2012 Cairo International Biomedical Engineering Conference (CIBEC)
Cairo, Egypt, December 20-21, 2012

High Performance CUDA-based Implementation for the 2D Version of the Maximum Subarray Problem (MSP)

Salah Saleh[1,2 †], Marwan Abdellah[1,3 ‡], Ahmed A. Abdel Raouf[1 *] and Yasser M. Kadah[1 ¶]

[1]Systems & Biomedical Engineering Department, Faculty of Engineering, Cairo University
[2]Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
[3]Ecole Polytechnique Fédérale de Lausanne (EPFL)
[†]*salah.m.saleh@studium.fau.de*, [‡]*marwan.abdellah@epfl.ch*
[*]*ahmed.abdelsalam@medstreaming.com*, [¶]*ymk@k-space.org*

***Abstract*— The Maximum Subarray Problem (MSP) finds a segment of an array that has the maximum summation over all the other possible combinations. Different applications for this problem exist in various fields like genomic sequence analysis, data mining and computer vision. Several optimum linear-time solutions exist for the 1D version, however, the known upper bounds for the 2D version are cubic or near-cubic time; which makes it a problem of high complexity. In this work, a stage by stage high performance Graphics Processing Unit (GPU)-based implementation for solving the 2D version of the problem in a linear time relying on the Compute Unified Device Architecture (CUDA) technology is presented. It achieves more than $7X$ of speed-up in performance compared to a single-threaded sequential implementation on the Central Processing Unit (CPU) for an array of size 512^2.**

***Keywords*—Maximum Subarray, Prefix Sum, GPU Implementation, CUDA, Speedup**.

I. Introduction

The Maximum Subarray Problem (MSP) problem is concerned about retrieving a segment of consecutive array elements that has the maximum summation over all the other possible combinations in this array [1]. If the array elements are of only positive numbers, then the maximum summation in such case will be the entire array; therefore the MSP always deals with arrays of positive and negative numbers. This problem has significant projection in several applications in the medical field such as genomic sequence analysis at which it predicts the membrane topology of proteins leading to understanding their functions; which is an essential step for the development of antibodies and drugs [2]. In computer vision, this problem is employed in finding the brightest region of an image; which can be used later for image analysis. As a practical application in the medical field, we use this technique in this work for the detection of macrocalcifications in mammographic images. The algorithm has been applied on mammographic images of mini-MIAS database [3] twice. The first time, it considers only direct application of the algorithm on the images without any preprocessing. In the second time, it considers the removal of the bright regions of the chest from the images. Additionally, we subtracted different pivot values from the elements of the image to investigate the possible improvement in detection of masses. This can be illustrated by Fig. 1.

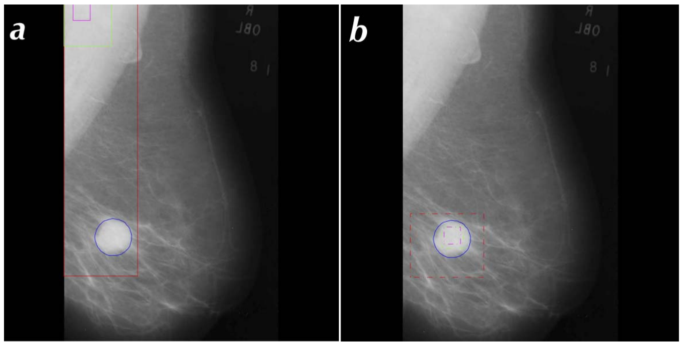

Figure 1. In (a), the MSP algorithm was applied on a sample mammographic image without any preprocessing, while in (b) the chest region was eliminated before applying the algorithm. The blue circles represent the exact location of the macrocalcification while the colored rectangles represent the application of the algorithm during the subtraction of different pivot values.

However, this problem has an unbalanced workload distribution between its parallel running threads, but it has a high degree of parallelism that makes it fitting the GPU nature. In this work, we present a high performance implementation for solving the 2D version of this problem on CUDA-enabled GPUs exploiting their powerful underlying computing architecture and also their high memory bandwidth.

