database Sequences

In this lab assignments, protein alignments sequences (.fasta) was acquired from [9]. This protein sequences is the latest version of Swiss-Prot and contain 552265 sequences in it. To improve the performance of protein database, a conversion was done using the dbconv utility inside the CUDA folder, resulting a database file (out.gpudb) and description file (out.gpudb.descs).

1.2 OpenCL

OpenCL™ (Open Computing Language) is an open parallel programming framework which available to cross-platform of various computing environment [2]. Using OpenCL has a main advantage to improve the speed and responsiveness of applications, ranging from gaming and entertainment to scientific and medical software.

When developing software using OpenCL, the following two tools are required:

- OpenCL Compiler
  In order to execute a code, it must first be compiled to a binary using the OpenCL Compiler designed for specific environment.
- OpenCL Runtime Library
  OpenCL Runtime Library contains all OpenCL specification and tools that will be used to run an OpenCL based application.

In the lab assignment 3, OpenCL was chosen as the programming environment to implement Smith-Waterman protein alignment algorithm [1].

1.2.1 Device and Host

In the OpenCL paradigm, host means a general purpose CPU and device usually means a GPU. For example, a host program means an outer control logic that performs the configuration for a GPU-based application. An OpenCL program also contains one or more kernel functions that are designed for parallel execution on the device (GPU).

1.2.2 Memory in OpenCL

One of important aspects of OpenCL that need an extra attention is memory structure since OpenCL uses memory in a different way than conventional programs do [8]. Memory utilization is a key aspect on gaining a high and effective performance because memory usually becomes the bottleneck on running an application.

Figure 1 illustrates the memory spaces in an OpenCL system. Below is several types of memory that is shown in the mentioned figure :

- Private memory
  Private memory is an exclusive memory that is accessible by one work item.
- Local memory
  Local memory is a shared memory between work items within a workgroup. Local memory is useful if multiple work item in a group needs to access a particular chunk of global memory. Utilization of local memory is favorable since it required much less time to access local memory than to access global memory.
- Global memory
  Global memory is the biggest memory available which is accessible to all work item.
- Constant memory
  Constant memory is a specialized section of global memory used for unchangeable data throughout the execution of kernel. It is faster to access constant memory than global memory.

1.3 Previous Implementation

There are two Smith-Waterman implementation that was done in C and CUDA environment. On both implementation, a deep analysis was done to understand the

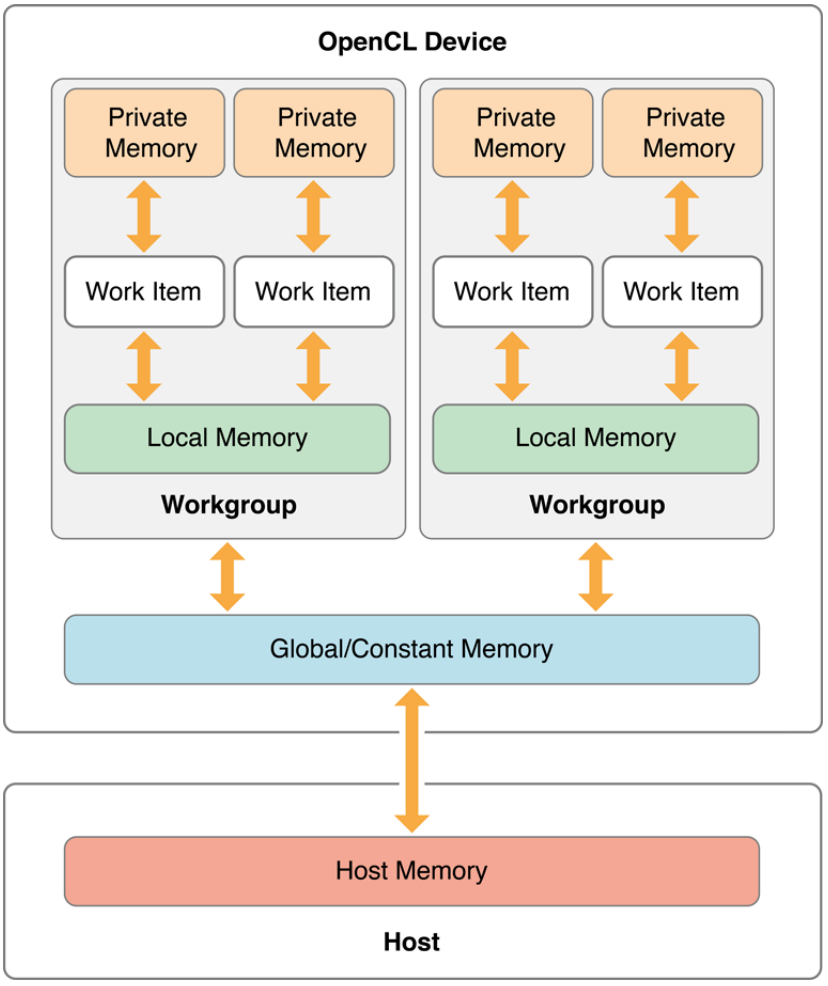

Figure 1: Physical Memory of an OpenCL System [8]

details of Smith-Waterman algorithm. Time measurement of previous implementation was also done to check the efficiency of currently built OpenCL version.

