ISSN 2303-498X

XIII INTERNATIONAL CONFERENCE ON SOCIAL AND TECHNOLOGICAL DEVELOPMENT

PROCEEDINGS

XIII MEĐUNARODNA KONFERENCIJA O DRUŠTVENOM I TEHNOLOŠKOM RAZVOJU

ZBORNIK RADOVA

Trebinje, June, 06-09, 2024
Trebinje, 6 - 9. juni 2024. godine

ISSN 2303-498X

XIII INTERNATIONAL CONFERENCE ON SOCIAL AND TECHNOLOGICAL DEVELOPMENT

PROCEEDINGS

XIII MEĐUNARODNA KONFERENCIJA O DRUŠTVENOM I TEHNOLOŠKOM RAZVOJU

ZBORNIK RADOVA

Trebinje, June, 06-09, 2024
Trebinje, 6 - 9. juni 2024. godine

Co-organizers:

UNIVERSITY of DEBRECEN

Univerza v Mariboru
Fakulteta za logistiko

TECHNICAL UNIVERSITY OF KOŠICE
Faculty of Manufacturing Technologies

AUE FON
AMERICAN UNIVERSITY OF EUROPE

АКАДЕМИЈА СТРУКОВНИХ СТУДИЈА

VELEUČILIŠTE VIMAL
University of Applied Sciences

УДРУЖЕЊЕ ИНЖЕЊЕРА ТЕХНОЛОГИЈЕ
УИТРС
РЕПУБЛИКЕ СРПСКЕ

EUMMAS

VUB
VISOKA ŠKOLA ZA USLUŽNI BIZNIS

XIII INTERNATIONAL CONFERENCE ON SOCIAL AND TECHNOLOGICAL DEVELOPMENT
XIII MEĐUNARODNA KONFERENCIJA O DRUŠTVENOM I TEHNOLOŠKOM RAZVOJU

PROCEEDINGS

Publisher:
University PIM, Banja Luka, Republic of Srpska, Bosnia and Herzegovina

For publisher
Dejan Kojić, PhD, General Manager

Editorial board:
Dejan Kojić, PhD, Vice-Rector for Science
Darjana Sredić, PhD.

Design and Computer processing:
Ljubica Janjetović, PhD

Print:
„Vilux“ Banja Luka

Circulation:
25 Cipies

CIP - Каталогизација у публикацији
Народна и универзитетска библиотека
Републике Српске, Бања Лука

33:005(082)
62(082)

МЕЂУНАРОДНА конференција о друштвеном и технолошком развоју (13 ; 2024 ; Требиње)
Zbornik radova / XIII Međunarodna konferencija o društvenom i tehnološkom razvoju, Trebinje, 6-9. juni 2024. godine = Proceedings / XIII International conference on social and technological development, Trebinje, June, 06-09, 2024 ; [editorial board Dejan Kojić, Darjana Sredić]. - Banja Luka : Univerzitet za poslovni inženjering i menadžment = University PIM, 2024 (Banja Luka : Vilux). - 608 стр. : илустр. ; 25 cm. - (Međunarodna konferencija o društvenom i tehnološkom razvoju = International conference on social and technological development, ISSN 2303-498X)

Радови на срп. и енгл. језику. - На кор: Sted. - Тираж 25. - Напомене и библиографске референце уз текст. - Библиографија уз сваки рад. - Abstracts.

ISBN 978-99955-40-79-1

COBISS.RS-ID 141600769

Zbornik radova sa međunarodne naučne konferencije o društvenom i tehnološkom razvoju – STED, publikuje se uz finansijsku podršku Ministarstva za naučnotehnološki razvoj i visoko obrazovanje u Vladi Republike Srpske.

Proceedings from the International scientific conference on social and technological development - STED, is published with the financial support of the Ministry of Scientific-Technological Development and Higher Education in the Government of the Republic of Srpska.

