Performance comparison of GPU programming frameworks with the striped Smith-Waterman algorithm

Takeshi Kakimoto, Keisuke Dohi, Yuichiro Shibata, Kiyoshi Oguri
Department of Science and Technology
Nagasaki University, Japan
{kaki,dohi}@pca.cis.nagasaki-u.ac.jp
{shibata, oguri}@cis.nagasaki-u.ac.jp

ABSTRACT

This paper evaluates and discusses how different GPU programming frameworks affect the performance obtained from GPU acceleration of the striped smith-waterman algorithm used for biological sequence alignment. A total of 6 GPU implementations of the algorithm on NVIDIA GT200b and AMD RV870 using the CUDA and the OpenCL frameworks are compared to analyze cons and pros of explicit descriptions for architecture specific hardware mechanisms in the code. The evaluation results show that the primitive descriptions with the CUDA are still efficient especially for small size data, while better instruction scheduling and optimizations are carried out by the OpenCL compiler. On the other hand, the combination of OpenCL and RV870 which provides a relatively simple view of the architecture is efficient for the large data size.

1. INTRODUCTION

Graphics processing units (GPUs) have received increasing attention as a highly-efficient and relatively low-cost acceleration platform in a wide range of application domains. Technologies related to GPUs have also rapidly progressed in past years, yielding many variants of GPU architectures and frameworks for GPU programming.

At present, GPU programming frameworks may be classified into two categories. One is dedicated for a specific series of GPU architectures as represented by CUDA by NVIDIA. With these frameworks, low-level hardware mechanisms which highly depend on target architectures can be explicitly utilized to extract the maximum potential performance, while an intimate understanding of compatibility between algorithms and architectures is necessary. The other intends to cover wider varieties of architectures beyond GPUs like OpenCL. For these frameworks, some architecture specific mechanisms are not available in compensation for retaining inter-architecture portability of the code.

So far, a lot of comparative study between GPU and other platforms such as many-core processors and FPGAs have been carried out aiming at demonstrating cons and pros of GPUs[1][2]. However, there have been few reports focused on differences in GPU programming frameworks and thus there have not been any concrete guidelines for selecting an appropriate programming framework yet.

In this paper, we discuss how different GPU programming frameworks affect the performance obtained from GPU acceleration in a quantitative way. We compare the implementations of the striped Smith-Waterman (SSW) biological pairwise sequence algorithm on different GPU core architectures (NVIDIA GT200b and AMD RV870) and with different programming frameworks (CUDA and OpenCL).

The rest of this paper is organized as follows. Section 2 describes the SSW algorithm and Section 3 illustrates the differences among programming frameworks and GPU architectures we evaluated. The results of the performance comparison are presented and analyzed in Section 4. Finally, this paper is concluded in Section 5.

2. BACKGROUND

2.1 The Smith Waterman Algorithm

The Smith-Waterman (SW) algorithm is a dynamic programming algorithm which finds the best local fragment between two biological sequences [3][4]. An application of the algorithm is to find the closest sequence(s) from a large sequence database for given query sequence. The search for the optimal local alignment consists of two stages. Firstly, an alignment matrix of two biological sequences (e.g. protein, DNA) is calculated and results in a matching score for two sequences. After that, the alignment is traced back from the maximum score on the alignment matrix until a zero element is found.

More specifically, let $D = d_0d_1...d_{m-1}$ denotes a database sequence of length m. Let $Q = q_0q_1...q_{n-1}$ denotes a query sequence of length n. Let $W(q_i, d_j)$ denotes the substitution scoring matrix[5] , which gives a score describing the likelihood of substitution between characters q_i and d_j. Let G_{init} and G_{ext} denote penalties for opening a new gap, and continuing an existing gap respectively.

With the above, the alignment matrix computation of the SW algorithm is described by the following equations:

$$E_{i,j} = \max\begin{cases} H_{i,j-1} - G_{init} \\ E_{i,j-1} - G_{ext} \end{cases} \tag{1}$$

$$F_{i,j} = \max\begin{cases} H_{i-1,j} - G_{init} \\ F_{i-1,j} - G_{ext} \end{cases} \tag{2}$$

$$H_{i,j} = \max\begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(q_i, d_j) \end{cases} \tag{3}$$

The values of $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are defined as 0 if $i < 0$ or $j < 0$. From these equations, we observe that the value of $H_{i,j}$ depends on the values of its upper neighbour $H_{i,j-1}$, left neighbour $H_{i-1,j}$ and left-upper neighbour $H_{i-1,j-1}$, as shown in Figure 1.