II. Previous Work

In 1977, the MSP was firstly investigated by Ulf Grenander when he was trying to find the region of the maximum summation over all rectangular regions of a given *mxn* array of real numbers to use it as an estimator of a certain kind of pattern in a digitized picture [5]. Later, the problem was simplified to the 1D version in order to have a better understanding of it. Meanwhile, Kadane was able to give a linear-time algorithm to the problem [6]. By extending Kadane's Algorithm to work on the 2D version, a time complexity of O(N^3) was obtained which is highly computationally expensive solution [7]. The development of parallel algorithms was remarkably started by Wen [8] presenting a parallel algorithm for the 1D version of the problem running in $O(logN)$ time using $O(N/logN)$ processors on the EREW PRAM (Exclusive Read, Exclusive Write Parallel Random Access Machine) and a similar result was given by Perumalla and Deo [9]. Qiu and Akl used interconnection networks of size p, and achieved O(N/p + log p) time complexity for solving the 1D version, and $O(logN)$ complexity with $O(N^3/logN)$ processors for the 2D version [10].

III. GPU & CUDA

The accelerated evolution of the GPUs and their software application programming interfaces (APIs) has turned them out to be high performance parallel platforms for solving highly complex problems and algorithms. They are dedicated to process graphics operations no more. In their modern architectures, GPUs are designed to process huge number of generic floating point operations in parallel based on their huge amount of parallel cores and their high memory bandwidth. As well as, their APIs granted the user direct, abstract and flexible access to their underlying powerful architecture without deep knowledge of their complex designs. CUDA is a general-purpose GPU architecture that delivers a different parallel programming paradigm and novel instruction set architecture; which enable the users leveraging the high computing power of the core computing engine embedded in the CUDA-enabled GPUs for addressing complex scientific and engineering applications [11]. Fig. 2 illustrates the CUDA architecture hardware model.

IV. Algorithm

In [12], a detailed algorithm for solving the 2D version of the problem was presented. This algorithm is shown in Algorithm 1. Based on the work discussed in [7], we slightly modifying the algorithm trying to make it efficiently parallelizable for mapping it on the GPU. The main idea behind using this technique is to transform the 2D problem into a 1D like problem. So if we have an input matrix 'A' (see Fig. 3), we fix *g* and *i* which represent the starting and ending row indices of the 2D strip respectively, and then try to find *h* and *j* which respectively represent the starting and ending column indices so that the sum bounded by them; *m(g, h)|(i, j)* is maximized. Note that if we have *mxn* matrix, then we have $m(m+1)/2$ possible combination of *g* & *i*.

V. Implementation

On the implementation side, the algorithm in general can be divided into three consecutive stages: *prefix sum calculations*, *finding the maximum summation of each strip* and *finally the reduction of the maximum summations into the maximum value that represents the maximum of all possible combinations of the input two-dimensional matrix*. Our strategy considered implementing a serial version of each stage on the CPU followed by having this stage mapped to the GPU as a separate kernel to have eventually three subsequent kernel units constructing the entire pipeline. Having this pipeline successfully constructed, we then benchmark the entire pipeline on each platform.

The naive implementation on the CPU has followed the sequence illustrated by Algorithm 1. However, to efficiently

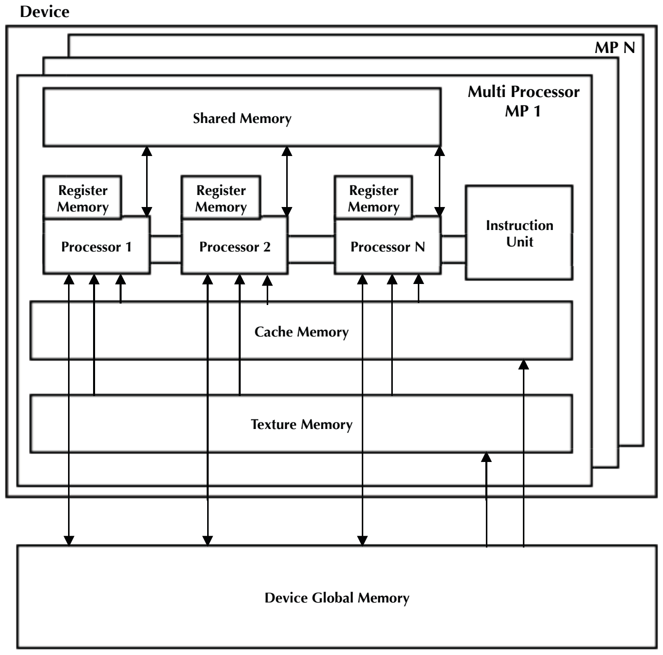

Figure 2. CUDA Architecture Hardware Model. The device is composed of parallel Multi-Processors where each one contains parallel running processor units connected to different kinds of memories.

