Automatic Parameter Optimization for Edit Distance Algorithm on GPU

Ayumu Tomiyama and Reiji Suda

The University of Tokyo

Abstract. In this research, we parallelized the dynamic programming algorithm of calculating edit distance for GPU, and evaluated the performance. In GPU computing, access to the device memory is likely to be one of the primal bottleneck due to its high latency, and this effect gets noticeable especially when sufficient number of active threads cannot be secured because of the lack of parallelism or overuse of GPU resources. Then, we constructed a model that approximates the relations between the values of parameters and the execution time considering latency hiding, and by using this model, we devised a method of automatic tuning of parallelization parameters in order to attain high performance stably even when the problem size is relatively small.

1 Introduction

The problem of analyzing the similarities of a given string with patterns and finding their optimum alignment is called approximate string matching. It is one of the important problem in information science applied not only to text retrieval but also to a variety of fields including computational biology. The computation of approximate string matching, however, includes some computationally expensive steps. In this research, we worked on acceleration of the algorithm for calculating Levenshtein edit distance by using graphics processing units (GPUs) and made an evaluation of its performance.

GPUs, as the name suggests, were originally created as a hardware for image processing. They have quite a lot of computing units, which produce high peak performance at a relatively-low cost. Then, the idea to utilize the huge computing power of GPU for calculation other than image processing was born under the name of GPGPU (General Purpose computing on GPUs,) and today this technique has a wide variety of applications. Although GPGPU programming became easily accessible thanks to the improvement of GPU architectures and appearance of some useful developing environments including CUDA (Compute Unified Device Architecture) [7] provided by NVIDIA, it is generally still difficult to exploit the full performance of GPUs. In order to utilize all of the computing units in GPUs, it is important to select highly parallelized algorithm and to create a large number of threads. The cost of memory access is also a major factor determining overall performance. To reduce this cost, it is important in terms of latency hiding to increase the number of simultaneously executed threads by setting a limitation on the size of used resource per thread and increasing the total number of threads itself. Moreover, efficient utilization of low-capacity but fast shared memory is also a common practice. These optimization can not be achieved only by the selection of parallel algorithms, but it is also indispensable to adjust values of parameters related to the parallelization depending on the capacity of used GPUs.

As for approximate string matching on GPU, there are several researches on implementation of parallel versions of Smith-Waterman algorithm for GPUs. The Smith-Waterman algorithm closely resembles the edit distance algorithm in the dependence relationship between computed elements, so almost the same parallelization and optimization scheme can be applied.

The origin of the implementation of the Smith-Waterman algorithm for GPUs dates back to the age before the CUDA became common. Liu et al. [3] implemented the Smith-Waterman algorithm on GPU by mapping the algorithm on rendering pipeline with using OpenGL API. They parallelized the algorithm by making use of the fact that the cells on the same anti diagonal in the computation region can be simultaneously calculated. After the appearance of CUDA, several researches on the parallel Smith-Waterman algorithm on GPU are conducted. Manavski et al. [5] and Munekawa et al. [6] implemented the algorithm by using CUDA. These implementations are similar in that alignments of multiple combination of strings are calculated in parallel, but different in that each thread computes the whole alignment (inter-task parallelization) in the approach of Manavski et al, while it is covered by whole threads in a thread block (intra-task parallelization) in the approach of Munekawa et al. Liu et al. [4] not only improved the performance but also eased the restriction caused by overuse of memory on GPU by covering both of the inter-task and inner-task parallelization. Ling et al. [2] also resolved the restriction on the length of the strings arisen from the limitation of available resources of GPUs by adopting a divide and conquer approach. Dohi et al. [1] improved the performance by using some technique including the divide and conquer approach and an idea of reducing the cost of thread synchronization.

Also after those, several implementations appear and show pretty good performance as compared with the ones for CPU accelerated by some heuristic methods. Most of them put emphasis on the performance when there is a lot of queries, which enables simultaneous calculation of matchings of multiple combinations of strings and patterns, and ensures sufficient number of threads. On the other hand, they seem to be giving little consideration on the case when the number of active threads is limited because of the lack of queries relative to the capacity of the used GPUs. When sufficient number of active threads is not secured, the performance drops beause of failure in latency hiding or load balancing.

In this article, we propose a parallel algorithm of calculating edit distance of a pair of strings on GPU. We analyzed the relationship between computation time and the values of parameters associated with parallelization: the block size parameters, and the number of active threads. As a result, we confirmed that the latency of the device memory and the hiding of it have the strongest effect on the performance other than the simple number of arithmetic and memory reference instructions. Then, considering them, we constructed an estimation model of execution time, and applied it to our system to automatically select the optimum block size.

The rest of this article is constituted as follows. First we introduce the parallel edit distance algorithm with brief explanation of GPU architecture in Chapter 2. Then, in Chapter 3, we present our estimation model of computation time. Then we show the experimental results in Chapter 4, and finally we conclude this article in Chapter 5.