2 Implementation

Since a CUDA version is available, an OpenCL implementation can be done by following the workflow of CUDA version with some changes in syntax and several adjustment of feature which available in CUDA but not in OpenCL.

A diagram of host side implementation on OpenCL can be seen on figure 2, and kernel side implementation can be seen on figure 3. As shown on figure 2, on host side implementation on OpenCL, there are several differences compared to host side implementation on CUDA. Those differences can be summed up : initializing platform, initializing device, creating a context, creating a command queue, creating a program, compiling a program and creating kernel had to be explicitly programmed in OpenCL. Meanwhile in kernel side implementation, both kernel side implementation in OpenCL and CUDA[4] diagram has similar work flow.

In this lab, the sequence used for algorithm testing is the first sequence of fasta file or 001R_FRG3G Putative transcription facto. But other sequences are also successfully implemented. It can tested by changing the *getSequenceLength* and *getSequence*.

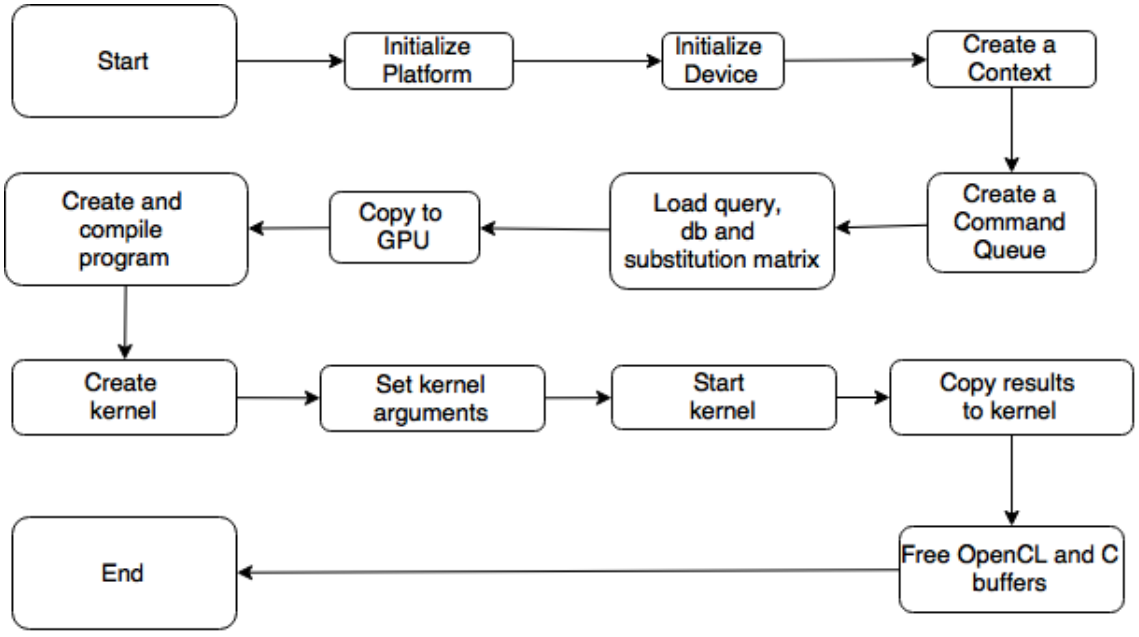

Figure 2: Host Side Implementation

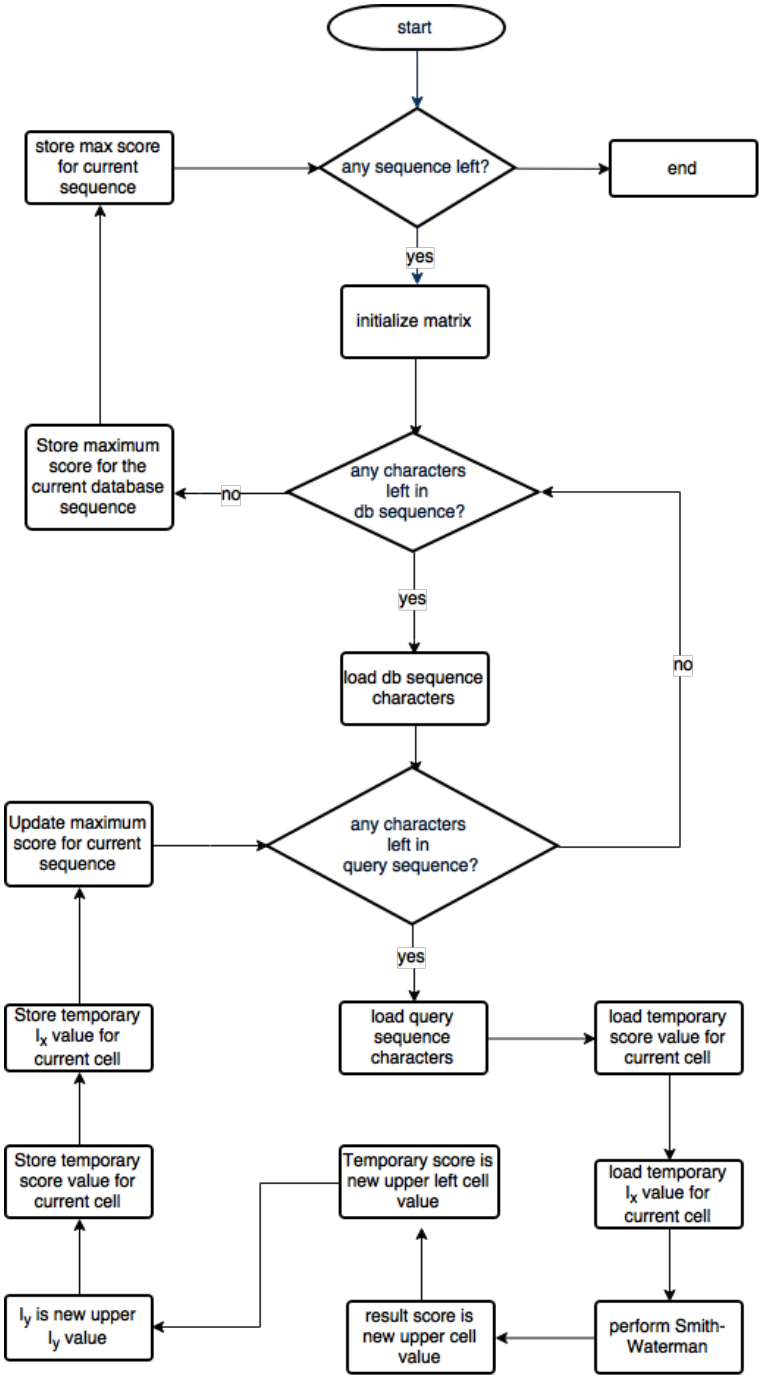

Figure 3: Kernel Side Implementation

2.1 Porting CUDA to OpenCL

Distinguished porting steps done on this implementation is explained below.

- In CUDA implementation, block and thread allocated are 20 (computing units * 4) and 128 (Maximum thread per block / 8) respectively. Meanwhile in OpenCL, number of CUDA blocks is equivalent with number of OpenCL work groups and number of threads per block is equivalent with number of OpenCL work items . Thus, in OpenCL implementation the *globalWorkSize* which defines the global workspace is set by multiplying number of CUDA blocks (work groups) and number of CUDA thread per block (work items), and *localWorkSize* which defines the local workspace is set by assigning the number of CUDA threads per block.
