International Journal of Parallel Programming
https://doi.org/10.1007/s10766-018-0617-3

ASW: Accelerating Smith–Waterman Algorithm on Coupled CPU–GPU Architecture

Huihui Zou[1] · Shanjiang Tang[1] · Ce Yu[1] · Hao Fu[1] · Yusen Li[2] · Wenjie Tang[3]

Received: 24 September 2018 / Accepted: 16 November 2018

Abstract

Smith–Waterman algorithm (SW) is a popular dynamic programming algorithm widely used in bioinformatics for local biological sequence alignment. Due to the $O(n^2)$ high time and space complexity of SW and growing size of biological data, it is crucial to accelerate SW for high performance. In view of the GPU high efficiency in science computation, many existing studies (e.g., CUDAlign, CUDASW++) speedup SW with GPU. However, the strong data dependency makes SW communication intensive, and the previous works fail to fully leverage the heterogeneous capabilities of the GPU machine for either the neglect of the CPU ability or the low bandwidth of PCI-e. In this paper, we propose ASW, which aims at accelerating SW algorithm with accelerated processing unit (APU), a heterogeneous processor integrates CPU and GPU in a single die and share the same memory. This coupled CPU–GPU architecture is more suitable for frequent data exchanging due to the elimination of PCI-e bus. For the full utilization of both CPU and GPU in APU system, ASW partitions the whole SW matrix into blocks and dynamically dispatches each block to CPU and GPU for the concurrent execution. A DAG-based dynamic scheduling method is presented to dispatch the workload automatically. Moreover, we also design a time cost model to determine the partition granularity in the matrix division phase. We have evaluated ASW on AMD A12 platform and our results show that ASW achieves a good performance of 7.2 GCUPS (gigacells update per second).

Keywords Smith–Waterman algorithm · Heterogeneous processors · APU · Load balancing

✉ Shanjiang Tang
tashj@tju.edu.cn

✉ Ce Yu
yuce@tju.edu.cn

✉ Wenjie Tang
tangwenjie@nudt.edu.cn

Extended author information available on the last page of the article

Published online: 01 December 2018

1 Introduction

Sequence alignment is a fundamental operation in bioinformatics, which can be used to compute a similarity score between two DNA or RNA sequences. However, as new bioinformatics techniques have been developed in the past decades, the size of biological databases has increased at an unprecedented rate, and it is hard to align those large size sequences in an acceptable time.

One approach is to globally compare two sequences using a dynamic programming method called Needleman–Wunsch (NW) algorithm [10], in which all the residues are compared to get a general similarity. A similarity matrix with the size of $m \cdot n$ is calculated, where m and n are the length of the two sequences. Based on NW, authors in [14] proposed the Smith–Waterman (SW) algorithm for local sequence alignment, which searches the most similar region between the two sequences. However, the computation complexity and the space complexity of the SW algorithm are both $O(n^2)$, which makes it impractical to handle large-size sequences. In order to reduce the execution time, two heuristic algorithms, FASTA [7] and BLAST [1] were proposed. However, they do not guarantee that the finding region is the optimal alignment.

To improve the performance of SW, lots of efforts have been made on accelerating SW algorithms on various accelerator processors. Rucci [11,12] accelerates SW algorithm using Intel/Alteras FPGA for long DNA sequences. SWAPHI-LS [8] adopts Xeon-Phi to speedup the computation. CUDAlign [3,4] presents a GPU-accelerated SW implementation, which supports single or multiple GPUs. However, in these works, CPUs are only used to transfer data and launch GPU workloads, and are idle during the computation. Co-running computation [16], has been proved to be an efficient method to address this problem by partitioning workloads and executing them on both CPUs and accelerators concurrently.

The emergency of Accelerated Processing Unit (APU) [2,13] brings new opportunities for SW algorithm acceleration. APU is *coupled* CPU–GPU architecture that combines CPU and GPU on a single chip. Compared to *discrete* CPU–GPU architectures, APU removes the PCI-e bus and instead unifies the memory address space between CPU and GPU, which can help accelerate the communication between them. It is aimed at achieving scalable and power-efficient high performance computing. APU is quite beneficial for communication-intensive applications and co-running computation systems that use both CPU and GPU for concurrent execution. SW is a communication-intensive algorithm based on dynamic programming that has a strong dependency for intermediate data during the computation [17]. It implies that it is suitable and efficient to run SW on APU system for high performance.