Algorithm 1 Prefix Sum Algorithm

```
1: $sum[0][1..n] \leftarrow 0,$
2: $sum[1..m][0] \leftarrow 0$
3: for $i = 1 \rightarrow m$ do
4:    for $j = 1 \rightarrow n$ do
5:       $sum[i][j] = sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1] + a[i][j]$
6:    end for
7: end for
8: $min \leftarrow 0$
9: $M \leftarrow 0,\ s[0] \leftarrow 0$
10: for $g = 1 \rightarrow m$ do
11:    for $i = g \rightarrow m$ do
12:       $min \leftarrow 0$
13:       for $j = 1 \rightarrow n$ do
14:          $s[j] \leftarrow sum[i][j] - sum[g-1][j]$ $//s[j] = sum_{g,i}[j]$
15:          $cand \leftarrow s[j] - min$
16:          $M \leftarrow$ MAX $M, cand$
17:          $min \leftarrow$ MIN $min, s[j]$
18:       end for
19:    end for
20: end for
21: $output\ M$
```

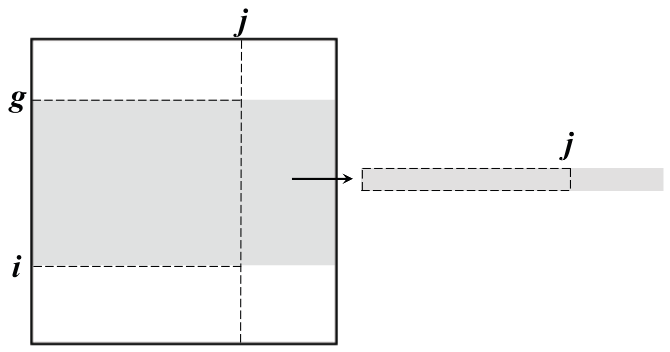

Figure 3. The 2D strip bounded by *g*, *i* is transferred to it's corresponding 1D prefix sum array.

exploit the high parallelism of the GPU, this algorithm has been modified – as illustrated by Fig. 4 – as follows:

1) In the first stage, we start constructing the prefix sum matrix by horizontal summation of the elements such that each element of the prefix sum matrix holds the summation of all elements above it from the original input matrix, then following that by vertical summation to the resulted matrix.
2) In the second stage, and to get a unique mapping of indices, we construct m^2x1 array to hold the maximum summation of each strip instead of previously $m(m+1)/2$ (where some slots of this array will be empty). We do that through letting i start from the first row of the matrix, not as previously from g, and then before starting the for loop in step 13 (see Algorithm.1), we check on i not being less than g.
3) In the final stage, we use the same reduction technique presented in [13] to get the maximum of the m^2x1 array of the previous step.

Assuming NXN matrix, the first stage complexity will be $O(N)$ using N threads. The second stage will be also of $O(N)$ complexity but using N^2 threads, and the final stage complexity will be $O(logN)$. The overall time complexity of the parallel algorithm will be $O(N)$.

VI. Results & Discussion

The implementation has been benchmarked on a workstation shipped with an Intel Core i7 CPU running at 2.0 GHz, 4 GB of memory, and NVIDIA GeForce GT 525M GPU having 1 GB of DDR3 memory and 96 CUDA cores each operating at 1.2 GHz. On the software side, CUDA 4.2 toolkit was installed on Ubuntu 12.04. Benchmarking has been done relying on a library called cuYURI [4]; which transparently performs function and kernel profiling using high precision Boost and CUDA timers for CPU and GPU respectively.

As depicted in Fig. 5, testing the implementation on a list of arrays of sizes ranging from 128^2 to 1024^2, the GPU implementation outperforms the CPU one by a speedup of $5.54X$ for the 128^2 array and increases gradually to reach

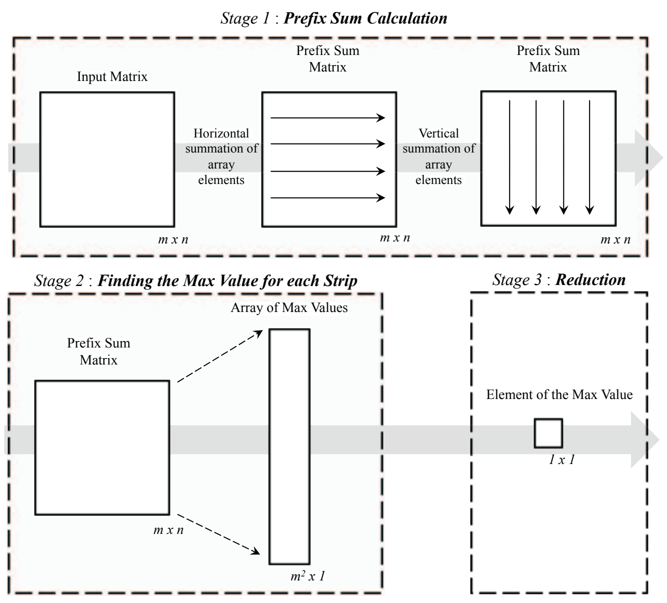

Figure 4. The implementation of the MSP on the GPU.