2 Edit Distance Algorithm and Parallelization

First, we introduce the definition of edit distance and the serial version of the dynamic programming algorithm for calculating it. Then we propose our parallel algorithm after showing the features of CUDA GPUs. Note that in this research we assume the use of GPUs supporting CUDA whose compute capability is 1.2 or 1.3.

2.1 Dynamic Programming Algorithm

Edit distance of two strings is defined as the minimum number of editing operations required for transforming one string into the other. There are three types of operations available: insertion, deletion, and substitution of a character. For exapmle, edit distance of strings "change" and "hunger" is three, for the former string can be transformed into the latter one by the following three operations: deletion of 'c', substitution of 'u' for 'a', and insertion of 'r'.

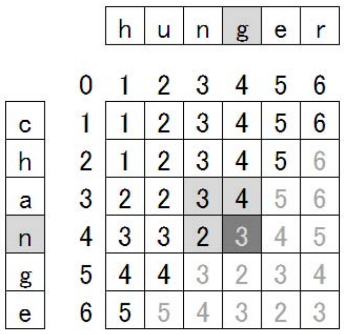
In this paper, we let $d(str1, str2)$ denote edit distance of the two strings $str1, str2$. Also, $|str|$ denotes the length of string str, $str[i]$ the i-th character of str $(0 \leq i < |str|)$, $str[i..j]$ the substring from the i-th character to the j-th character of str $(0 \leq i \leq j < |str|)$.

Edit distance $d(str1, str2)$ can be calculated by using dynamic programming. Specifically, it can be obtained from edit distances $d(str1[0..i], str2[0..j])$ of prefixes of the strings. Considering how to match the last characters of $str1[0..i]$ and $str2[0..j]$ by using one or none of the three editing operations, the edit distance $d(str1[0..i], str2[0..j])$ can be calculated according to the following formula:

$$\begin{aligned} d(str1[0..i], str2[0..j]) = min(&d(str1[0..i-1], str2[0..j]) + 1, \\ &d(str1[0..i], str2[0..j-1]) + 1, \\ &d(str1[0..i-1], str2[0..j-1]) + c(str1[i], str2[j])) \end{aligned} \tag{1}$$

where $c(str1[i], str2[j])$ is 0 if $str1[i]$ equals to $str2[j]$, and otherwise this value is 1.

Therefore, the edit distance $d(str1, str2)$ can be calculated by completing a table of edit distances $d(str1[0..i], str2[0..j])$ of prefixes of $str1$ and $str2$ like



		h	u	n	g	e	r
	0	1	2	3	4	5	6
c	1	1	2	3	4	5	6
h	2	1	2	3	4	5	6
a	3	2	2	3	4	5	6
n	4	3	3	2	3	4	5
g	5	4	4	3	2	3	4
e	6	5	5	4	3	2	3

Fig. 1. Table of edit differences of suffixes

one shown in figure 1. First, the values of leftmost cells and topmost ones in the table are trivially obtained. On the other hand, the value of each inner cell can be obtained from those of the immediate left, upper left, and upper cells of its own by using the formula (1), so they can be sequentially calculated from the upper left corner to the lower right corner. The amount of calculation is $O(|str1||str2|)$.

The calculation has a certain level of parallelism, and there is also flexibility in what order to calculate values of cells.

2.2 GPU Architecture

GPU mainly consists of several multiprocessors (MPs) and a device memory. A MP is a group of eight simple processors called scalar processors (SPs). SPs in the same MP simultaneously execute the same instruction on different data like SIMD instructions. GPU provides great performance by making its numerous SPs carry the same operation in parallel.

The device memory is accessible from CPU and all MPs in the GPU. Generally, GPU program first copies data from host memory on CPU to the device memory on GPU. Then, it runs parallel codes, called CUDA kernels, which make each MP read data from the device memory, perform calculation, and write the results back to the memory. Finally it transfers the results to the host memory. Besides the device memory, each MP has its own low-latency memory called shared memory. Via shared memory, each SP can access computational results of other SPs in the same MP very fast.

In order to make GPU execute tasks, we have to give it a group of threads called grid. A grid consists of numerous groups of threads called thread blocks. The relationship of grid, thread block, and thread corresponds to that of GPU, MP, and SP. Each thread block is assigned to a MP, and each thread in the thread block is assigned to a SP in the MP. Each SP (or MP) concurrently executes several threads (or thread blocks) by frequently switching the thread (or thread block) to be executed.

In order to exploit the performance of GPU, there is a lot of things to be considered. In this research, we put emphasis on mainly two points.

One point is the hierarchical structure of tasks and the hardware. What SPs can do is restricted in that SPs in the same MP can simultaneously execute only the same instruction, so the sequence of executed instructions should be made as nearly equal as possible among threads in the same thread block by avoiding conditional branching. On the other hand, threads in the same thread block have advantage in that they can communicate with each other very fast by utilizing the shared memory and fast synchronization instruction, whereas communication among threads in different thread blocks is not supported. Therefore, it is an important matter how to split the entire processing into grids, into thread blocks, and into threads considering regularity and dependency of calculation.