The above operations are massively parallelisable since the anti diagonal elements of the alignment matrix are independent of each

This work was presented in part at the first international workshop on Highly Efficient Accelerators and Reconfigurable Technologies (HEART2012), Naha, Okinawa, Japan, May 31 June 1, 2012.

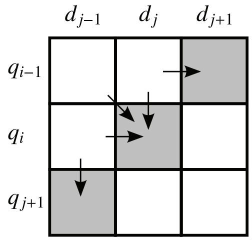

Figure 1: Data dependency of the SW dynamic programming algorithm. Arrows indicate dependency for the lower right element.

other (as labeled with the dot pattern in Figure 1), and hence can be computed in parallel. In addition, the computation of different alignment matrices between a query sequence and different subject sequences can also be done in parallel. Since GPUs have the ability of allocate thousands of parallel threads to a particular task, it is a very appealing acceleration platform for the SW algorithm[6][7][8][9].

The algorithm is done many times proportional to the number of sequences in a database, which means the application has large parallelism. Note that the trace back stage procedure will be done only for one or few subject sequences with the highest scores. Thus, we only focus on GPU implementation of the first stage in which alignment matrices are calculated.

2.2 The Striped Smith Waterman Algorithm

The Striped Smith-Waterman (SSW) algorithm proposed by Farrar[10][11] is a variant of Smith-Waterman algorithm for SIMD instructions. Calculation of an alignment matrix has parallelism between elements. Some previous work use SIMD instructions to accelerate the SW algorithm exploiting this parallelism[12][13], but the matrix calculation has a data miss-alignment problem. This problem tends to lead performance degradation especially on SIMD instructions. The SSW algorithm resolves this problem by re-ordering sequence and splitting process in two parts; one for SIMD-oriented calculation and the other for else.

The SSW algorithm uses a substitution scoring matrix which depends on a query sequence called a profile. The aforementioned normal substitution scoring matrix which depends on a query sequence is an n-by-n matrix where n denotes the number of types of elements that construct sequences. On the other hand, the profile is a $|Q|$-by-n matrix where $|Q|$ denotes query sequence length. Using the profile, Eq.(3) is replaced by the following equation:

$$H_{i,j} = \max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + P(i, d_j) \end{cases} \tag{4}$$

where P denotes the profile and $P(i, d_j)$ means a substitution score between the i-th character of query sequence and the character d_j of the database sequence. Access to the profile has lower costs than access to the raw substitution scoring matrix, since the load of q_i can be eliminated. Additionally, while random access to the substitution scoring matrix is required for the SW algorithm, all the elements within the same SIMD group are continuously accessed in the SSW algorithm, which also reduces the number of load instructions.

```
1:  segLen =(length(Q)+15)/16 + 1;
2:  for i = 0 to Subject Length - 1 do
3:      s = subject[i];
4:      vF = 0;
5:      vH = vHStore[segLen-1] << 1;
6:      swap(vHLoad, vHStore);
7:      for j = 0 to segLen-1 do
8:          vH = vH + profile[s][j];
9:          vScore = max(vScore, vH);
10:         vH = max(vH, vE[j], vF);
11:         vHStore[j] = vH;
12:         vE[j] = max(vE[j] - vGext, vH - vGinit);
13:         vF = max(vF-vGext, vH-vGinit);
14:         vH = vHLoad[j];
15:     end for
16:     while AllElement(vF $\leq$ vHStore[j] - vGext - vGinit)
        do
17:         vF = vF << 1;
18:         for j=0 to setLen-1 do
19:             vHStore[j] = max(vHStore[j], vF);
20:             vF = vF - vGext;
21:         end for
22:     end while
23: end for
```

Figure 2: Striped Smith-Waterman algorithm

Table 1: Specification of GPUs.

GPU	GeForce GTX 295	Radeon HD 5970
Core	GT200b x2	RV870 x2
Core clock	1242 MHz	725 MHz
# of arithmetic unit*	240	1600
Performance*	0.89 TFLOPS	2.32 TFLOPS
Memory*	GDDR3 896 MB	GDDR5 2 GB
Memory clock	999 MHz	1000 MHz
Bandwidth*	111.9 GB/s	128 GB/s

*Performance per GPU core

Figure 2 shows the pseudo code for the SSW algorithm. Variables with prefix 'v', for example 'vH', mean SIMD registers. We assumed that all of variables were initialized by zero before the code was executed. The operator << means an SIMD element-based shift left operation. The 'AllElement' function returns a non-zero value if all of the elements within the results of SIMD instructions take a non-zero value, and returns a zero otherwise.