- Because OpenCL kernel can not receive pointer as its argument, each database data (*block offsets*, *sequence number*, and *seqeunces*) is allocated on buffer of memory object and set as kernel arguments. *Block offsets* are allocated as much as number of blocks on database. *Sequence number* arrays are allocated as much as number of database blocks multiplied by total group making up each database blocks (16). *Sequence* is allocated based on whole database file size reduced by the *block offsets* and *sequence number* size.
- *Query profile* is also copied into texture memory. In OpenCL, texture memory can be mimicked by Image 2D object. To access the value of Image 2D object, *read_imagei* built in function is used. The way accessing the value is relatively same with CUDA *tex2D* function by specifying the coordinate x and y.
- OpenCL C99 did not support multiple *__align__* data type, for example *TempData2*. So the datatype*n* (datatype: int, float, etc. n defines total number ov vector) is used. For example, instead of using *TempData2*, *scoreType8* is used and elements of its are access using *.s0*, *.s1*, *.s2*, etc or *.x*, *.y*, *.z* and *.w*.
- Because in OpenCl C99, casting pointer with different data type and pointer are not permitted, built-in function *vloadn* is used store eight value of *sequences* array. Meanwhile, the code in CUDA is *s = *(seqType8*)sequence*.
- Built-in function *mul24* is also available in OpenCL. So it is still used for fast multiplication for 24 bits integer.
- Due to the nature of OpenCL C99, passing by reference using **&** as parameter of non-kernel function is not permitted (for example *int &maxScore*). Thus, passing by reference using ***** is used, though it will requires another ***** to get the value of the pointer inside the function.
- For non-kernel function, its argument will be on *private* address. So, every variable with address space qualifier other than *private* need its qualifier to be written again in non-kernel function's argument declaration.

- To measure the total execution time of kernel, OpenCL function called *clGetEvent-ProfilingInfo* is used. *CL_PROFILING_COMMAND_START* flag is set to get the time when kernel starts executing and *CL_PROFILING_COMMAND_END* flag is set to get the time when kernel execution ends. Then time when kernel end is reduced by the time when kernel starts to get the kernel's execution time.