To implement SW algorithm on APUs, there are three challenges needed to be addressed. (1) *Workload partition* [20] To do the computation concurrently and efficiently, the score matrix is normally divided into blocks. A large block size can bring high utilization of CPU or GPU and reduce the communication. However, it will affect the cooperation between CPU and GPU, and decrease the overall utilization. In contrast, a small size will lead to opposite effects. (2) *Load balancing* Given the *coupled* CPU–GPU architecture, the performance can only be maximized by enabling the workload computation on both CPU and GPU simultaneously. However, it is difficult to achieve load balancing between CPU and GPU because of various factors, such as the difference in computing ability, the intrinsically strong data dependency of dynamic programming and the additional overhead of CPU for controlling and scheduling. (3) *Memory optimization* Due to the special memory model in APU, memory access pattern should be carefully designed to reduce the latency.

In this paper, we propose ASW, an efficient method that accelerates SW algorithm for APUs. To the best of our knowledge, this is the first design for accelerating SW algorithm on this platform based on co-running computation approach. (1) To address the workload partition and load balance, we propose a time cost model to obtain a suitable partition size; (2) we also design a dynamic scheduling method based on the directed acyclic graph (DAG) to dispatch workloads based on the difference between CPU and GPU; (3) in addition, we exploit the on chip shared memory of GPU, called local data store (LDS), to reduce the memory access latency of GPU cores. We have implemented ASW on an AMD A12 APU, and the experimental results show that ASW can achieve a performance of 7.2 GCUPS. By utilizing cost based block partition approach, it helps improve ASW's performance by 2.3x over that of the default in experience.

2 Related Work

2.1 SW Acceleration

There are a lot of studies (e.g., [3,4,8,9,11,12]) on accelerating SW on heterogeneous computing platforms (e.g., FPGA, Intel Xeon Phi, GPU). Enzo Rucci et al. [11,12] utilize FPGA to speed up SW algorithm with OpenCL, in which the score matrix is divided into vertical blocks and each block is executed row by row. Compared to SW acceleration on GPUs, this work can achieve a good power efficiency. SWAPHI-LS [8] aligns long sequences on Xeon Phi co-processor. It has explored the instruction-level parallelism within 512-bit SIMD instructions and thread-level parallelism over the many cores within a single Xeon Phi. It also achieves good performance on Xeon Phi clusters using MPI. CUDAlign [3,4] is a GPU-based SW implementation for comparing two sequences by exploiting CUDA and NVIDIA GPUs. In this work, a big score matrix is split into blocks and a wavefront-based method is used to execute each block. Additionally, a block pruning strategy is proposed to further reduce the number of blocks to be calculated. In the latest version, CUDAlign is implemented with supporting multiple GPUs. Yongchao Liu et al. [9] proposed a protein database search algorithm which carries out concurrent CPU and discrete GPU computation. A static workload distributing method in their work according to the computing power of CPU and GPU, which is inflexible and cannot achieve a good load balance between these two kinds of devices. However, these work cannot be directly applied to a *coupled* CPU–GPU architecture and the efficiency of co-running computation in these work is restrained by the high latency of PCI-e bus and the synchronization between CPU and GPU.

2.2 APU Computation

Many works have been done on performance optimization on APU platforms. He et al. [5] proposed a hash-join algorithm for APU platforms. In order to maximize the device efficiency and reduce memory stalls, they presented a cost model to estimate the execution time of different algorithm modules. Moreover, they also proposed an in-cache query co-processing paradigm [6] to optimize main memory On-Line Analytical Processing (OLAP) database queries. Zhang et al. [18] given APU-based implementations of algorithms in the Rodinia benchmark, which can be categoried into three types, CPU-only, GPU-only and co-running friendly algorithms. DIDO [19] is an in-memory key-value store system on APUs by split the system workflow into fine-grained tasks and pipeline them on both CPU and GPU. To balance workloads between these two parts, several optimization methods, including dynamic pipeline partitioning, flexible index operation assignment and work stealing, have been integrated into it. Nevertheless, the aforementioned works mainly focus on optimizing algorithms in database systems, which have a distinct abstract with the SW algorithm.

3 Background

3.1 Smith–Waterman Algorithm

Let S and T denote the sequences to get aligned. Let m and n denote the lengths of S and T, respectively. Denote by $H_{i,j}$ the maximal alignment score of $S_0 \ldots S_i$ and $T_0 \ldots T_j$. Denote by E, F the matrices to record the horizontal and vertical gap extending penalty. The first row and column of the score matrix are initialized to 0. Let $c(S_i, T_j)$ denote the score of S_i aligned to T_j. The SW algorithm is described as below.