NOTE:

The authors have full responsibility for the originality and content of their own papers

CONTENT
SADRŽAJ

DOI 10.7251/STEDZ2401318B

KRATKI NAUČNI RAD - SHORT SCIENTIFIC PAPER

LEARN MORE ABOUT PARALLEL PROGRAMMING TECHNIQUES BY IMPLEMENTING A HISTOGRAM ON GPU

Dushan Bikov[1], Limonka Koceva Lazarova[1], Zoran Panov[2]

[1]Goce Delcev University, Faculty of Computer Science, Krste Misirkov 10-A, 2000 Stip, Macedonia, dusan.bikov@ugd.edu.mk, limonka.lazarova@ugd.edu.mk
[2]Goce Delcev University, Faculty of Natural and Technical Sciences, Krste Misirkov 10-A, 2000 Stip, Macedonia, zoran.panov@ugd.edu.mk

ABSTRACT

A histogram is a graphical representation of the distribution of data, commonly used as an analysis tool in statistics to visualize the frequency distribution of a dataset. Our focus of interest lies in the Frequency Histogram, a special graph that employs vertical columns to illustrate the frequency of occurrence of each data element.

Here, we will present a parallel SIMT (Single Instruction Multiple Threads) Histogram algorithm. This algorithm is composed of basic parallel primitives like sorting and parallel compact. Implementing in this manner for educational endeavors allows students to delve into the details, learn, and enhance their understanding of the CUDA parallel programming model. The presented algorithm utilizes a sparse representation of the vector to store results, distinguishing it from well-known standard parallel Histogram algorithms. Besides delving into the details of implementing parallel primitives, we will also discuss how to perform an efficiency evaluation of the parallel program.

Keywords: Histogram, Parallel primitives, Parallel programing, CUDA, GPU.

INTRODUCTION

The modern parallel programming model is gradually replacing the traditional sequential model due to the growing demand for extensive computing in applications, driving the shift towards parallel computing. Almost all scientific and research areas are impacted to some degree by this trend. Many problems with mathematical backgrounds, applicable in various fields such as cryptography, coding theory, and data processing, necessitate massive and efficient computation. Parallel computing facilitates large-scale and efficient data processing to address these challenges.

The modern trend in computer hardware technology unmistakably leans towards a massive parallel hardware resource structure. This implies a growing demand for experts who understand how to harness these resources. In other words, there will be a requirement for programmers to be familiar with the concepts of parallel programming and to adeptly apply them.

Our intention here is to present the implementation of a parallel Histogram algorithm, composed of several basic parallel primitives, for educational purposes. Through the algorithm's implementation, readers can learn and grasp the fundamental principles of parallel programming. While sequential histogram computation is trivial, achieving efficient parallel computation on the GPU is traditionally challenging. The presented parallel histogram algorithm exhibits certain and significant differences from the general parallel algorithm approach. The primary parallel operations on which this algorithm relies are sorting and parallel compact. Compact itself is composed of fundamental building blocks such as predicate and exclusive scan, and all mentioned building blocks play a significant role in constructing various parallel algorithms.

Histograms are one of the most common operations and serve as fundamental building blocks for data mining applications and multimedia analysis tasks, including filtering (Porikli 2008), classification (Tang et al., 2012), object detection (Palaniappan et al., 2016), visual tracking (Yao et al., 2016), etc. The histogram algorithm counts the number of data elements (observations) that

fall into predefined categories known as bins. The input data set and the number of categories depend on the specific domain or field of use. Here, a scenario is presented with a large number of bins ($0 \leq 2^n$, $n=\mathbb{Z}$), while the general case of a histogram involves a potentially extensive range of bins ($-2^n \leq i \leq 2^n$, $n=\mathbb{Z}$). It is not necessary for all bins to contain values, and the number of possible bins depends on the specific case (input dataset). In our use case for the input dataset, we generate random integer numbers.

The histogram algorithm is implemented in the parallel programming model CUDA (Compute Unified Device Architecture) (CUDA Zone, 2024), developed by NVIDIA for its GPUs (Graphics Processing Units) intended for general-purpose computing. Modern GPUs are highly efficient and powerful processing devices with a massive parallel structure, higher memory bandwidth, and processing capability. This enables the implementation of very efficient algorithms that can simultaneously process large amounts of data, making them attractive for scientific computing applications. The CUDA platform allows direct interaction with GPU resources, facilitating the effective utilization of computational power.

The paper follows this organization: the subsequent section presents a review of work related to histogram algorithms. It then explores the general principles of the parallel computing model, followed by the presentation of details about the algorithm implementation and a discussion on performance evaluation. The paper concludes with a summary.