2.2 Bottleneck

Measurement	**CUDA Kernel Execution Time (s)**	**CUDA Memory Copy Host to Device (s)**
First	1.7	0.03
Second	1.69	0.04
Third	1.69	0.03
Average	1.693	0.033

Table 1: Profiling on CUDA Protein Alignment Application

Measurement	**OpenCL Kernel Execution Time (s)**	**OpenCL Memory Copy Host to Device (s)**
First	2.048	0.0255
Second	2.052	0.0257
Third	2.041	0.0258
Average	2.030	0.0257

Table 2: Profiling on OpenCL Protein Alignment Application

Tabel 1 and 2 shows only the execution time of each kernel and memory copy from host to device. It only need those to because the optimization of load file initialization (database, sequence, etc.) is not part of the work. And Memory copy from device to host only requires a relatively less time because it only copies *scores* array.

Based on the time measurement in tabel 1 and tabel 2, , it can be concluded that the kernel is the bottleneck of protein alignment application. Meanwhile, the data transfer between host and device is relatively faster. Hence in this lab, the kernel of OpenCL implementation is focused as optimization problem.

2.3 Optimization

2.3.1 Change the *TempColums* Address Space Qualifier

In CUDA implementation, it can be observed that the *TempColums* buffer does not hold any value from host and also does not output any value from device. It only holds value resulted by computation in kernel. So, intuitively, *TempColums* buffer is unique for each work items. Then the *TempColums* buffer address space qualifier is set to *__private*. Then it is allocated only 1024. Then the line code *TempColums[0] += gridDim.x*blockDim.x* also is replaced by *TempColums[0]++* because the allocation of its buffer is not as many as its previous implementation.

This step does not improve a significant speed up, but it is required to implement the next step.

2.3.2 Adding More Work Items

In this step, maximizing the parallelism in kernel is used. By setting the *globalWorkSize* 20*128, the total work item will be 2560. Even though, the total blocks in the database is 5792. So by multiplying the *globalWorksize* and *localWorkSize* 2 times, 5120 work items can be obtained. The results is shown by tabel 3.

Measurement	**OpenCL Kernel Execution Time (s)**
First	1.9303
Second	1.9276
Third	1.9154
Average	1.9244

Table 3: Optimization Results by Adding More Work Items

In this step, the improvement obtined is approximately 100 ms compared to the first OpenCL implementation.

2.3.3 Change the Data Type of Computation Variables Inside Kernel

Every local variables computed insider kernel are changed to single precision floating point or *float*, either *char*, *int* or *short*. And vector data type is also changed to *floatn*. For example *int4 ixLeft* becomes *float4 ixLeft*. The built-in function *mul24* is

also not used, because the inputs need to be integer. Instead, regular floating-point multiplication operator is used.

The exception happens only on global variables, such as *blockOffsetType*, *seqNums*, *sequences*, *queryProfile*, and *scores*. Then when those global variables are operated with other *float* variables, they will be casted. For example, *scores* will use *convert_ushort_sat_rte* to get value from its float counter part and *queryProfile* use *convert_float4_rtp* to load its data to a *float4* variable.

Measurement	**OpenCL Kernel Execution Time (s)**
First	1.573
Second	1.580
Third	1.575
Average	1.576

Table 4: Optimization Results by Single Precision Floating Points Operations

In this step, the improvement obtained is approximately 400 ms compared to the previous OpenCL implementation.

2.3.4 Change the floating-point multiplication with Integer mul24 Built-in Function

Fortunately, integer multiplication using built-in function *mul24* provides fast computation compared to regular floating-point multiplication, although it needs additional process, i.e. conversion from integer to float using *convert_float_rtp* and conversion from float to integer using *convert_ushort_sat_rte*. There are also several changes to variable's type, such as *tempColoumn* becoming *ushort4*. Improvement regarding this step is shown by table 5. This step can improve about 200 ms compared previous step.

Measurement	OpenCL Kernel Execution Time (s)
First	1.370
Second	1.365
Third	1.371
Average	1.369

Table 5: Optimization Results by replacing floating-point multiplication with mul24 Function