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

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

$$H_{i,j} = max \begin{cases} 0, \\ E_{i,j}, \\ F_{i,j}, \\ H_{i-1,j-1} - c(S_i, T_j) \end{cases} \tag{3}$$

3.2 Coupled CPU–GPU Architecture

Figure 1 shows two kinds of heterogeneous CPU–GPU architecture, *discrete* CPU–GPU architecture and *coupled* CPU–GPU architecture. In Fig. 1a, CPU and GPU are designed with a separated memory space and CPU is responsible for exchanging data with GPU through PCI-e bus. Programmers are required to handle the communication between CPU and GPU carefully to obtain a better performance. In addition, duplicate data may be kept both in the host and GPU memory. In Fig. 1b, CPU and GPU are integrated on the same die and share the same memory space. These features make the data exchange between CPU and GPU more efficient. Also, GPU can share the

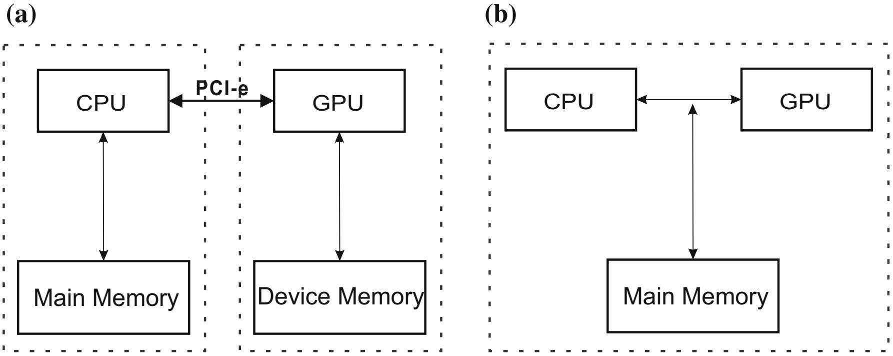

Fig. 1 An overview of hetergenerous CPU–GPU architecture. **a** *Discrete* CPU–GPU architecture and **b** *coupled* CPU–GPU architecture

big memory with CPU. It is better for co-running computation. Numerous products have been produced in recent years, such as AMD APU [2,13].

3.3 OpenCL

In AMD OpenCL [15] programming, threads are grouped into workgroups, each of which executes entirely on a compute unit (CU). Within a workgroup, threads run in groups of 64 consecutive threads called wavefronts. Threads within a wavefront always execute the same instruction at the same time. If threads in a wavefront need to run different instructions, the instruction executions are serialized. The situation mentioned above is called *divergence*, which should be avoided to achieve high performance. Each compute unit has 32 KB of LDS, which is shared among all active work-groups. LDS is allocated on a per-work-group granularity, which allows multiple wavefronts to share the same local memory allocation. However, large LDS allocations eventually reduce the number of workgroups that can be active.

4 Implementation

In this section, we first present the data partition method adopted in the ASW, and then show the scheduling algorithm for workload distribution and load balance. Additionally, we also give an introduction to the memory optimization strategy.

4.1 Data Partition

In order to calculate the score matrix concurrently and efficiently, we need to partition this matrix into blocks and then process these blocks with both CPU and GPU. Considering the limited size of APU's LDS and the difference in the computing capability between CPU and GPU, it is crucial to determine a suitable block size. To some extent, a large block size can help improve the utilization of GPU, but it may generate too

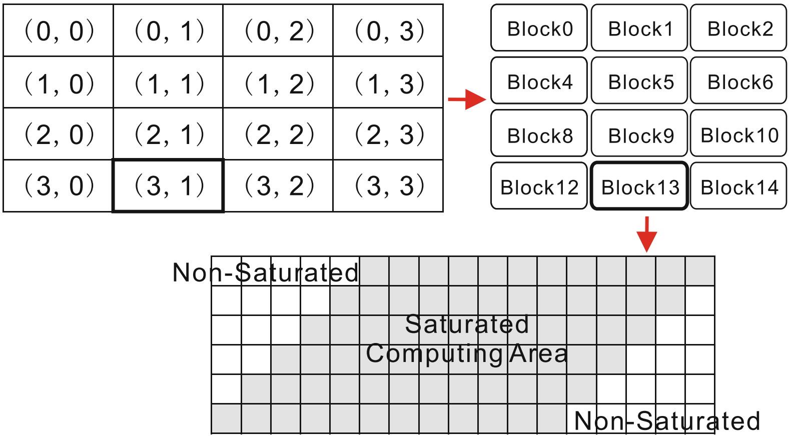

Fig. 2 Block partition. The whole score matrix is split into a block matrix. Cells in blocks are executed as wavefront method. Area where all the threads are busy named by *Saturated Computing Area*, corresponds to the gray cells. The area where existing idle threads called *Non-Saturated Computing Area*

Table 1 Notations in the time cost model

Notation	Description
m	The width of the score matrix
n	The length of the score matrix
w	The width of a score block
l	The length of a score block
o	The overhead of maintaining the DAG graph and data communication
T_c	The average time of computing one matrix cell
T_B	The time of computing a score block
c	The concurrency degree
O_t	The proportion of non-saturated period. $O_t = w/(l + w - 1)$

many intermediate results, which can not be maintained in the LDS, and further leads to a memory performance degradation. On the contrary, a small block size may result in a low utilization of GPU cores. An example of this partition phase is shown in Fig. 2. A SW score matrix is split into a 4 × 4 block matrix, and each block is assigned with a id according to the Cartesian coordinate. These blocks can be categoried into three computing areas: two non-saturated computing areas and a saturated computing area. In the saturated domain, all processing units can be fully utilized.