The other point is the latency hiding. In GPU computing, access to the device memory is one of the primal bottleneck because of its high latency. Generally, it is hidden by executing instructions of other threads during the latency. Therefore, it is important to secure sufficient number of active threads in order to fill the latency. The number of active threads depends not only on the number of total threads but also on the amount of resources each thread block uses such as shared memory and registers.

2.3 Parallelization and Blocking

We parallelized the dynamic programming algorithm for GPU, and in order to reduce grid execution and access to the device memory, we brought in blocked algorithm.

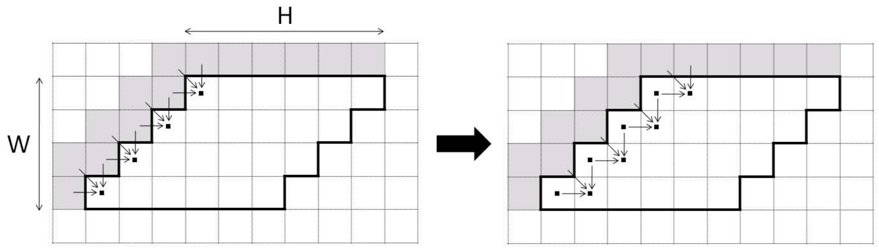


Fig. 2. Shape of a block and calculational procedure

Figure 2 shows the shape of a block by which the computational region of the dynamic programming algorithm is divided in this parallel algorithm, and how to calculate values of cells in the block in parallel. As explained in Section 2.1, the value of each cell can be directly calculated if those of the immediate left, upper left and upper cells are available. Therefore, if all the values of gray cells in Figure 2 are available, values of the leftmost cells in all the W rows of the parallelogram block can be simultaneously calculated. Once their values are calculated, then by using them, values of all their immediate right cells can be simultaneously calculated. In the same way, values of W diagonally aligned cells are calculated in parallel, and it takes H steps to complete the processing of all the cells in the block, in sequence from left cells to right ones.

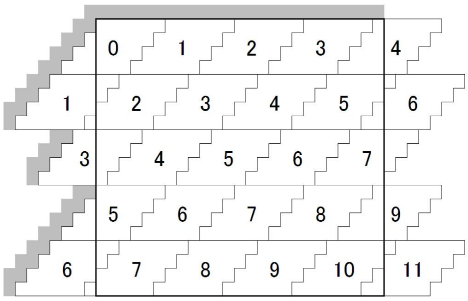


Fig. 3. Blocking

Figure 3 shows how to divide the computational region into the parallelogram blocks and what order to process the blocks. The blocks can be processed in ascending order of the number, and all blocks assigned the same number can be processed in parallel without inter-block communication. In parallelization of this algorithm for GPU, a grid is assigned to process all the parallelogram blocks tagged the same number, and in the grid, each thread block is assigned to calculate values of cells in a block region. In a block region, just W values of cells which are in different rows each other can be calculated in parallel at any step, so the processing of a block region can be evenly parallelized by assigning each thread calculation of values of all cells in a row in the block.

To be more precise, behavior of a thread block is expressed in Figure 4. First, threads load necessary data, values of gray part in the left block of Figure 4 and

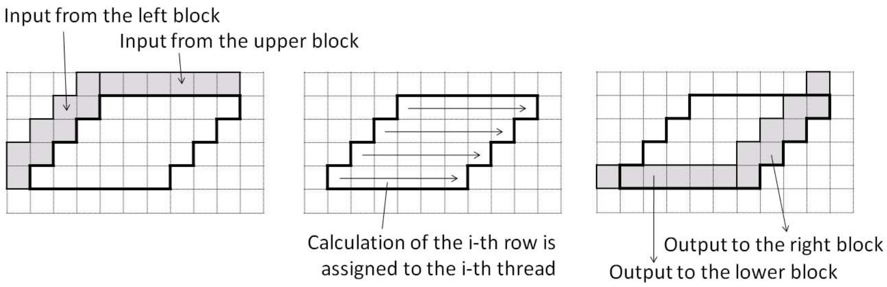

Input from the left block
Input from the upper block
Calculation of the i-th row is assigned to the i-th thread
Output to the right block
Output to the lower block

Fig. 4. Processing of a thread block

characters in the strings, from the device memory and store them on shared memory. The values of gray cells are just calculated by the previous grid execution. Then, each thread starts calculation of values of the assigned cells. W threads in the thread block act in synchronization, and process diagonally aligned cells in parallel, receiving necessary data, which was just calculated in the previous step by the neighbor threads, via shared memory. Finally, after all the cells are processed, they store the results which are necessary for the subsequent calculation in the next grid execution on the device memory. The necessary region includes only boundary part of the parallelogram block, which is colored gray in the left block of Figure 4, so the thread block has to store only the values of gray part in the right block of Figure 4 and do not have to do those of the whole inside of the block.