2.4 Verification

To verify OpenCL implementation, several testing were done on both CUDA and OpenCL implementation. One of the testing result is available on figure 4 and figure 5. Both result were tested using sequence Q6GZX4 which has sequence length 256. From both figures, it can be seen that both protein alignment produce the same results for its top 20 scores.

```
Protein Alignment CUDA Implementation
Sorting results...
Results:
  0. sp|Q6GZX4|001R_FRG3G Putative transcription factor  SCORE: 1361
  1. sp|Q91FP2|VF282_IIV6 Putative transcription factor  SCORE: 251
  2. sp|Q196Y1|VF282_IIV3 Putative transcription factor  SCORE: 203
  3. sp|A4XL64|PNP_CALS8 Polyribonucleotide nucleotidyl  SCORE: 80
  4. sp|Q98544|Y494R_PBCV1 Putative transcription facto  SCORE: 74
  5. sp|P42271|TBG2_DROME Tubulin gamma-2 chain OS=Dros  SCORE: 72
  6. sp|A6Q2V7|TRMA_NITSB tRNA/tmRNA (uracil-C(5))-meth  SCORE: 72
  7. sp|B9MR54|PNP_CALBD Polyribonucleotide nucleotidyl  SCORE: 69
  8. sp|P29357|HSP7E_SPIOL Chloroplast envelope membran  SCORE: 69
  9. sp|A7IXI8|Y663R_PBCVN Putative transcription facto  SCORE: 67
 10. sp|Q7NAR6|ENGB_MYCGA Probable GTP-binding protein   SCORE: 66
 11. sp|Q9QXZ0|MACF1_MOUSE Microtubule-actin cross-link  SCORE: 66
 12. sp|Q2VZQ4|PNP_MAGSA Polyribonucleotide nucleotidyl  SCORE: 66
 13. sp|A4G0R3|PYRB_METM5 Aspartate carbamoyltransferas  SCORE: 66
 14. sp|P30773|T2X1_XANCC Type-2 restriction enzyme Xcy  SCORE: 66
 15. sp|P37374|CADF_STAAU Cadmium resistance transcript  SCORE: 66
 16. sp|Q9CEF5|PKNB_LACLA Probable serine/threonine-pro  SCORE: 65
 17. sp|Q6ZQQ6|WDR87_HUMAN WD repeat-containing protein  SCORE: 65
 18. sp|P33144|BIMB_EMENI Separin OS=Emericella nidulan  SCORE: 64
 19. sp|Q53EZ4|CEP55_HUMAN Centrosomal protein of 55 kD  SCORE: 64
```

Figure 4: Top 20 Sequence Scores in CUDA Protein Alignment Application

Another checking procedure was done by using a website [10] which able to compare all results from CUDA and OpenCL implementation. In this comparison, all results from both implementation was shown to be exactly similar. Thus, protein alignment is successfully implemented in OpenCL.

```
Protein Alignment OpenCL Implementation
Sorting results...
Results:
  0. sp|Q6GZX4|001R_FRG3G Putative transcription factor  SCORE: 1361
  1. sp|Q91FP2|VF282_IIV6 Putative transcription factor  SCORE: 251
  2. sp|Q196Y1|VF282_IIV3 Putative transcription factor  SCORE: 203
  3. sp|A4XL64|PNP_CALS8 Polyribonucleotide nucleotidyl  SCORE: 80
  4. sp|Q98544|Y494R_PBCV1 Putative transcription facto  SCORE: 74
  5. sp|P42271|TBG2_DROME Tubulin gamma-2 chain OS=Dros  SCORE: 72
  6. sp|A6Q2V7|TRMA_NITSB tRNA/tmRNA (uracil-C(5))-meth  SCORE: 72
  7. sp|B9MR54|PNP_CALBD Polyribonucleotide nucleotidyl  SCORE: 69
  8. sp|P29357|HSP7E_SPIOL Chloroplast envelope membran  SCORE: 69
  9. sp|A7IXI8|Y663R_PBCVN Putative transcription facto  SCORE: 67
 10. sp|Q7NAR6|ENGB_MYCGA Probable GTP-binding protein   SCORE: 66
 11. sp|Q9QXZ0|MACF1_MOUSE Microtubule-actin cross-link  SCORE: 66
 12. sp|Q2VZQ4|PNP_MAGSA Polyribonucleotide nucleotidyl  SCORE: 66
 13. sp|A4G0R3|PYRB_METM5 Aspartate carbamoyltransferas  SCORE: 66
 14. sp|P30773|T2X1_XANCC Type-2 restriction enzyme Xcy  SCORE: 66
 15. sp|P37374|CADF_STAAU Cadmium resistance transcript  SCORE: 66
 16. sp|Q9CEF5|PKNB_LACLA Probable serine/threonine-pro  SCORE: 65
 17. sp|Q6ZQQ6|WDR87_HUMAN WD repeat-containing protein  SCORE: 65
 18. sp|P33144|BIMB_EMENI Separin OS=Emericella nidulan  SCORE: 64
 19. sp|Q53EZ4|CEP55_HUMAN Centrosomal protein of 55 kD  SCORE: 64
```

Figure 5: Top 20 Sequence Scores in OpenCL Protein Alignment Application

3 Final Results

3.1 Environment Sets

In designing OpenCL application, there are several factors needed to be considered. In this lab session, the type of GPU used in running the application is GeForce GTX 750 Ti. This device has several specific characteristics, which are

- Maximum work item per work group: 1024
- Global memory: 2047 Mb
- Shared memory: 48 kb
- Constant memory: 64 kb
- Block registers: 65536
- OpenCL Version : 1.2
- Parallel Computing Units : 5

3.2 Profiling

In this subsection, the chosen sequence to compare the implementation of CPU, CUDA and OpenCL is sequence Q6GZX4. This sequence length is 256.

During measurements, running a CPU implementation on an enormous database required a lengthy amount of time and also prone to error (sometimes it detected error after running for several hours). Thus, it was decided to cut the sequence to 257852 sequences from and use function 5 to count approximation time if it runs on full sequences.

$$Time_{full\ sequence} = \frac{Sequences_{full\ sequence}}{Sequences_{cut\ sequence}} * Time_{cut\ sequence} \tag{5}$$

Results related to CPU implementation can be found on table 6 and table 7.