To better understand how the block size affect the overall performance, we present a theoretical analysis and notations adopted can be found in Table 1.

1. The execution time of computing a score block T_B is:

$$T_B = (w/O_t) \cdot T_c \tag{4}$$

2. Suppose that m is divisible by w and n is divisible by l, the total computing time $T_{computing}$ can be obtained by:

$$T_{computing} = \frac{m \cdot n}{w \cdot l} \cdot T_B \tag{5}$$

3. And the total overhead $T_{overhead}$ is defined as:

$$T_{overhead} = \frac{m \cdot n}{w \cdot l} \cdot o \tag{6}$$

4. Finally, we can calculate the total execution cost T_{matrix} as:

$$\begin{aligned} T_{matrix} &= (T_{computing} + T_{overhead}) \cdot c \\ &= \frac{m \cdot n \cdot c}{w \cdot l} \cdot ((l + w - 1) \cdot T_c + o) \end{aligned} \tag{7}$$

4.2 DAG-Based Dynamic Scheduling Method

Given the data partition results, it is important to design and implement a scheduling algorithm to balance workloads between CPU and GPU. Traditional scheduling method is to dispatch and execute blocks on the same diagonal at the same time in a wavefront manner, which is adopted in [3,18]. All the blocks in the same anti-diagonal are assigned to the CPU or GPU for further processing, but not both of them. To achieve the load balance and improve the resource utilization of APU platforms, a new scheduling algorithm is proposed in this section (Fig. 3).

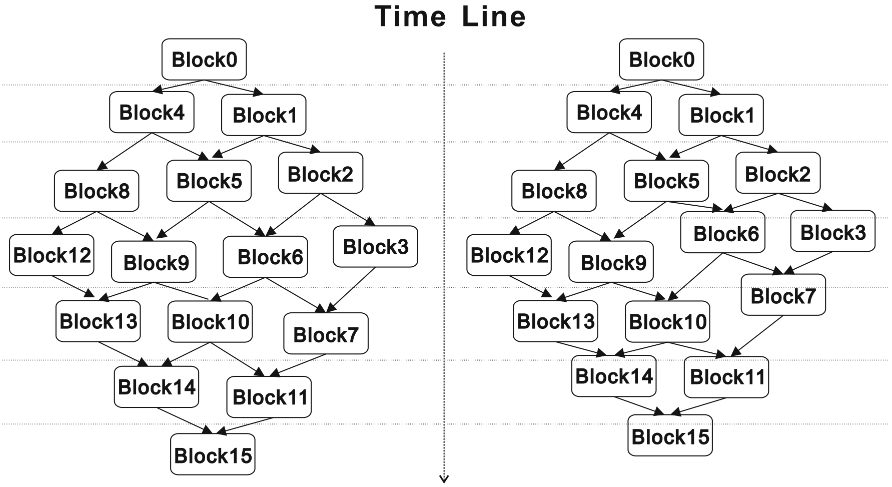

Fig. 3 The benefit of DAG-based scheduling method. Blocks in Fig. 2 map into two methods, left is the wavefront methods and right is the DAG method. It is clear that DAG-based scheduling method estimates the synchronization

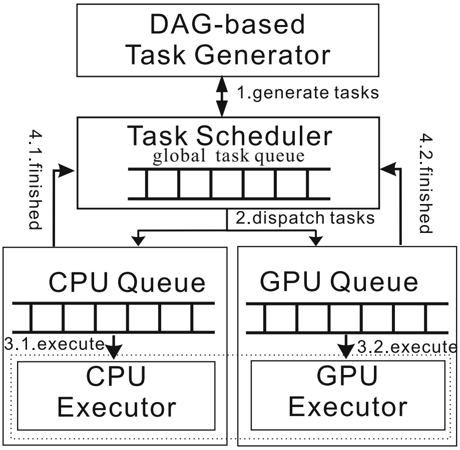

Fig. 4 The entire procedure of scheduling method

Algorithm 1: Traditional DAG-based task generator

```
Procedure Task Generator(QueueC/QueueG, task Queue)
while $!isEmpty$(QueueC/QueueG) do
    $A = taskQueue.front()$
    $taskQueue.pop()$
    if $A = \frac{m \cdot n}{l \cdot w} - 1$ then

        return $finished$
    end if
    $changeState(A)$
    $changeNeigbourState(A)$
end while
Traverse(DAG)
End Procedure
```