WORK RELATED TO HISTOGRAM

In this section, we will briefly discuss work related to histogram algorithms. The implementation of the sequential Histogram algorithm is trivial, but an effective parallel GPU implementation proves to be quite challenging. Various general parallel approaches exist for Histogram computation, and they are closely correlated with factors such as the application field, input dataset, number of bins, hardware limitations, etc.

The simplest approach involves the use of global atomic operations, but resolving collisions between threads can be very expensive. Another group of approaches employs different types of memory (local and shared), implying a parallel per-thread (local or) sub-histogram, which is then reduced (local histograms) into a final histogram (Nugteren et al. 2011; Podlozhnyuk 2007). However, this group of approaches has obvious limitations stemming from memory resources and results in a small number of histogram bins.

There is a group of approaches with different structures, each containing one or more parallel primitive blocks (or algorithms) with additional operations and functions. The first approach within this group (Shams et al. 2010), where they propose a method that involves counting while sorting the input data. Another approach, similar to the previous one, borrows elements from the algorithm used for computing the weight spectrum of binary linear codes (Pashinska et al. 2020). In this approach, the data is first sorted, and then an 'array W' is generated, from which the weight spectrum of binary linear codes, essentially a histogram, can be extracted. Another approach (Udacity courses, 2024) first involves sorting and then reducing by key.

CUDA PROGRAMING MODEL

The CUDA programming model, created by NVIDIA for general-purpose computing on GPUs (CUDA C Programming Guide 2024), is characterized by several key aspects. At its core is the Streaming Multiprocessor (SM), which houses Single-Instruction Multiple-Threads (SIMT) cores. These cores execute the same instructions simultaneously but with different data, enabling massive parallel processing. GPUs, designed for throughput, can handle thousands of threads simultaneously, organized into groups of 32 called "warps." These warps execute in lockstep, with execution alternating between active and temporarily inactive warps.

Threads within warps are organized into blocks, and a thread block may contain up to 1024 threads. Threads in a block can cooperate through shared memory and synchronize their execution. CUDA programmers define functions, known as kernels, which are executed N times in parallel by N different CUDA threads. The kernel invocation is specified using the syntax:

mykernel<<<BlocksPerGrid, threadsPerBlock>>> (...); The global index (tID) for each thread is computed using the formula: tID = threadIdx.x + blockIdx.x * blockDim.x;

The number of threadsPerBlock and BlocksPerGrid determines the number of threads (a grid of threads) executing the kernel. Blocks within the grid and threads within a block are identified using indices like 'blockIdx.x', 'blockDim.x', and 'threadIdx.x'. This indexing allows natural computation across elements in a domain.

The GPU memory hierarchy includes global, shared, local, constant, and texture memory. Global memory has the largest capacity but the slowest speed. Shared memory is faster and accessible by all threads within a block. Each thread has private local memory.

At the application level, a master process handles tasks like memory operations, kernel launches, and result retrieval, optimizing data flow between main memory and GPU global memory.

PARALLEL HISTOGRAM ALGORITHM

In this section, we present the details of the parallel histogram algorithm. The algorithm consists of two main parts: first, sorting the input array *In* and aggregating it into an output *Structure of Arrays* (*SoA*) that defines the histogram distribution; second, using the parallel primitive *Compact*. The *SoA* stores data fields in separate arrays, optimizing parallel processing, memory access, and bandwidth use. *SoA* can enhance memory coalescing and cache utilization, leading to improved performance in parallel computations.

The input data is an array of integers representing random integer values '*In*' (first row in Fig. 1) and is in device memory. The output of the parallel computations on the GPU is a *SoA* with *SoA.value* and *SoA.index* arrays, shown in sparse vector representation (Fig. 1). This form was chosen to minimize data transfer from the GPU to the CPU. Additionally, the Structure of Arrays allows for more efficient use of bandwidth compared to Array of Structures and often results in better memory coalescing access. From the SoA, the histogram distribution can be extracted, and the number *N* represents the length of the *SoA*. Algorithm 1 (Fig. 1) outlines the steps for calculating the histogram for a given dataset.