Furthermore, we adopted bidirectional calculation. Edit distance can also be calculated by putting together edit distance of prefixes of the two strings and that of suffixes. They can be calculated in parallel, and by using this property, the process of calculating edit distance can be divided into two half-size processings and post-processing whose amount of calculation is at most proportional to the length of the strings. By this division, the total amount of calculation slightly increases by the post-processing. Instead, the parallelism doubles, so the number of thread blocks per grid, and naturally that of threads, also doubles. In GPU computing, it is important to increase the number of threads in terms of latency hidind and load balancing, and as a whole it mostly results in an improvement in performance.

3 Optimization of Block Size

It is important to configure appropriate values of block size parameters, W and H in Figure 2, because they determine various quantities related to performance of the parallel algorithm for GPU.

First, block size parameters determine the number of thread blocks in each grid, which is important in terms of load balancing. At the same time, they indirectly influence the number of active threads per MP by determining the size of resources, such as shared memory and registers, needed by a thread block. In grid execution, each MP executes multiple thread blocks in parallel by frequently switching the thread block to be executed. Here, the resources of the MP are distributed to the concurrently executed thread blocks. Therefore, the number of active thread blocks is limited by the total size of resources divided by the size a thread block uses. Basically it is desirable to increase the number of active threads in terms of latency hiding. Note that the number of active thread blocks per MP can not be more than eight because of the specification of CUDA, and accordingly, too small block size reduces not only the number of threads per thread block but also that of active threads per MP.

The block size parameters also determine the total amount of access to the device memory and other additional calculation arisen from parallelization, including grid executions. Generally, selecting small block size increases such cost.

Besides, it is also important to appropriately configure the proportion of W to H in order to reduce the cost.

Therefore, the block size should be carefully chosen considering trade-offs among these factors. In this research, we constructed a model to estimate the grid execution time from the parameters, and made the system to choose the block size which minimize the total computation time estimated on the model.

Considering GPU architecture, there are not so many options of appropriate values of block size parameters, so we adopted full search: estimating the total computation time for all of the options, and actually calculating edit distance on the block size which minimize the estimated execution time.

In the following sections, we introduce a model for estimating the grid execution time, and then how to estimate values of parameters in the model.

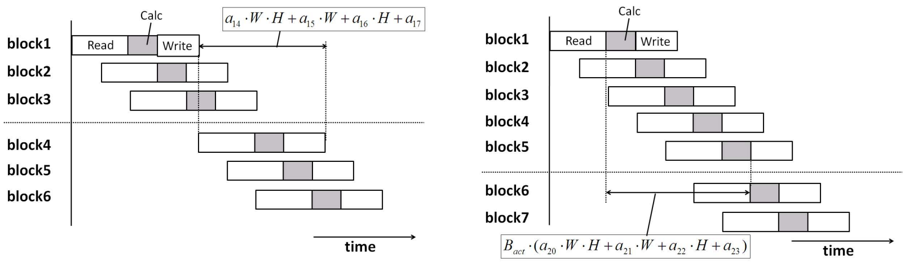
3.1 Model of Grid Execution Time

In order to choose optimum block size parameters, we constructed a model to estimate the grid execution time from block size parameters W and H, and the maximum number B_{MP} of thread blocks per MP, which is obtained from the total number B_{total} of thread blocks and the number N_{MP} of MPs as follows

$$B_{MP} = \left\lceil \frac{B_{total}}{N_{MP}} \right\rceil .$$

First, we calculated the maximum number B_{act} of active blocks. It is limited by the following two factors. One is the used resource size. In our algorithm, the number of used registers is not so large as to limit the number of active threads, but the shared memory usage may do. A thread block uses $(W+H)$ of character size area for storing the compared substring, and $(W+H)$ of integer size area for sharing values of the cells among the threads. Then, the number of active blocks is not more than the quotient of the total size of shared memory per MP and the used amount described above. The other factor is the specification of CUDA. The maximum number of active threads per MP is 1024, so B_{act} must not be more than the quotient of 1024 and the number of thread per thread block. In addition, the number B_{act} of active blocks itself is also limited not to exceed 8. Consequently, B_{act} is obtained as the maximum number such that all the above conditions are satisfied.

Then, we introduce our model to estimate the grid execution time from W, H, and B_{act}. In this parallel algorithm, each thread block first loads necessary data from the device memory all at once. Then, after synchronizing all threads in the block, threads start calculating values of the cells assigned to themselves with synchronization and communication with the neighbor threads through the shared memory. Finally, they write the results back on the device memory. The amount of main calculation per thread block is approximately proportional to the number of cells in the block region, that is to say, the product of W and H. On the other hand, the size of the data read from and written on the device memory has linear relationship to W and H. Besides, there are trivial processing whose amount is at most linear to W or H. Therefore, as the most fundamental approximation, the total amount of calculation of the grid per MP can be expressed as $B_{MP}(a_0WH+a_1W+a_2H+a_3)$, where each a_i is a constant. This approximation, however, may not be accurate depending on the degree of latency hiding.