The clause of the line 16 to 22 in Figure 2 is called the lazy-f evaluation. This a kind of clean up code to correct the values of H and F which can be sometimes miss-calculated in the above SIMD part. The number of iterations for the lazy-f evaluation can be vary depending on the database sequence and the query sequence.

3. ARCHITECTURES AND FRAMEWORKS

We describe GPU architectures and frameworks we used. GPUs have a lot of computational elements called cores or processors. Different GPU vendors give different words to describe the computational elements like Streaming Processors or Stream Cores. In this paper, we call these computational elements "arithmetic units" regardless of they can take their own instruction flow or not, when we mention comparison of different GPU architectures.

3.1 GT200b GPU architecture

The GT200b GPU core is an NVIDIA's GPU architecture that has 240 arithmetic units called the Streaming Processors, or SPs. A group of 8 SPs constructs the Streaming Multiprocessor, or SM. The GT200b has 30 SMs in total. All the SPs within an SM share the same instruction flow, so we can see the GT200b as a 30-core multiprocessor each with 8-way SIMD instructions. Table 1 shows specification of GT200b in more detail.

Another important characteristic of the GT200b is the memory hierarchy. GT200b has a various kind of memories such as registers, shared memory, global memory, texture memory, etc. Registers are the fastest memory in GT200b and each SP has 1,024 private registers. The shared memory, which is 16KB per SM, is also relatively fast memory and shared by SPs within each SM. This memory is widely used for communication between SPs. The global memory and the texture memory are shared by all the SPs and provide a large memory space of Giga Bytes. However, they require a long latency. Unlike the global memory, the texture memory is read-only but equipped with a cache mechanism. In order to bring out the maximum potential performance, we should understand characteristics of these memories and use the memory hierarchy carefully.

3.2 RV870 GPU architecture

The RV870 GPU core is an AMD's GPU architecture that has 1,600 arithmetic units called stream cores. This GPU core has 20 SIMD engines, and each of which has 16 thread processors. Each thread processor has 5 stream cores; 4 for basic arithmetic and one for additional complex functions such as division, bit-shift, trigonometric functions. The specification of RV870 is also shown in Table 1.

RV870 has a memory hierarchy like GT200b; each thread processor has the general purpose registers and each SIMD engine has 4 texture caches. However, all the external memory interfaces are equipped with a cache mechanism in contrast to GT200b.

3.3 Programming Frameworks

Compute Unified Device Architecture (CUDA) is a development environment for GPGPU offered by NVIDIA Corporation[14]. The language specification is an extension of C/C++ with a multi-thread programming model.

At the lowest level, the concept of CUDA is parallel processing with hierarchically grouped multiple threads. The group of threads is called "Thread block" and the group of thread blocks is called "Grid". Each thread executes the same code which is described as a sequential process, called "Kernel". In CUDA, a kernel execution is called "Kernel call". Each thread has a unique ID within a grid and is able to process different data according to the ID, while they execute the same code. CUDA provides a synchronization statement where all threads within a thread block are synchronized. On the other hand, there is no statement that can synchronize all of the threads within a grid. However, we can synchronize them out of the kernel call by using CUDA API. Actual processing of kernel function goes along with a 32-thread unit called a "Warp" in an SIMD manner.

The OpenCL[15] is another programming environment for multicore platforms, including GPGPU. Although the OpenCL provides similar framework to the CUDA, words and concepts are a little different. Table 2 shows a correspondence table of processing elements and memory models between CUDA and OpenCL. Since OpenCL is intended to cover a wide range of architectures, it can handle both GT200b and RV870 while the CUDA can be used only for NVIDIA GPUs including GT200b.

Table 2: Correspondence table of processing elements and memory model.