DAG-based dynamic scheduling method includes three parts, DAG-based task generator, Task Scheduler and CPU/GPU executor as illustrated in Fig. 4. CPU executor and GPU executor are responsible for the executing on CPU and GPU. Initially, the first block is pushed into task scheduler and then dispatched to CPU/GPU executor. Once the tasks are finished, task scheduler will dispatch other tasks to them. If there is no task in task scheduler, the DAG-based task generator will be notified to generate new computing tasks until the last one is finished. To further improve the performance, we have optimized the task generating method. The traditional DAG-based Task Generator method (Algorithm 1) needs to traverse the entire DAG to generate the tasks. It may waste lots of time. Therefore, we propose an optimized task generating method (Algorithm 2) according to SW algorithm to reduce the traverse time. Note that, the block computation order is from left to right, from upper to bottom in a wavefront manner. We follow this order in our method and there is no need to traverse all blocks.

Algorithm 2: SW-specific DAG-based task generator

```
Procedure Task Generator(QueueC/QueueG, task Queue)
while !isEmpty(QueueC/QueueG) do
    $A = taskQueue.front()$
    $taskQueue.pop()$
    if $A = \frac{m \times n}{l \times w} - 1$ then
        return $finished$
    else if $A\%n/l == 0$ then
        $taskQueue.push(A + 1)$
        $taskQueue.push(A + n/l)$
    else if $(A + 1)\%n/l == 0$ then
        break;
    else
        $taskQueue.push(A + 1)$
    end if
end while
End Procedure
```

4.3 Memory Optimized Design

Two kinds of memory can be accessed by GPU in the APU, the host main memory and local data store (LDS). LDS is much more faster but smaller than the host memory, which can be treated as an user-controlled GPU cache. Taking the memory access pattern of the SW algorithm into account, the computation of most cells will require data from its upper, left and left-upper cells. It may result in a high latency if all data needed for the computation of a score block on the GPU are stored in the main memory.

First, we adopt a method to fit the computing inner a block into LDS. For convenience, an anti-diagonal line is referred as a $line$. As showed in Fig. 5, the computation of cells in the current line (CL) only depends on cells in the second preceding line(SPL) and the preceding line (PL). It means that it is safe to free or reuse the memory space of lines before the second preceding line.

Second, since the computation of a data block need the results from the upper, left and left-upper blocks, it is expensive to maintain all of them. On the contrary, as

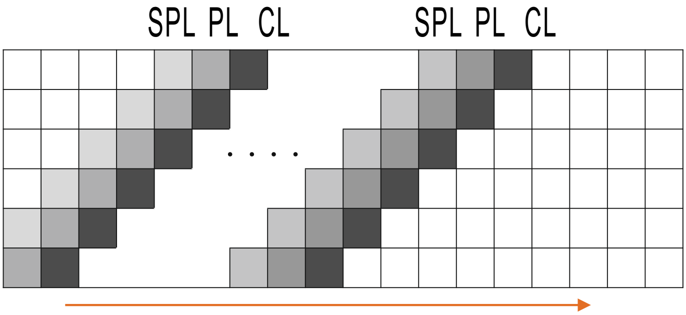

Fig. 5 Solution of reducing memory cost in block execution. Only the *SPL*, *PL*, *CL* are kept in LDS

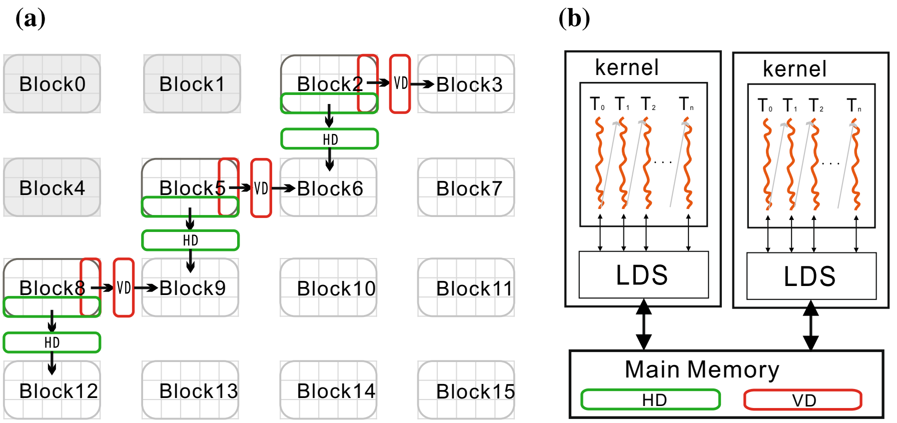

Fig. 6 Solution of solving the dependency between blocks. Results in the right-most column and the bottom row in finished blocks, which called *VD* and *HD*, will be flush into the main memory buffer. And they will be accessed in the subsequent execution. Block execution is deployed in LDS. **a** Data transfer procedure and **b** buffer deployment