Calc
$a_{14} \cdot W \cdot H + a_{15} \cdot W + a_{16} \cdot H + a_{17}$
block1
Read
Write
block2
block3
block4
block5
block6
time
Calc
block1
Read
Write
block2
block3
block4
block5
block6
block7
$B_{act} \cdot (a_{20} \cdot W \cdot H + a_{21} \cdot W + a_{22} \cdot H + a_{23})$
time

Fig. 5. Latency hiding

Figure 5 represents a simplified model of the flow of instruction execution in a thread block of the parallel edit distance algorithm. The left figure corresponds to the case when B_{act} is three and the latency of the device memory access is not fully hidden, while the right one corresponds to the case when B_{act} is five and the latency is fully hidden. In this algorithm, threads in the same thread blocks are synchronized before and after the access to the device memory. Therefore, during the memory access of one thread, only threads in different active thread blocks can contribute to latency hiding by executing the main task of calculating edit distance. The latency is completely hidden when the total latancy of one thread is shorter than the occupation time of computing units by threads in all active blocks except one.

Based on this model, the grid execution time, if the total number of blocks B_{MP} is less than B_{act}, can be simply approximated by the expression

$$T_{BS}(W, H, B_{MP}) = B_{MP} \cdot (a_{00}WH + a_{01}W + a_{02}H + a_{03}) + \\ (a_{04}WH + a_{05}W + a_{06}H + a_{07}).$$

The first term of the expression represents the main computation time of all the threads, while the second term represents the latency remained unhidden.

On the other hand, it depends on whether the latency is fully hidden or not how much extra time is needed for grid execution when the number of thread blocks in the grid increases by B_{act}. When the latency is not fully hidden, as described in the left one of Figure 5, a new thread block should wait for the completion of an old one without executable instruction in the MP, so the extra time can be approximated by the turn around time of execution of one thread block, the expression of which is

$$T_{DL}(W, H, B_{act}) = (a_{14}WH + a_{15}W + a_{16}H + a_{17}).$$

When the latency is fully hidden, some sort of calculation is always executed on the MP even during the latency of memory access, so the extra time can be approximated by the total amount of calculation of B_{act} thread blocks expressed as

$$T_{DH}(W, H, B_{act}) = B_{act} \cdot (a_{20}WH + a_{21}W + a_{22}H + a_{23}).$$

By using these functions, the grid execution time in each condition can be approximated by the expression

$$T_{TL}(W, H, B_{MP}, B_{act}) = T_{BS}(W, H, B_{MP}\%B_{act}) + T_{DL}(W, H, B_{act}) \cdot \left\lfloor \frac{B_{MP}}{B_{act}} \right\rfloor$$

if the latency is not fully hidden, and otherwise, it is approximated by

$$T_{TH}(W, H, B_{MP}, B_{act}) = T_{BS}(W, H, B_{MP}\%B_{act}) + T_{DH}(W, H, B_{act}) \cdot \left\lfloor \frac{B_{MP}}{B_{act}} \right\rfloor.$$

Whether the latency is fully hidden or not can be determined by comparing the function values: fully hidden when the value of $T_{DL}(W, H, B_{act})$ is smaller than that of $T_{DH}(W, H, B_{act})$, and otherwise, not fully hidden. Therefore, the execution time is also approximately expressed as

$$max(T_{TH}(W, H, B_{MP}, B_{act}), T_{TL}(W, H, B_{MP}, B_{act})).$$

In this way, the problem of estimating grid execution time comes down to that of estimating parameters a_{ij} in the functions T_{BS}, T_{DL}, and T_{DH}. Note that, in practice, we extended the forms of the functions T_{DL} and T_{DH} into

$$\begin{aligned} T_{DL}(W, H, B_{MP}) = B_{MP} \cdot (a_{10}WH + a_{11}W + a_{12}H + a_{13}) + \\ (a_{14}WH + a_{15}W + a_{16}H + a_{17}) \\ T_{DH}(W, H, B_{MP}) = B_{MP} \cdot (a_{20}WH + a_{21}W + a_{22}H + a_{23}) + \\ (a_{24}WH + a_{25}W + a_{26}H + a_{27}), \end{aligned}$$

the same in form as T_{BS}, for improving the quality and for convenience sake.

3.2 Parameter Estimation

We introduce the way of estimating values of parameters a_{ij} in the functions T_{BS}, T_{DL}, and T_{DH} from the sample data of block size parameters, the number of thread blocks, and the grid execution time.

As explained in the previous section, the grid execution time is approximated by the greater value of the two: linear combination of T_{BS} and T_{DL}, and that of T_{BS} and T_{DH}. Moreover, the three functions have linear relationships to the parameters a_{ij}. Therefore, values of the parameters a_{ij} can be approximated by using linear least-squares method if it is possible to determine in which case each performance data belongs to: the case where the latency is fully hidden or the other.