Algorithm 1. Calculating Histogram
Input: Input array *In*, random integer numbers, with 2^n entries
Output: Array SoA of length *N*, Histogram *H*
// The *In* array it's in device memory
Step 1. (parallel kernels on GPU) Sorting
Step 2. (parallel kernels on GPU) Compact
Step 2.1. (parallel kernels on GPU) Predicate
Step 2.2. (parallel kernels on GPU) Exclusive Scan
Step 2.3. (parallel kernels on GPU) Mapping (compact)
// Transfer N number to CPU memory
// Transfer SoA array to CPU memory
Step 3. (process on CPU) Calculating Histogram *H*

There is no predefined number of bins here. This means that Algorithm 1 compresses the output data to occupy only the necessary bins. Below, I will describe in more detail the steps of Algorithm 1.

In the first step, it is necessary to sort an input array (*In*) with a length of 2^n. For sorting the input array, the parallel sorting primitive from the Thrust Library (Thrust Library, 2024) is used. Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). To use the sorting primitive from the Thrust library, device memory dispatching is required (Thrust Library 2024). As shown in Figure 1 (second row), the sorting operation is the first one performed. A Sort device array is defined and allocated to store the sorted input elements.

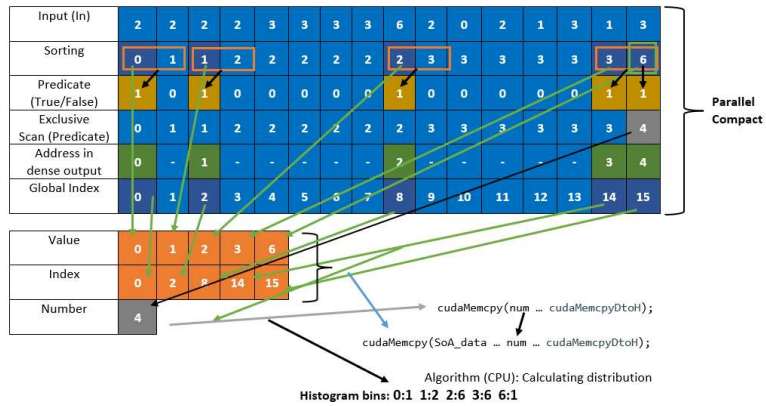

Figure 1. Parallel Algorithm for computing histogram.

Compaction can be used when there is a large amount of data, and you only want to perform some computation on a subset of that data. Compaction represents a predicate function that removes elements returning false and 'squeezes' the data into the 'required space'. The fundamental building subroutines for compaction involve defining a predicate function and performing an Exclusive Scan.

The Predicate sub-step aids in generating the SoA array from the array containing the sorted input dataset. A Predicate is a function that takes one element as an input parameter and returns either true or false. To define the predicate, the sorted input array is examined by activating 2^n threads. Each thread compares the data from the sorted array at positions tID and tID + 1. If the two elements are different, the thread at tID sets the value to 1 in the Predicate array (device-allocated) at position Predicate[tID] (Fig. 1). The last thread does not make any comparisons, as the predicate value determines the number N, which is used to define the length of the SoA array.

A simple and commonly used parallel primitive building block is the all-prefix-sums (scan) operation. Prefix sums are trivial to compute in sequential models of computation, using the formula $y_i = y_{i-1} + x_i$ (for an array of n elements $[x_0, x_1, ..., x_{n-1}]$) to compute each output value in sequential order. Despite their ease of computation, prefix sums provide solutions for a set of tasks that seem challenging for parallel implementation. This operation is a useful primitive and is employed as a subroutine in other parallel algorithms, such as sorting, string comparison, lexical analysis, stream compaction, polynomial evaluation, histograms, etc.

The scan operation may be either inclusive or exclusive. In the exclusive scan, each element *j* of the result is the sum of all elements up to but not including *j* from the input array. In an inclusive scan, all elements including *j* are summed. As shown in Figure 1, the exclusive scan is the main building block of our parallel compact primitive. For the exclusive scan, the input is the *Predicate* array, and the result of this operation is stored in another device memory array (Scan). Here, the 'Scan-Scan-Add' approach is used, and there is room for improvement if the 'Reduce-then-Scan' approach is employed (Harris 2010).