In CUDA and OpenCL implementation, total time is counted using equation 6. Results related to CUDA implementation on searching sequence Putative transcription factor can be found on table 8 and table 9. OpenCL implementation results is available on table 10 and table 11.

$$Total\ Time = Initialization + Load\ Kernel + Mem\ Copy\ to\ Device + Run\ Kernel \quad (6)$$

Measurement	**Time on Cut Sequence(s)**	**Time on Full Sequence(s)**
First	8153.49	17463.07
Second	8104.74	17358.66
Third	8107.42	17364.40
Average	8121.88	17395.37

Table 6: CPU Implementation Time Results

Measurement	**GCUPS**
First	0.003222
Second	0.003241
Third	0.003240
Average	0.003234

Table 7: CPU Implementation GCUPS Results

Measurement	**Initialization**	**Load Kernel(s)**	**Run Kernel(s)**	**Total**
First	0.26	0.03	1.68	1.97
Second	0.26	0.03	1.69	1.98
Third	0.26	0.03	1.69	1.98
Average	**0.26**	**0.03**	**1.69**	**1.98**

Table 8: CUDA Implementation Time Results

Measurement	**GCUPS**
First	30.08
Second	29.91
Third	29.91
Average	**29.96**

Table 9: CUDA Implementation GCUPS Results

Measurement	Initialization	Load Kernel(s)	Run Kernel(s)	Total
First	0.54	0.17	1.36	2.07
Second	0.55	0.17	1.36	2.08
Third	0.54	0.16	1.36	2.07
Average	**0.54**	**0.17**	**1.36**	**2.07**

Table 10: OpenCL Implementation Time Results

Measurement	GCUPS
First	37.16
Second	37.16
Third	37.16
Average	**37.16**

Table 11: OpenCL Implementation GCUPS Results

3.3 Effectiveness Checking

To check the effectiveness of program, speed up comparison on searching sequence Putative transcription factor was done. To count speed up between OpenCL and sequential implementation, equation 7 is utilized. Speed up comparison on kernel time between OpenCL and CUDA uses equation 8.

$$S_{Implementation\ 1\ to\ 2} = \frac{Total\ Time\ on\ Implementation_1}{Total\ Time\ on\ Implementation_2} \tag{7}$$

$$S_{Kernel} = \frac{Kernel\ Execution\ Time\ on\ CUDA}{Kernel\ Execution\ Time\ on\ OpenCL} \tag{8}$$

3.3.1 Comparison between Sequential & OpenCL Implementation

Comparison between sequential and OpenCL implementation on searching sequence Q6GZX4 can be found in table 12. Based on the comparison, it can be found that using OpenCL utilization able to make the program run much faster compared to sequential implementation, shown by an average speed up 1672.74131.

Measurment	Total Time in Sequential (s)	Total Time in OpenCL (s)	SpeedUp
First	17463.07	10.3789	1682.558581
Second	17358.66	10.3986	1669.324515
Third	17364.40	10.4207	1666.340834
Average	17395.37	10.3994	1672.74131

Table 12: Speed Up OpenCL Compared to Sequential

3.3.2 Comparison between CUDA & OpenCL Implementation

As shown by table 8 and 10, the CUDA implementation achieves a faster total time than the OpenCL version even though run kernel time in OpenCL is quicker. This result is caused of CUDA required less time in initialization and load kernel than OpenCL, thus it can neglect CUDA disadvantage of running longer while running a kernel.

To provide a better comparison between OpenCL and CUDA implementation on run kernel time, another testing was done using additional 14 sequences, which made the total sequence used for testing was 15 sequences. Below are the mentioned sequences :

1. B3EWH9, sequence length : 60
2. P0C9F4, sequence length : 141
3. Q6GZV5, sequence length : 148
4. Q6GZX4, sequence length : 256
5. P13748, sequence length : 365
6. Q621J7, sequence length : 709
7. Q6GZV6, sequence length : 851
8. P29815, sequence length : 1003
9. Q6GZT5, sequence length : 1165
10. Q5SSH7, sequence length : 2924
11. O43149, sequence length : 2961
12. P28167, sequence length : 3005
13. P20930, sequence length : 4061

14. Q9UKN1, sequence length : 5478

The result of this comparison is available on table 13. To provide a better insight of this result, graphs showing a comparison between CUDA implementation and OpenCL implementation on performance, kernel time and speed up are also provided. These three graphs are available on figure 6, figure 7, and figure 8.