Then, we adopted iterative refinement method alternately repeating classification and parameter estimation. From the viewpoint of the model introduced in the previous section, whether latency is fully hidden or not basically depends on the ratio between the number of times loading from and storing on the device memory per thread and the total amount of calculation of all active thread blocks except one. By comparing this ratio to appropriately predetermined threshold, we can roughly judge in which class each performance data belongs, and by regarding this result as initial classification, values of the parameters can be estimated by the linear least-squares method. Once the parameter values are approximately obtained, the classification can be updated by another criterion: which of the two estimating function T_{TH} and T_{TL} each sample of performance data is near to. Repeating this cycle of parameter estimation and classification a few times, better approximation of values of parameters can be obtained.

4 Experiments and Results

In this chapter, we show some results of estimation of grid execution time, and efficiency of block size optimization based on this estimation.

Here, we used two GPUs: an old one called Quadro FX 4800, and a relatively new one called Tesla C2075, which is based on Fermi architecture. Quadro FX 4800 has 24 MPs, each of which has 16384 bytes of shared memory and can execute at most 1024 threads in parallel, while Tesla C2075 has 14 MPs and 49152 bytes of shared memory at each, and the maximum number of active threads per MP is 1536.

We measured the grid execution time varying the block size W from 32 to 512 on multiples of 32, H from 16 to $(1024 - W)$ on multiples of 16, and the total number of blocks B_{total} from 1 to 504. Then, we estimated parameter values of the model of grid execution time by the method introduced in Section 3, and compared estimated time by the model with the measured value.

The actual grid execution time and estimated one on some H from 160 to 384 at step 32 when W was 32 are shown in Figures 6 - 9. In these graphs, the horizontal axis corresponds to the number of thread blocks B_{total} while the vertical axis corresponds to the execution time on the millisecond time scale.

When W is 32, the number of active threads per MP is limited by the restriction of that of active blocks per MP, so in this example the latency of the device memory access was not fully hidden. Therefore, the execution time jumped at points where B_{total} exceeded multiples of $B_{act} \cdot N_{MP}$, which corresponds to the left case in Figure 5. Broadly speaking, the graph of estimated time reproduced that of actual time well.

However, they have some errors in detail. Figure 10 and Figure 11 show the ratio of estimated grid execution time to actual one when W was 32, with B_{total} on the horizontal axis. In both graphs, periodical waves were observed, but they are qualitatively different. As Figure 6 shows, the slow increment of time, represented by T_{BS} in the previous chapter, is almost the same independent of the range of the number of thread blocks on Quadro FX 4800, so its error is

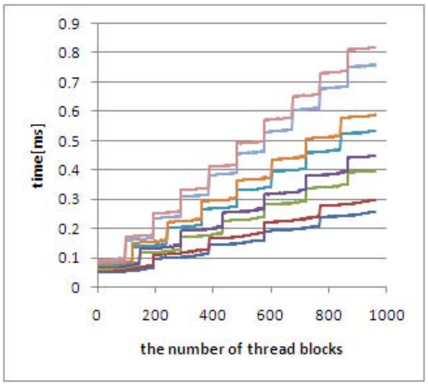


Fig. 6. Actual grid execution time ($W = 32$, Quadro FX 4800)

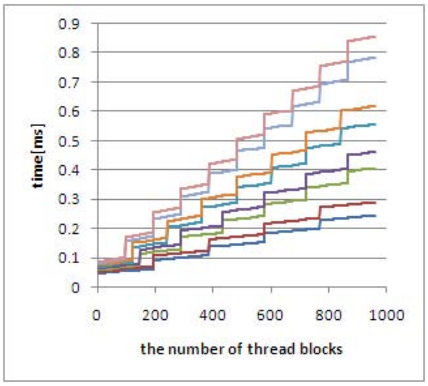


Fig. 7. Estimated grid execution time ($W = 32$, Quadro FX 4800)

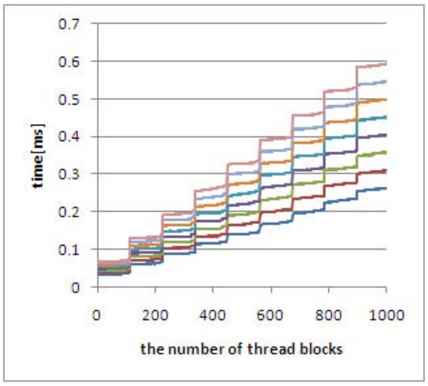


Fig. 8. Actual grid execution time ($W = 32$, Tesla C2075)