CUDA	OpenCL
Thread	Work-item
Warp	Wavefront
Thread block	Work-group
Register	Private memory
Shared memory	Local memory
Global memory	Global memory
Texture memory	-

The important differences between CUDA and OpenCL are on the texture memory and the warp vote functions. We can use the texture memory via CUDA API, but OpenCL has no API to use the texture memory in an explicit way. Warp vote functions are special primitives provided only in the CUDA environment. We used $int\ \_\_all(int\ p)$ warp vote function. This function return a non-zero value if and only if p takes a non-zero value for all the threads within the warp. By using this warp vote function to implement the 'AllElement' in the lazy-f evaluation process, costs for synchronization and communication between threads within a warp can be effectively mitigated. Since the warp vote functions are not available for the OpenCL, a simple reduction method using the shared memory will be required in the OpenCL environment.

4. EVALUATION AND DISCUSSION

4.1 Conditions for evaluation

The cell updates per second (CUPS) is a commonly used measure for sequence alignment execution performance as it is normalized in terms of the database and query sequence sizes. Given a query sequence Q and a sequence database D, the Giga CUPS (GCUPS) is defined as below:

$$GCUPS = \frac{|Q| \times |D|}{t \times 10^9}$$

where $|Q|$, $|D|$, and t represent the length of the query sequence, the total length of the database sequences, and the elapsed time of the GPU kernels in seconds, respectively.

We used the Swiss-Prot[16] release 2012_01 as a sequence database which contains 534,242 sequence entries, comprising 19,968,487 amino acids. As a substitution matrix, we used the BLOSUM50 with $Ginit = 12$ and $Gext = 2$. Note that selection of the substitution matrixes would have little effect on performance because the algorithm looks up the profile instead of the substitution matrix.

Table 3 shows two implementation platforms we used. Note, we used only one of two GPU cores on each GPU. Considering the algorithm we used, the performances of the host processors will not have significant impact on the whole performance.

Table 4 shows configurations of each implementation we prepared. "Profile storage" means the place where profiles are stored; the global memory or the texture memory. "Reduction" shows the method for implementation of the 'AllElement' function; with the warp vote function or a simple reduction without the warp vote funtion. Conf.3 uses the warp vote function but does not use the texture memory, while Conf.4 uses the texture memory but not the warp vote function. Conf.5 is a base implementation without the texture memory nor the warp vote function. Conf.6 is an implementation on RV870 using OpenCL.