The final step computed by the GPU is the *mapping* (*compact*) operation, which generates the *Structure of Arrays* (*SoA*) from the array (*Sort*) containing the sorted input dataset, the array (*Predicate*) containing the predicate, and the array (*Scan*) containing the prefix sums elements of the *Predicate* array. As mentioned, the *SoA* contains *SoA.value*, which correlates with the *Sort* array, and *SoA.index*, which correlates with *Scan* (Fig. 1). The Predicate array plays the main role during the mapping process. Each thread evaluates the predicate and copies only if it is true. In other words, each thread evaluates the predicate at positions *tID* (*Predicate[tID]*), and if it is set to 1, it performs the mapping between the arrays *Sort* and *Scan* to the Structure of Arrays (*SoA*). This squeezing of the output data optimizes the data transfer from GPU global memory to CPU memory.

Before performing the last step (Step 3 in Algorithm 1), it is first necessary to transfer the data (*SoA*) from GPU global memory to CPU memory. This step of the algorithm computes the

histogram distribution *H*. The master process initially copies the value *N*, which defines the length of the *SoA* array. Following that, the master process copies the *SoA* array from GPU global memory to CPU memory. Extracting the histogram from the *SoA* is followed by the calculation of the *SoA* array using Algorithm 2.

Algorithm 2. Calculating histogram *H(i)*

```
Input: Array SoA
Output: Histogram H(i)
  temp ← 0;
  for t ← 0 to N do
    H(i).value[t] = SoA.value[t];
    H(i).number[t] = (SoA.index[t] + 1) − temp;
    temp ← SoA.index[t] + 1;
```

The variable H(i) also represents a Structure of Arrays with *H(i).value* array and *H(i).number* array. The *H(i).value* array stores the input values '*i*,' and the *H(i).number* array stores the number of input values with the value '*i*.' The variable '*temp*' is used to save the *SoA.index[t]*, and to this, 1 is added for the previous '*t*.' This variable 'temp' is then utilized to calculate the number of input values.

Exploring the implementation of the parallel histogram algorithm helps readers understand fundamental principles of parallel programming. It relies on sorting and parallel compaction operations, utilizing key building blocks such as predicate and exclusive scan, which are crucial for various algorithms.

DISCUSSION ABOUT PERFORMANCE EVALUATION

Performance evaluation on GPUs involves benchmarking, profiling, and algorithm analysis, comparing speedup against sequential implementations. This includes optimizing memory access patterns, reducing data transfer overhead, and tuning kernel configurations. Comparative studies with sequential (CPU) histogram algorithm implementations help justify GPU adoption, while scalability analysis and energy efficiency considerations are crucial for optimizing GPU-accelerated solutions. The sequential algorithm is implemented in C++. The computer hardware that was used is shown in Table 1.

Table 1. Computer hardware description.

Hardware	Specifications
CPU	i7-12700F, 2.10 GHz
Memory	32GB DDR4 4400 MHz
GPU	GeForce RTX 3060 Ti
OS	Windows 11, 64-bit
IDE/ Compiler	MS Visual Studio 2019
CUDA SDK	12.1
GPU Driver	V 555.85

The CUDA programming model yields better computational performance with larger datasets. Our experimental evaluation is performed on randomly generated integer datasets. Random dataset values were generated and used as input for the Histogram Algorithm. To measure the execution time, timers are strategically placed at suitable positions. The input dataset consists of randomly generated integer arrays with 2^n entries ($n = 10, 12, 14, 16, 18, 20$). The generated dataset is already stored in global memory, and the execution time includes Algorithm 1 parallel kernel computation. The time spent on Algorithm 2 is insignificant.

Considering the possible number of bins ($0 \leq 2^n$, $n=\mathbb{Z}$) and the different scenarios examined, the GPU execution time remains roughly the same regardless of the number of bins. The scenario with $2^n/4$ bins is presented in Table 2.

The main criterion for performance validation is the acceleration of parallel versus sequential implementation, defined as speedup by the formula:

$$S_P = \frac{T_{(1,n)}}{T_{p(n)}}$$

where n is the size of the input data, $T_{(1,n)}$ is the execution time of the sequential algorithm, and $T_{p(n)}$ is the execution time of the parallel algorithm. The speedup of the parallel algorithm is shown in the 'Speed up' column. In the following experiments, GPU time includes both computation and data transfer between device global memory and main memory. Since the purpose of this algorithm is to find the histogram of pre-generated data, the transfer time between main memory and global memory is not considered.