showed in Fig. 6, only the bottom row and the rightmost column will be referred in the computation of other data blocks, thus we only design and utilize two buffers to save them. Figure 6a shows the data transfer procedure. Once a block is finished, results in its rightmost column and bottom row which we called vertical dependency (*VD*) and horizontal dependency (*HD*), will be flushed into the main memory to solve the dependency. Blocks in subsequent execution will access the main memory to obtain the results before execution. Figure 6b shows the deployment of data. It is better to transfer data between blocks by using the main memory since the amount of data is large. While, LDS is used for data sharing between threads.

5 Evaluation

5.1 Experimental Setup

Platform: All experiments are performed on an AMD A12-9800 APU. The GPU part has eight computing units (CUs) running at 1 GHz and each of them consists of four lanes with 16 ALUs. Each CU has its own LDS (32 KB) and L1 caches whereas a L2 cache is shared among all the CUs. The CPU part has 4 cores running at 4.2 GHz. There is an 8 GB main memory shared by the CPU and GPU. The thermal design power is merely 65 W. Device specification are described in Table 2.

Dataset: Real DNA sequences from the NCBI site (www.ncbi.nlm.nih.gov) are used. As shown in Table 3, the sizes of the real sequences range from 10 KBP to 3 MBP. The best scores and their ending positions obtained from the comparison are listed.

Algorithm 3: Process Block

```
Procedure KERNEL($B_k$)
$(i, j) \leftarrow GetBlockCoordinate(B_k)$
$tx \leftarrow GetThreadCoordinate()$
$Dep \leftarrow LoadDepFromGtoL(i, j)$
$Seq \leftarrow LoadSeqFromGtoL(i, j)$
$Initialize(Level_{ll}, Level_l, Level_c, score)$
for $line = 0 \rightarrow l + w - 1$ do
    $i0 \leftarrow get\_local\_id(0)$
    $j0 \leftarrow line - i$
    while $j0 >= 0$ and $j0 <= line$ do
        $Level_c \leftarrow SW(Level_{ll}, Level_l, Dep, Seq, score, (i0, j0))$
    end while
    $Level_{ll} \leftarrow Level_l$
    $Level_l \leftarrow Level_c$
end for
$Barrier$
$UpdateDepFromLtoG(i, j)$
return $(score, B_k)$
End Procedure
```

Table 2 Device specification

	GTX 750	A12
Stream processors	640	512
Core frequency	1.02	1.0
LDS (KB/CU)	–	32
Performance (GFLOPS)	1306	930
Memory bandwidth (GB/S)	86.4	18

Table 3 Comparison of the real sequences

Cmp.	Sequence 1		Sequence 2		Score	Position
	Accession	Size	Accession	Size		
10K	AY352275.1	10K	AF133821.1	10K	5027	(9418 , 9114)
50K	NC 001715.1	57K	AF494279.1	57K	51	(41,352 , 33,438)
160K	NC_000898.1	162K	NC_007605.1	172K	18	(41,058 , 44,353)
500K	NC_003064.2	543K	NC_000914.1	536K	48	(308,558 , 455,134)
1M	CP000051.1	1044K	AE002160.2	1073K	88,353	(1,072,950 , 722,725)
3M	BA000035.2	3147K	BX927147.1	3283K	4226	(2,991,493 , 2,689,488)

5.2 Effect of Partition Size

We have conducted a contrast experiment to show the difference between the estimated execution time trend according to the cost model (Formula 7) and the real execution time trend under different block sizes. The 160K sequence pair is used for testing. Results are summarized in Fig. 7. The horizontal axis gives the block size and the vertical axis shows the calculating time trend. The blue curve denotes the theoretical