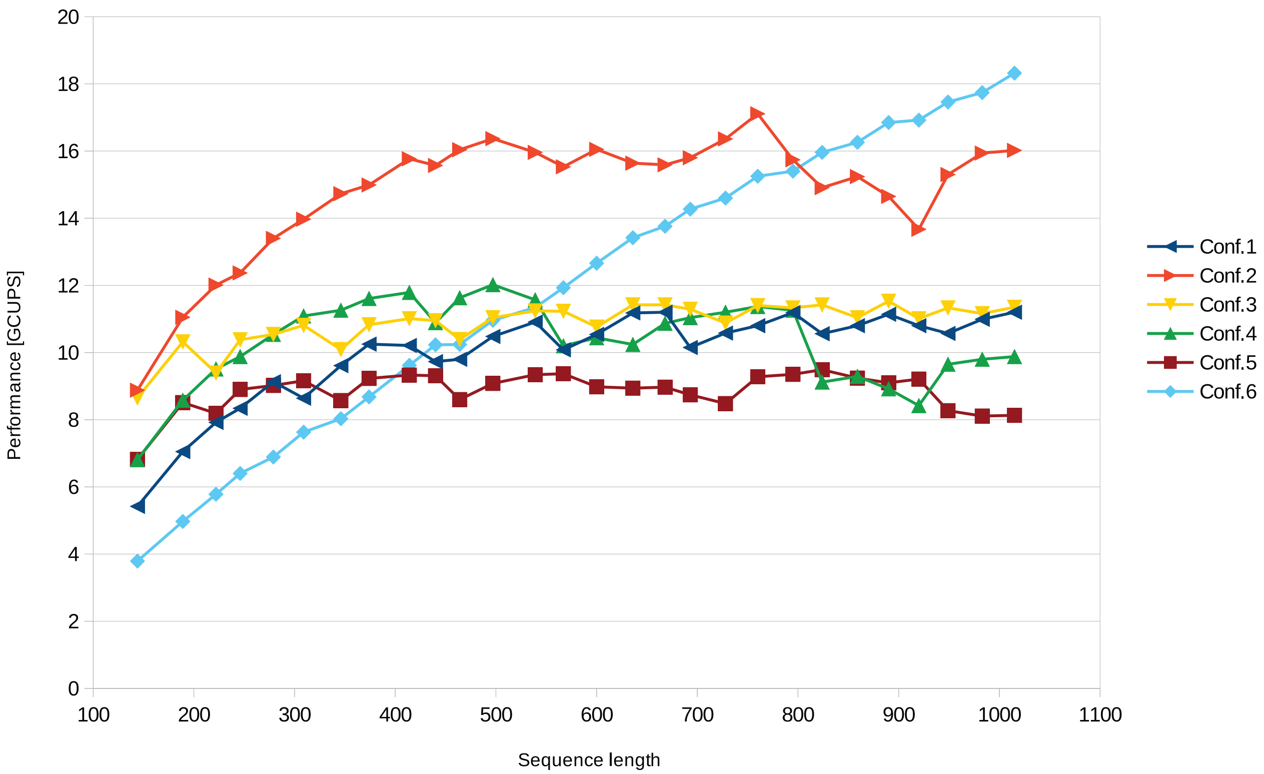

Figure 3: Performances of each implementations

Table 3: Specification of environments.

	Machine#1	Machine #2
GPU	GeForce GTX 295	Radeon HD 5970
CPU	Core i7 920	Core2 Quad Q9650
Memory	DDR3-1066 6 GB	DDR3-1333 8 GB
OS	openSUSE 11.2	openSUSE 12.1
Framework	CUDA, OpenCL	OpenCL

4.2 Performance comparison

Figure 3 shows performance evaluation results of all of the configurations. We first compared two GT200b implementations; the implementation using the OpenCL(Conf.1) and the fastest implementation using CUDA(Conf.2). Conf.2 is faster than Conf.1 for all of the sequences. The maximum performances achieved by Conf.1 and Conf.2 are approximately 11.2 GCUPS and 17.1 GCUPS, respectively. The cause of this performance gap may be twofold:

- The use of warp vote function and/or the texture memory
- The efficiency difference of the code generated by the CUDA compiler and the OpenCL compiler

To find out actual performance impacts of the two factors separately, we analyzed three additional configurations, Conf.3, Conf.4 and Conf.5. Advantages of warp vote function and texture memory can be analyzed by comparing Conf.3 with Conf.5 and Conf.4 with Conf.5, respectively. The efficiency of the CUDA compiler and the OpenCL compiler can be discussed by comparing Conf.1 with Conf.5, since these two configurations have the same architectural setup.

The result of comparison between Conf.3 and Conf.5 shows that the warp vote function improves the performance for all the sequence lengths. The maximum speed up of 39% is shown when the sequence length is 1,015 and 23% of performance improvement is observed on average. The effect of the use of the texture memory is more complex. The comparison results between Conf.4 and Conf.5 show the use of the texture memory improves the performance by up to 35% when the sequence length is shorter than around 800. However, for the lengths of 800 to 920, the negative impacts of the texture memory arise. Since Conf.2, which also utilizes the texture memory, shows similar performance degradation for these sequence lengths, this inefficiency seems to come from a cache organization of the texture memory such as associativity. Using both the warp vote function and the texture memory (Conf.2), 97% of performance improvement is shown when the sequence length is 1,015 compared to Conf.5 (67% on average). Additionally, there is no performance crossing between Conf.2 and Conf.5.

We next compared Conf.1 with Conf.5 to discuss the efficiency difference between the CUDA compiler and the OpenCL compiler, where any CUDA-unique features are not utilized. The results show Conf.5 is faster than Conf.1 when the sequence length is shorter than around 300, while Conf.5 is slower for longer sequences.