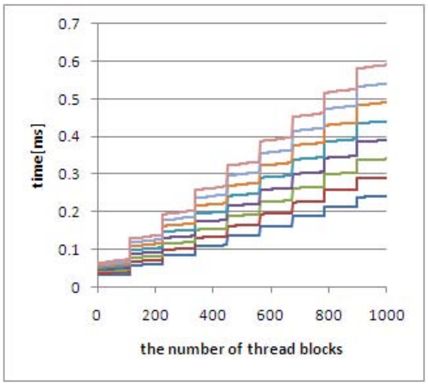


Fig. 9. Estimated grid execution time ($W = 32$, Tesla C2075)

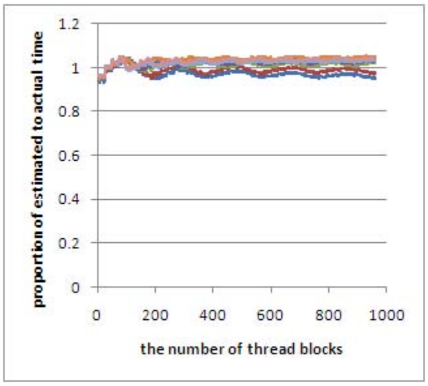


Fig. 10. Approximation ratio ($W = 32$, Quadro FX 4800)

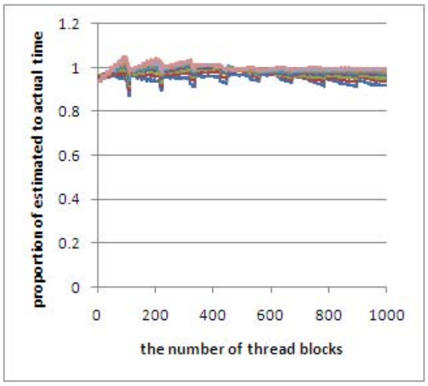


Fig. 11. Approximation ratio ($W = 32$, Tesla C2075)

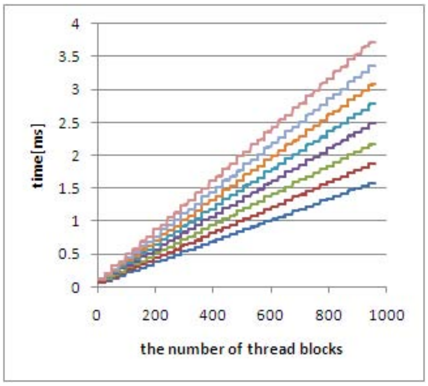


Fig. 12. Actual grid execution time ($W = 256$, Quadro FX 4800)

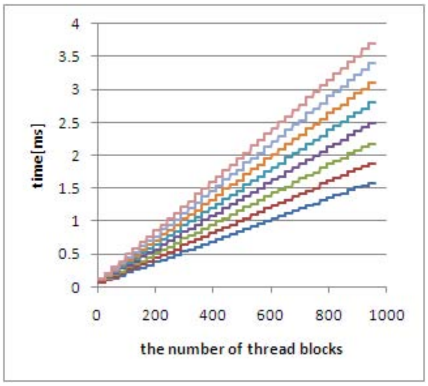


Fig. 13. Estimated grid execution time ($W = 256$, Quadro FX 4800)

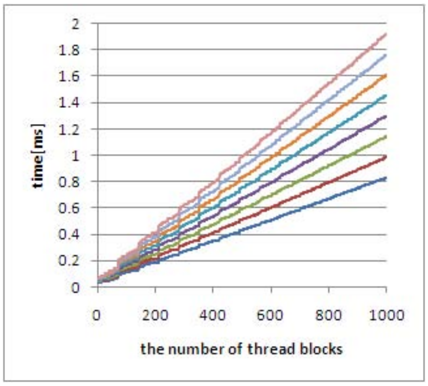


Fig. 14. Actual grid execution time ($W = 256$, Tesla C2075)

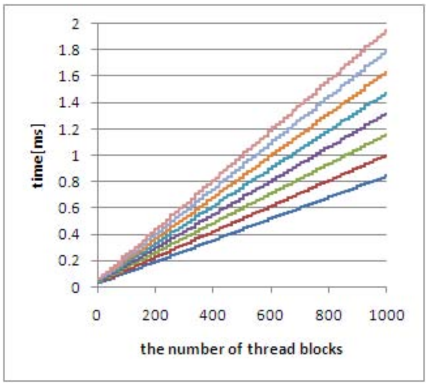


Fig. 15. Estimated grid execution time ($W = 256$, Tesla C2075)

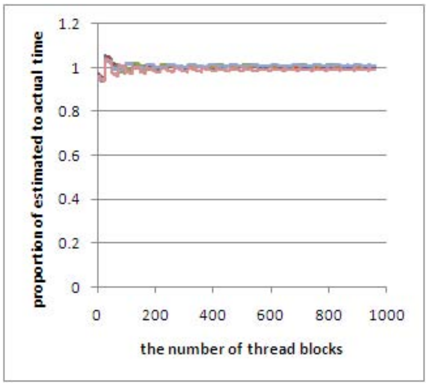


Fig. 16. Approximation ratio ($W = 256$, Quadro FX 4800)