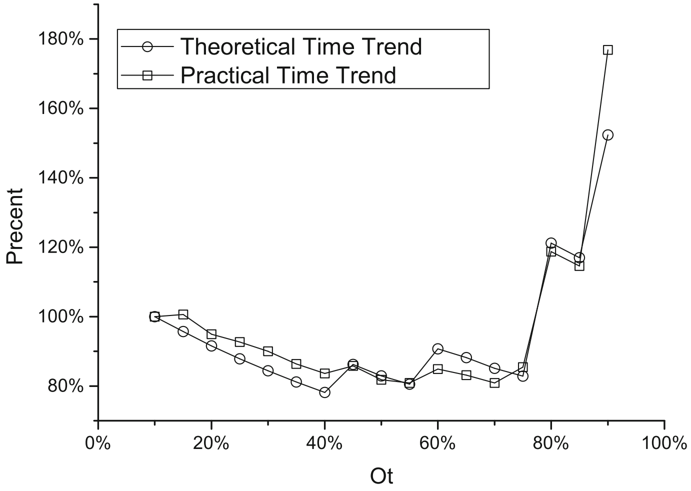

Fig. 7 Efficiency of data block partition. It shows the execution time trend between the real time trend and theoretical trend under different O_t. The chosen O_t (40%) according to the cost model achieves 97% of the best performance ($O_t = 70\%$)

time according to the model. For simplicity, we do a normalization comparison with the time cost when O_t is 10%. The overhead is taken into consideration. We can clearly see that: First, the time cost is decreasing progressively within a certain range and increasing suddenly at some block point. It is because the resource allocation is dependent on the block size and has an influence on workgroup size in OpenCL execution. For example, when O_t is in 85–90%, there is a sudden increase. Because resource allocation reaches a bottleneck and the workgroup number on each core decreases at that point; Secondly, the two curves are roughly the same. It verifies the correctness of our simulation.

5.3 Load Balancing

This section comes to evaluate load balancing strategy by running ASW on CPU-only, GPU-only and CPU–GPU. Table 4 shows the experimental results. Methods are marked as *APU*, *ASW_C* and *ASW_G*, respectively. We can obtain that *APU* performs better than *ASW_C* and *ASW_G*. It indicates that our ASW can achieve a good load balancing between CPU and GPU.

5.4 Efficiency of ASW

In this section, comparison with CUDAlign and MASA-OpenCL are tested and listed in Table 5. All experiments are performed on GTX 750. ASW_N refer to running ASW under NVIDIA GTX 750. ASW_A represents it on APU A12 GPU. We can obtain that:

Table 4 Efficiency of load balancing strategy

Cmp.	ASW_C CPU-only	ASW_G GPU-only	ASW CPU + GPU
10K	0.345	1.5	1.7
50K	0.434	5.05	5.27
160K	0.464	6.18	6.32
500K	0.457	6.81	7.19
1M	0.455	7.04	7.2
3M	0.46	6.9	7.11

Table 5 Efficiency of ASW

Cmp.	CUDAlign GTX750	MASA-OpenCL GTX750	ASW_N GTX750	ASW_A A12	MASA-OpenMP A12
10K	6.73	5.3	6.52	1.7	0.47
50K	15.45	14.29	17.6	5.27	1.11
160K	20.66	18.57	20.08	6.32	1.17
500K	19.73	17.91	17.12	7.19	1.14
1M	20.13	18.03	17.3	7.2	1.11
3M	19.59	18.30	17.9	7.11	1.05

1. In the same NVIDIA GPU, ASW almost has the same performance with CUDAlign. And even more better when the size of sequences are small.
2. ASW_A is worse than other methods.

We can summarize four reasons to explain that ASW_N is lower than CUDAlign. First, the high overhead of OpenCL. CUDAlign and MASA-OpenCL use the same strategies to achieve high performance. It is clear that MASA-OpenCL is not as good as CUDAlign, which shows the high overhead of OpenCL. Second, the block partition strategy. CUDAlign adopt the parallelogram wavefront methods, which has no Non-Saturated Computing Area. The whole blocks are divided into two parts: long and short phrase to make sure the correct answer. However, to run the workload on CPU concurrently, ASW use the normal wavefront. There are two Non-Saturated Computing Area (Fig. 2). Also, this is the reason why ASW do better in small sequence comparison. Third, the device specification. ASW is designed to perform on APU. Data is placed in LDS to avoid using the high latency memory (DDR3), so that the number of workgroup reduces in a compute unit. And this strategy is not very suitable in discrete GPU with a high bandwidth memory. Forth, the parameter setting. In CUDA, threads run in a groups of 32 consecutive threads. In OpenCL, there are 64 threads. There exists a difference. Due to the four reasons, ASW shows not as good as CUDAlign. But it achieves a good performans in small size sequence comparison.

Compared with GTX 750, A12 has fewer cores, a lower memory bandwidth and performance. The memory bandwidth is very low because of the DDR3, and the discrete GPUs use GDDR5. Also, to integrate GPU on a same chip with CPU, the performance of GPU is sacrificed. There are less cores and memory on the APU platform. But we have a lower TDP and price. But this is orthogonal to our ASW algorithm. Since coupled CPU/GPU architecture is a trend for future chip. We believe its computing capacity will catch up with or even exceed NVIDIA GPU in the future. Then there will be a significant performance improvement by using ASW. To summarize, our research is a leading attempt on future coupled CPU/GPU architecture.