Table 2. Experimental evaluation of CPU versus GPU

size	CPU (ms)	GPU (ms)	Speed up: CPU vs. GPU	#bins
2^{10}	0.002	0.034	/	$2^{10}/4 = 256$
2^{12}	0.016	0.038	/	$2^{12}/4 = 1024$
2^{14}	0.039	0.105	/	$2^{14}/4 = 4096$
2^{16}	0.380	0.226	1.68x	$2^{16}/4 = 16384$
2^{18}	0.712	0.259	2.74x	$2^{18}/4 = 65536$
2^{20}	1.666	0.575	2.89x	$2^{20}/4 = 262144$

The implemented parallel histogram is not optimal and can be improved, stemming from the use of basic parallel primitives, resulting in a deterioration of overall performance directly related to synchronization issues. Additionally, the number of threads per block and the memory pattern employed influence and contribute to the slowdown in execution time. Conversely, from an educational standpoint, our intention was to delve into the details and enhance readers' understanding of the CUDA parallel programming model by implementing the histogram using a few parallel primitives.

CONCLUSIONS

Here, we introduce a histogram algorithm constructed with widely recognized parallel primitives, with a primary emphasis on educational purposes. It provides detailed insights into the fundamental principles of parallel programming. The discussion on performance evaluation emphasizes the importance of enhancing performance compared to sequential implementations. Further optimization can be achieved by combining and adjusting certain sub-steps, leading to increased efficiency and better performance.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of Goce Delcev University Stip, Macedonia, through the individual scientific fund.

LITERATURE

CUDA Zone. (2024). Report. Retrieved May 10, 2024, from https://developer.nvidia.com/cuda-zone.

CUDA C Programming Guide (2024). Report. Retrieved May 10, 2024, from https://docs.nvidia.com/ cuda/cuda-c-programming-guide/.

Harris, M. (2010). State of the Art in GPU Data-Parallel Algorithm Primitives. Tech. Rep. *In Proceedings GPU Technology Conference.*

Nugteren, C., van den Braak, G. J., Corporaal, H., & Mesman, B. (2011). High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs. *In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (pp. 1-8).

Pashinska, M., & Bouyukliev, I. (2020). A parallel algorithm for computing the weight spectrum of binary linear codes. *In Proceedings 2020 Algebraic and Combinatorial Coding Theory (ACCT)* (pp. 1-5). IEEE.

Palaniappan, K., Poostchi, M., Aliakbarpour, H., Viguier, R., Fraser, J., Bunyak, F., Basharat, A., Suddarth, S., Blasch, E., Rao, R., & Seetharaman, G. (2016). Moving object detection for vehicle tracking in wide area motion imagery using 4d filtering. *In Proceedings 2016 23rd International Conference on Pattern Recognition* (ICPR) (2830-2835). IEEE.

Podlozhnyuk, V. (2007). Histogram calculation in CUDA. White paper. Retrieved May 10, 2011, from https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf

Porikli, F. (2008). Constant time o (1) bilateral filtering. In Proceedings IEEE CVPR, 1–8.

Tang, S., Zheng, Y.-T. Wang, Y., & Chua, T.-S. (2012). Sparse ensemble learning for concept detection. *IEEE Transactions on Multimedia, 14*(1), 43–54.

Shams, R., Sadeghi, P., Kennedy, R., & Hartley, R. (2010). Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. *Computer methods and programs in biomedicine, 99*(2), 133-146.

Thrust library. (2024). the CUDA C++ template library. Retrieved May 10, 2024, from https://docs.nvidia.com/cuda/thrust/index.html.

Udacity courses. (2024). CS344: Intro to Parallel Programming. Retrieved May 10, 2024, from https://developer.nvidia.com/udacity-cs344-intro-parallel-programming.

Yao, R., Xia, S., Zhang, Z., & Zhang, Y. (2016). Real-time correlation filter tracking by efficient dense belief propagation with structure preserving. *IEEE Transactions on Multimedia, 19*(4), 772-784.