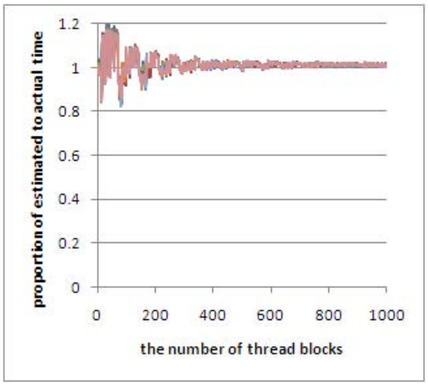


Fig. 17. Approximation ratio ($W = 256$, Tesla C2075)

reduced by refining the approximation of T_{BS}. In Figure 8, on the other hand, the slow increment when the number of thread blocks is from 1 to 104 and that when it is from 937 to 1040 is visibly different. It seems to be caused by adoption of device memory cache and development of scheduling system on Tesla C2075. Therefore, in order to improve the approximation accuracy on new GPUs, it is indispensable to revise our model itself which approximate the execution time by the sum of the slow increment T_{BS} and the rapid increment T_{DH} or T_{DL}.

Figures 12 - 15 show the actual and estimated time of grid execution respectively when W was 256. In this case, the number of active threads per MP was so large that the latency of the device memory access was fully hidden by instruction execution of other threads. Therefore, the execution time constantly increased every N_{MP} threads, which corresponds to the right case in Figure 5.

Figure 16 and Figure 17 show the ratio of estimated grid execution time to actual one when W was 256.

Just as the case when W was 32, there were some errors in T_{BS} when the number of blocks is small. In other points, however, the model accurately approximated the execution time, and the approximation error was within about 3% of the actual value except for the case with too small number of thread blocks.

The results of experiments suggests that most of the approximation error in our model was in T_{BS} and T_{DL}, that is to say, the approximation in case when the latency was not fully hidden. Our model puts emphasis simply on the number of instructions and the effect of latency hiding of the device memory access because they have the most powerful influence on the performance, but there are some other factors we did not considered. For example, it is officially informed that registers of GPU are also source of generating delays, and it is also hidden by securing sufficient number of active threads. Moreover, as mentioned above, the influence of device memory cache is not negligible on new machines. Our model cannot adapt to the two-stage bends of performance curve caused by latency hiding of both the device memory and registers, which may be a cause of the approximation error. This remains to be a future work.

5 Conclusion

In this research, we parallelized the dynamic programming algorithm for calculating edit distance on GPU. In GPU Computing, the cost of the device memory access is in many cases a primary factor to slow down the performance because of its high latency, so it is important to utilize the shared memory as a cache and to hide the latency by meantime-executed arithmetic instructions, in addition to merely reducing the number of access itself. Besides, load balancing is also important in order to exploit the full performance of the GPU's vast computing resources.

Considering these facts, we suggested a blocking algorithm, and constructed a model to estimate grid execution time for the purpose of optimizing block size. In our model, especially taking into account the latency of memory reference instructions, we expressed the relationship between the number of active threads, the block size, and the grid execution time. By selecting the block size to minimize the total computation time obtained from this model, we tried to find optimum trade-off between load balancing and the cost of memory access and other extra processing accompanied with the block splitting. Consequently, we succeeded in automatically selecting nearly optimum block size in terms of computation time.

Our model is specific to the dynamic programming algorithms, but there are several problems which have similar data dependency, the Smith-Waterman algorithm, SOR method, preprocessing of ICCG method, and so forth. Therefore, we think our model has some application range. In the edit distance algorithm, the number of active threads is determined almost exclusively by the length of strings, but other algorithms usually consume more GPU resources, and this may limit the number of active threads. In such cases, it is indispensable to consider the effect of latency hiding as done in our work. Application of the model for grid execution time to more complicated algorithms is one of the challenges for the future, as well as improvement in accuracy of the current model and its adaptation to new GPUs.

Acknowledgements. This work is partially supported by CREST project "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies", JST.

References

1. Dohi, K., Benkrid, K., Ling, C., Hamada, T., Shibata, Y.: Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs. In: ASAP, pp. 29–36 (2010)
2. Ling, C., Benkrid, K., Hamada, T.: A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In: 2009 IEEE 7th Symposium on Application Specific Processors, pp. 94–100 (2009)
3. Liu, Y., Huang, W., Johnson, J., Vaidya, S.: GPU accelerated Smith-Waterman. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3994, pp. 188–195. Springer, Heidelberg (2006)
4. Liu, Y., Maskell, D.L., Schmidt, B.: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Research Notes 2(1), 73 (2009)
5. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(suppl. 2), S10 (2008)
6. Munekawa, Y., Ino, F., Hagihara, K.: Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU. In: BIBE, pp. 1–6 (2008)
7. NVIDIA Corporation. NVIDIA CUDA C Programming Guide Version 4.0