6 Conclusion and Future work

The SW algorithm is a critical application in bioinformatics and has become the base of more sophisticated alignment technology. However, it is time consuming and there are many work attempting to accelerate it in different platforms (e.g., GPU). In this paper, we proposed ASW, the first method of communication-intensive SW algorithm for APUs, in which CPU and GPU communicate data via a shared memory instead of low bandwidth and high latency PCI-e bus. The experimental results show the efficiency of ASW. Our research is a leading attempt on future *coupled* CPU–GPU architecture. In future, we plan to extend our ASW for DNA or protein database searching problem.

Acknowledgements This work is supported by the Natural Science Foundation of Jilin Province (CN) (61602336), National Natural Science Foundation of China (61370010, 61702527) and Natural Science Foundation of Tianjin City (18JCZDJC30800).

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990)
2. Branover, A., Foley, D., Steinman, M.: AMD fusion APU: Llano. IEEE Micro **32**(2), 28–37 (2012)
3. De Oliveira Sandes, E.F., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G., Melo, A.C.M.: Cudalign 4.0: incremental speculative traceback for exact chromosome-wide alignment in GPU clusters. IEEE Trans. Parallel Distrib. Syst. **27**(10), 2838–2850 (2016)
4. De Oliveira Sandes, E.F., Miranda, G., De Melo, A.C., Martorell, X., Ayguade, E.: CUDAlign 3.0: parallel biological sequence comparison in large GPU clusters. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 160–169. IEEE (2014)
5. He, J., Lu, M., He, B.: Revisiting co-processing for hash joins on the coupled CPU–GPU architecture. Proc. VLDB Endow. **6**(10), 889–900 (2013)
6. He, J., Zhang, S., He, B.: In-cache query co-processing on coupled CPU–GPU architectures. Proc. VLDB Endow. **8**(4), 329–340 (2014)
7. Lipman, D.J., Pearson, W.R.: Rapid and sensitive protein similarity searches. Science **227**(4693), 1435–1441 (1985)
8. Liu, Y., Tran, T.T., Lauenroth, F., Schmidt, B.: SWAPHI-LS: Smith–Waterman algorithm on Xeon Phi coprocessors for long DNA sequences. In: IEEE International Conference on CLUSTER Computing, pp. 257–265 (2014)
9. Liu, Y., Wirawan, A., Schmidt, B.: Cudasw++ 3.0: accelerating Smith–Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinform. **14**(1), 117 (2013)
10. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970)
11. Rucci, E., García, C., Botella, G., De Giusti, A., Naiouf, M., Prieto-Matias, M.: OSWALD: OpenCL Smith–Waterman on Altera's FPGA for large protein databases. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 3, pp. 208–213. IEEE (2015)

12. Rucci, E., Garcia, C., Botella, G., Giusti, A.D., Naiouf, M., Prieto-Matias, M.: Accelerating Smith–Waterman alignment of long DNA sequences with OpenCL on FPGA. In: International Conference on Bioinformatics and Biomedical Engineering, pp. 500–511 (2017)
13. Ryzen APU. https://www.amd.com/en/products/apu/amd-ryzen-5-2400g (2018). Accessed 12 Feb 2018
14. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
15. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
16. Tang, S., He, B., Zhang, S., Niu, Z.: Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU–GPU architectures. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 875–886. IEEE (2016)
17. Tang, S., Yu, C., Sun, J., Lee, B.S., Zhang, T., Xu, Z., Wu, H.: EasyPDP: an efficient parallel dynamic programming runtime system for computational biology. IEEE Trans. Parallel Distrib. Syst. **23**(5), 862–872 (2012)
18. Zhang, F., Zhai, J., He, B., Zhang, S., Chen, W.: Understanding co-running behaviors on integrated CPU/GPU architectures. IEEE Trans. Parallel Distrib. Syst. **28**(3), 905–918 (2017)
19. Zhang, K., Hu, J., He, B., Hua, B.: DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In: IEEE International Conference on Data Engineering, pp. 671–682 (2017)
20. Zhang, F., Wu, B., Zhai, J., He, B., Chen, W.: FinePar: irregularity-aware fine-grained workload partitioning on integrated architectures. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, pp. 27–38. IEEE Press (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Huihui Zou[1] · Shanjiang Tang[1] · Ce Yu[1] · Hao Fu[1] · Yusen Li[2] · Wenjie Tang[3]

Huihui Zou
zouhuihui@tju.edu.cn

Hao Fu
haofu@tju.edu.cn

Yusen Li
liyusen@nbjl.nankai.edu.cn

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China

[2] School of Computing, Nankai University, Tianjin, China

[3] College of Systems Engineering, National University of Defense Technology, Changsha, Hunan, China