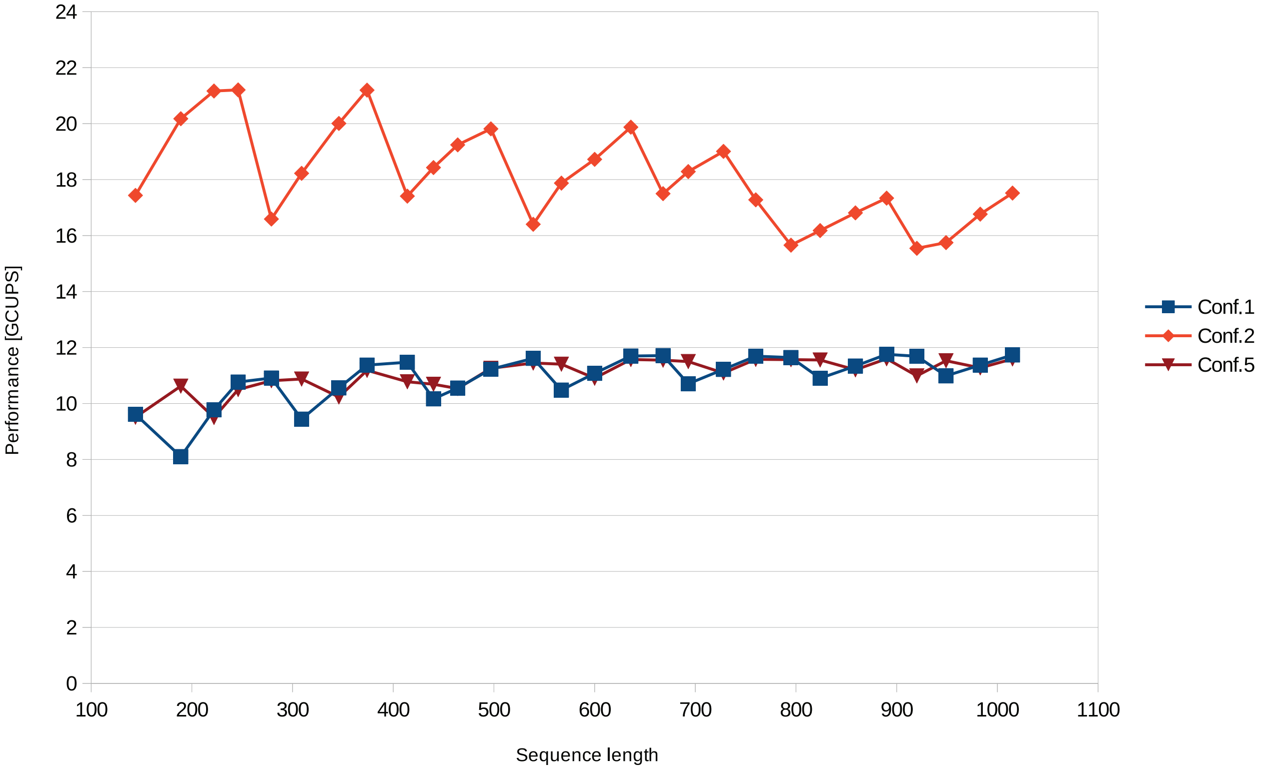

Figure 4: Performances of each implementations without the lazy-f evaluation

In other words efficiency difference between the CUDA compiler and OpenCL compiler depends the sequence length. We discussed this issue in more detail in a later section.

4.3 Measurements without the lazy f evalua tion

To address the performance differences shown in Figure 3 in more detail, we evaluated performances of each configuration without the lazy-f evaluation. Figure 4 summarizes the results. We only used Conf.1, 2 and 5 because the warp vote function is used only in lazy-f evaluation, which means Conf.2 and Conf.4 become the same, and Conf.3 and Conf.5 also become the same here.

In Figure 4, the performance for Conf.6 which uses the texture memory shows oscillation. We stored the profile as an array of int4 type variables to reduce the number of instructions to fetch the profile. An int4 type variable contains four vectorized int type elements, which means access to the profile is done by four elements at a time. Thus, the number of global memory access for the profile is proportional to the $\lceil l/4 \rceil$, where l denotes the sequence length. As a result, the computation time becomes a terraced shape for the sequence length, and the GCUPS performance which is normalized by the sequence length shows oscillation. Now the point of the discussion is why performances of Conf.1 and Conf.5 do not oscillate just as Conf.2.

After checking the PTX, which is the low-level intermediate language for NVIDIA GPU, we found that the compiler had optimized the access to the profile on the global memory. For example, when we need to fetch $4n + 2$ profiles from the global memory for each row, the compiler generates n load instructions for int4 type and one load instruction for int2 type to save the load sequences. As far as our experiments, this optimization is done only for global memory access, not for texture memory access. This is why performance oscillation is observed only for Conf.2 in Figure 4.