Name	**Kernel Time (s)**		**Performance (GCUPS)**		**Speed Up**
	OpenCL	**CUDA**	**OpenCL**	**CUDA**	
B3EWH9(60)	0.33	0.37	35.39	32.01	1.11
P0C9F4(141)	0.78	0.86	35.87	32.37	1.11
Q6GZV5(148)	0.80	0.88	36.68	33.20	1.10
Q6GZX4(256)	1.36	1.68	37.26	30.08	1.24
P13748(365)	1.95	2.41	37.05	29.90	1.24
Q621J7(709)	3.77	4.76	37.09	29.41	1.26
Q6GZV6(851)	4.62	5.73	36.37	29.32	1.24
P29815(1003)	5.48	6.76	36.11	29.29	1.23
Q6GZT5(1165)	6.38	7.89	36.04	29.15	1.24
Q5SSH7(2924)	15.93	19.72	36.23	29.27	1.24
O43149(2961)	16.16	19.98	36.18	29.26	1.24
P28167(3005)	16.40	20.28	36.18	29.25	1.24
P20930(4061)	22.15	27.49	36.20	29.16	1.24
Q9UKN1(5478)	29.85	37.15	36.22	29.11	1.24

Table 13: Result of CUDA and OpenCL implementation on 15 different sequences

From table 13, it can be seen that OpenCL shows a faster implementation than CUDA version. The fastest speed up is achieved when executing sequence Q621J7 which shows speed up 1.26x and the slowest speed up is shown when checking sequence Q6GZV5 (1.10x). On average, OpenCL implementation is able to reach speed up 1.21x compared to CUDA implementation.

OpenCL implementation shows a lower speed up when running a sequence with small size because it allocate a lot of resources, indicated by number of globalWorkSize and localWorkSize which is twice as large as the size of blockDim in CUDA version, makes it harder to distribute the work load or allocated resources is not fully utilized compared to when it run a larger sequence. In fact, when running sequence large

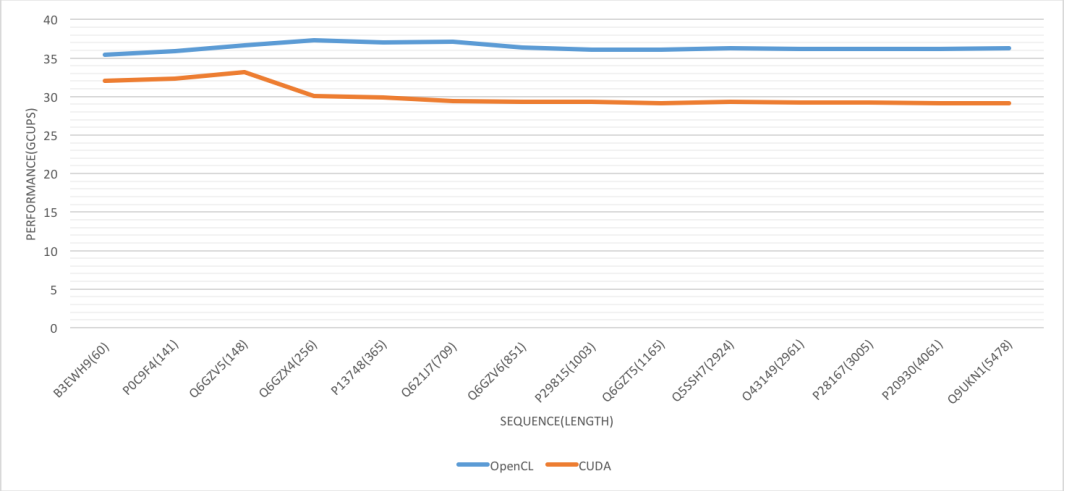

Figure 6: Performance comparison between OpenCL implementation and CUDA implementation

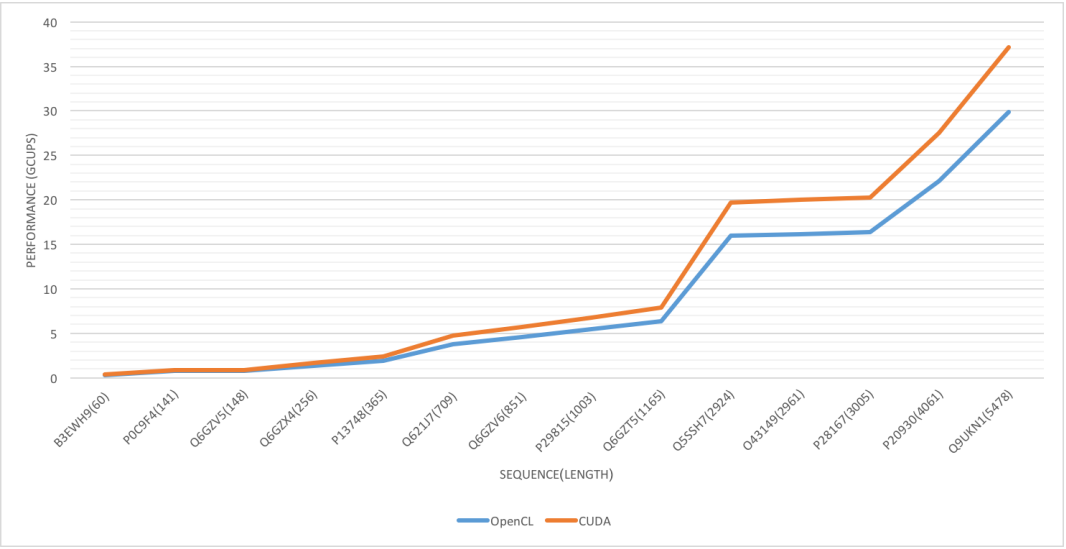

Figure 7: Kernel time comparison between OpenCL implementation and CUDA implementation

enough (approximately the sequence length is more than 200), it achieves a stable speed up 1.24x compared to CUDA implementation.