the maximum speedup of $7.35X$ for an array of size 512^2. Increasing the array size to 1024^2, a slight drop down in the gained speedup was observed as a result of the limited compute capabilities of the employed GPU. Additionally, extending the benchmarking process for larger arrays was not feasible due to the memory constraints of the GPU. In fact, the achieved speedups are considered suitable to the nature of the addressed problem which has inherently high unbalanced workload between the working threads during the second stage of the algorithm.

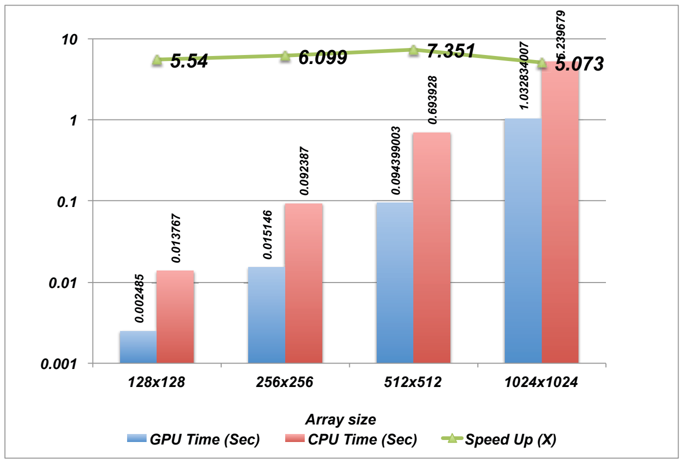

Figure 5. Benchmarking CPU verses GPU performance for the MSP on different array sizes.

VII. Conclusions

In this work, a stage by stage high performance linear time implementation for the MSP on CUDA-enabled GPUs has been presented. The performance on the GPU was increasing gradually until succeeded in achieving more than $7X$ speedup in arrays of size 512^2 and then a slight drop down was observed for higher array dimension of 1024^2. The results are considered very convenient considering the unbalancing of the parallel threads due to the nature of the problem.

VIII. Future Work

This work can be extended to support working on arrays with larger sizes and re-benchmarking the results on dedicated GPU clusters. Further optimization for the serial algorithm as well as the CUDA implementation should be investigated. A comparison between the GPU implementation and a multi-threaded CPU implementation would be also considered in the future.

References

[1] K. Perumalla and N. Deo, "Parallel Algorithms For Maximum Subsequence And Maximum Subarray", 1995.

[2] Sung Eun Bae, "Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem", University of Canterbury, Chapter1, 2007.

[3] mini-MIAS. http://www.andrew.cmu.edu/course/15-440-s12/applications/labs/lab4/mias.html. [Database of Mammograms].

[4] cuYURI. https://github.com/marwan-abdellah/cuYURI. [CUDA Image processing library].

[5] U. Grenander, *Pattern Analysis.* Springer Verlag, 1978.

[6] J. Bentley, "Programming pearls: algorithm design techniques". *Communications of the ACM 27,* pp. 865-873, 1984.

[7] J. Bentley, "Programming pearls: algorithm design techniques". *Communications of the ACM 27,* pp. 1087-1092, , 1984.

[8] Z. Wen, "Fast Parallel Algoritihms for the Maximum Sum Problem". *Parallel Computing 21,* pp. 461-466, 1995.

[9] K. Perumalla and N. Deo, "Parallel Algorithms for Maximum Subsequence and Maximum Subarray". *Parallel Processing Letters 5,* pp. 367-373, 1995.

[10] K. Qiu and S. Akl, "Parallel Maximum Sum Algorithms on Interconnection Networks". Tech. Rep. pp. 99-431, Queen‘s University, Department of Computer and Information Science, 1999.

[11] NVIDIA. "NVIDIA CUDA C Programming Guide", April 2012.

[12] Sung Eun Bae, "Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem", University of Canterbury , Chapter2, 2007.

[13] D. Kirk and W. Hwu, "Programming Massively Parallel Processors", chapter 6. Elsevier Inc., 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2010.