Interestingly, the performances of Conf.1 and Conf.5 are all most the same in Figure 4 unlike in Figure 3. It suggests that the reason of performance difference between Conf.1 and Conf.5 shown in Figure 3 is caused by the lazy-f evaluation. Whereas a calculation part of H and E in Figure 2 has a fixed iteration number regardless of the sequence length (line 7 to 15), the number of iterations for the lazy-f evaluation depends on the data of both subject sequence and query sequence. It makes difficult for compilers to unroll and optimize the code, and thus the performance of the lazy-f evaluation should become more sensitive to the quality of the compilers. By checking PTX code for Conf.1 and Conf.5, we found that compilers of the CUDA and the OpenCL had generated different code in terms of instruction scheduling. According to the experiments data, OpenCL is more efficient than CUDA for longer sequences (37% faster at maximum). Since no global memory access is made in the lazy-f evaluation, this performance gap is due to instruction scheduling and optimization of shared memory access.

4.4 Differences between NVIDIA and AMD GPUs

We finally discuss the performance difference between GT200b

Table 4: Configurations of each implementation.

Configuration	Conf.1	Conf.2	Conf.3	Conf.4	Conf.5	Conf.6
Architecture	GT200b	GT200b	GT200b	GT200b	GT200b	RV870
Framework	OpenCL	CUDA	CUDA	CUDA	CUDA	OpenCL
Profile Storage	Global	Texture	Global	Texture	Global	Global
Reduction	Simple	Warp vote	Warp vote	Simple	Simple	Simple

and RV870 GPU cores. Comparing Conf.1 with Conf.6 in Figure 3, it is shown that the GT200b is better than the RV870 when sequence length is shorter than 450. For longer sequences, however, the RV870 continues to improve the performance and finally it gets faster than the best implementation of GT200b. The RV870 has characteristics different from the other configurations in terms of relationship between the performance and the sequence length. The proportional relationship that Conf.6 shows suggest that RV870 has a relatively simple memory access mechanism in which merits of large and continuous data transfer can be easily leveraged.

When the data set is small, dedicated programming framework in which programmers can explicitly utilize a sophisticated memory hierarchy with special hardware primitives would be a good choice. When the algorithm handles larger data, however, the merits of the memory hierarchy seem to become difficult to exploit. In this sense, a combination of an efficient compiler and a simpler GPU architecture would be better choice for applications that have a high degree of parallelism and require a large data set.

5. CONCLUSION

In this paper we presented performance comparisons of GPU implementations of the striped Smith-Waterman algorithm on two kinds of GPUs using the CUDA and the OpenCL frameworks. For long sequences, the best performance of 18 GCUPS was achieved on the RV870 using OpenCL. However, the implementation on GT200b using CUDA was the fastest for shot sequences. For the GT200b, the code generated by the OpenCL compiler was more efficient than by the CUDA compiler. However, the use of architecture specific mechanisms which are available only for CUDA such as the warp vote function and the texture memory improved the performance up to 97% and this effect exceeded the code inefficiency. On the other hand, the benefit obtained from the sophisticated memory hierarchy of the GT200b was restricted for long sequences compared to the simple OpenCL implementation on the RV870. This suggests that the approach of programming frameworks like OpenCL where programmers can not directly control architectural details can be a good solution for applications that handle large size data. Our challenging future work includes quantitative comparison of the frameworks in terms of productivity.

6. REFERENCES

[1] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, IEEE, 2009.

[2] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 126–131, IEEE, 2009.

[3] T. Smith and M. Waterman, "Identification of common molecular subsequences," *J Molecular Biology*, vol. 147, pp. 195–197, 1981.

[4] O. Gotoh, "An improved algorithm for matching biological sequences," *J Mol Biol*, vol. 162, pp. 707–708, 1982.

[5] S. Henikoff and J. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, p. 10915, 1992.

[6] S. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[7] L. Ligowski and W. Rudnicki, "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*, pp. 1–8, IEEE, 2009.

[8] K. Dohi, K. Benkridt, C. Ling, T. Hamada, and Y. Shibata, "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*, pp. 29–36, IEEE, 2010.

[9] Y. Liu, B. Schmidt, and D. Maskell, "Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," *BMC Research Notes*, vol. 3, no. 1, p. 93, 2010.

[10] M. Farrar, "Striped smith-waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[11] M. Farrar, "Optimizing smith-waterman for the cell broadband engine,"

[12] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Comput Appl Biosci*, vol. 13, no. 2, pp. 145–150, 1997.

[13] T. Rognes and E. Seeberg, "Six-fold speedup of smith-waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[14] "CUDA ZONE." `http://www.nvidia.com/object/cuda_home.html`.

[15] "OpenCL." `http://www.khronos.org/opencl/`.

[16] "UniProt." `http://www.uniprot.org/`.