Meanwhile, figure 6 shows that our optimized OpenCL has higher computational speed (GCUP/s) than previous CUDA implementation in all tested sequences. figure 7 shows that our optimized OpenCL can achieve faster kernel execution compared to the CUDA one, although in short-length sequence both implementation has roughly

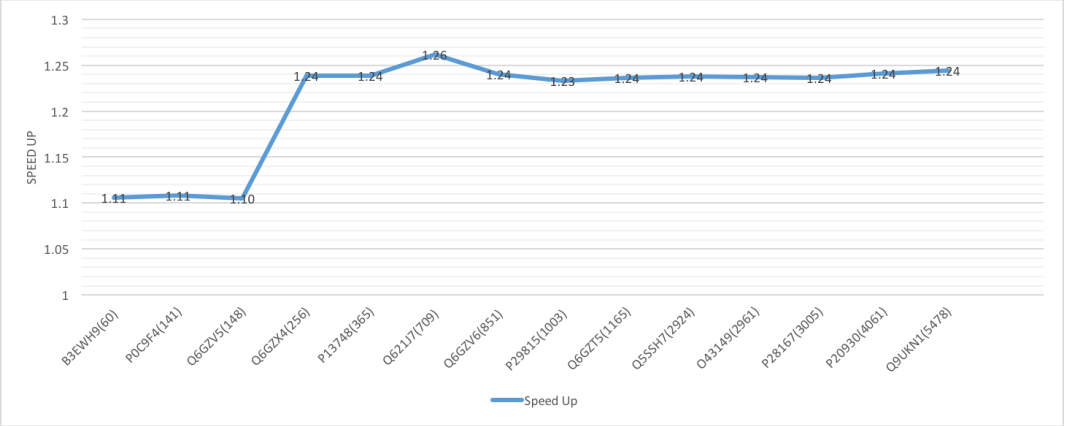

Figure 8: Speed up of OpenCL implementation compared to CUDA implementation

same execution time, as explained before. Then figure 8 shows that our optimized OpenCL implementation achieve stable speed up (around 1.24) when the sequence length is more than 200 and it has lower speed up when the sequence length is relatively short or below 200.

Based on several tables above, our optimized OpenCL implementation shows a faster speed up and performance compared to previous CUDA implementation. There are several factors that contribute to this result. Below these factors affecting the result are explained.

- GPU which was used in testing are optimized to perform FP32 type workloads, and integer workloads will either have to be emulated (using FP16) or done in a side pipeline. Thus floating point operation was preferred, like in OpenCL implementation.
- Number of globalWorkSize and localWorkSize in OpenCL is twice as large as the size of blockDim in CUDA, so kernel in OpenCL can execute faster.
- Using optimized built-in function in OpenCL kernel, such as *mul24*, can improve performance relatively significant.

4 Conclusion

1. OpenCL is a cross-platform parallel programming framework. Applying OpenCL has a main advantage to improve the speed and responsiveness of various applications, including protein alignment using Smith-Waterman Algorithm.
2. Parallel programming using CUDA and OpenCL has its own trade-off. CUDA provides more friendly language and interface, but it can only be applied on NVIDIA devices. While OpenCL can be applied to various devices or platform with different vendor, but its language is not as convenient as CUDA.

References

[1] Delft University of Technology. (2016). *Course EE4C07 Lab Manual Advanced Computing Systems*. Delft.

[2] https://www.khronos.org/opencl/. Retrieved 5 November 2016.

[3] ftp://ftp:ebi:ac:uk/pub/databases/uniprot/knowledgebase/

[4] Biological Sequence Alignment Using Graphics Processing Units. M.A. Kentie. 2010. TU Delft.

[5] Mount DM. (2004). Bioinformatics: Sequence and Genome Analysis (2nd ed.). Cold Spring Harbor Laboratory Press: Cold Spring Harbor, NY. ISBN 0-87969-608-7.

[6] Munshi, Aaftab (2011). The OpenCL Specification Version 1.2 (Rev.19). The Khronos Group Inc.

[7] http://vlab.amrita.edu/?sub=3&brch=274&sim=1433&cnt=1. Retrieved 10 November 2016

[8] https://developer.apple.com/library/content/documentation/Performance/Conceptual/OpenCL Retrieved 11 November 2016.

[9] ftp://ftp:ebi:ac:uk/pub/databases/uniprot/knowledgebase/uniprot_sprot:fasta:gz. Retrieved 23 October 2016.

[10] https://www.diffchecker.com/diff. Retrieved 10 November 2016.