A Performance Modeling Showcase

Giacomo Marciani
University of Rome Tor Vergata
Via del Politecnico 1
Rome, Italy 00133
gmarciani@acm.org

ABSTRACT

Modern software applications process massive amounts of data under stringent timing constraints. The advances in HPC make technologies such as OpenMP and CUDA really attractive to boost performance of complex algorithms. In this context, deep learning and prefix-sum operations provide a great benchmark to exploit HPC.

In this technical report we propose massively parallel implementations of two serial-born pieces of software, written in C. First, we propose an OpenMP and CUDA implementation of a deep learning algorithm for hand-written digits recognition leveraging the back-propagation. Then, we propose a CUDA implementation of the prefix-sum operation over a vector of integers.

The experimental results show that about neural networks show a speed-up of ∼1300% for the OpenMP and ∼1380% for CUDA implementation, with respect to the serial counterpart. On the other hand, the experimental results about prefix-sum show a good scaling with respect to the increasing input size; they show, nonetheless, an efficient use of the GPU resources. Although the promising results, we conclude our work delineating possible improvements for our model.

CCS CONCEPTS

• **Networks** → **Network simulations**; **Network performance analysis**; • **Theory of computation** → **Random walks and Markov chains**;

KEYWORDS

performance modeling; simulation tools

ACM Reference format:
Giacomo Marciani. 2018. A Performance Modeling Showcase. In *Proceedings of A Performance Modeling Showcase, Rome, Italy, January 2018 (PMCSN'18),* 5 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Modern software applications for scientific calculus, artificial intelligence and big data analytics require massively parallel algorithms to rapidly extract value from huge and complex datasets.

In this context, HPC technologies like OpenMP and CUDA are a de-facto standard for the implementation of such algorithms. In particular, OpenMP makes parallel programming on CPU a lot easier, thanks to compiler directives; whilst CUDA allows to fully exploit the computational power of GPU.

In this work, we present a massively parallel OpenMP/CUDA implementation of (i) a deep learning algorithm for hand-written digits recognition leveraging back-propagation and (ii) a prefix-sum algorithm on integer vectors.

Mobile devices are still limited in computing capabilities and battery lifetime. The problem of enhancing user experience may find a solution in cloud computing. Consider a "two-layer" cloud system, consisting of an edge cloud server (cloudlet) and a remote cloud server, where the cloudlet is at "one-hop" distance from a set of mobile devices. Applications running on these mobile devices autonomously select some of their tasks for offloading to an external server (e.g., because of performance or energy saving reasons), and send an offloading request to a controller located on the cloudlet. Upon receiving a request, the controller takes a decision about whether the task should be sent to the cloudlet or the cloud, with the goal of minimizing the mean response time. Typically, tasks hosted by the cloud server benefit of a higher execution rate, but suffer for greater network delay. We assume that the remote cloud server has virtually unlimited resources, hence it is always able to guarantee absence of interference among any number of tasks allocated to it. On the other hand, the cloudlet has limited resources, so that it is able to guarantee absence of interferences among tasks allocated to it as long as their number does not exceed a given threshold N. Consider the following modelling assumptions: • The users belong to two classes, class 1 and class 2, with the following arrival rates: lambda 1 = 3.25 task/s, lambda 2 = 6.25 task/s • The service both on cloudlet and on the cloud can be assumed exponential, with the following rates: mu 1clet = 0.45 task/s, mu 1cloud = 0.25 task/s mu 2clet = 0.30 task/s, mu 2cloud = 0.22 task/s Note that, as we stated above, the service time includes the transmission time. We assume mu iclet > mu icloud for both classes. Moreover, note that the execution on the cloudlet of a class 1 task is more convenient than the execution of a class 2 task. Let us denote with (n 1 , n 2) the state of the system, with n i the number of class i tasks in execution on the cloudlet, and with S <= N a given threshold. Under the above assumptions, the controller can take the following decisions upon task arrival events: class 1arrival: if n1=N ! send on the cloud else if n1+n2<S ! accept else if n2 > 0 ! accept the task on the cloudlet and send a class 2 task on the cloud else accept the task on the cloudlet class 2 arrival: if n1+n2>=S ! send on the cloud else accept the task on the cloudlet When a class 2 task is interrupted and sent on the cloud, a setup time s setup has to be considered to restart the task on the cloud. We assume an exponential time with

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PMCSN'18, January 2018, Rome, Italy
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

mean E[s setup]= 0.8 s. Define a queueing model for the system above. 2. Simulate the system model with N=20 and evaluate the system response time and the effective 1 throughput as a function of the threshold S. a. Determine if the system is stationary or not and design the experiments accordingly. b. For a S=N estimate the distribution of the response time. Motivate the chosen methodology and conjecture about the shape in respect to known distributions. 3. Determine a threshold S to minimize the response time. Hints: you can start with S=N and try 2 different values for S The steady state or transient statistics should be computed with a 95

The remainder of the paper is organized as follows.

In Section 2 we give some background notions to better understand and contextualize the adopted approach for the deep learning and prefix-sum algorithms. In Section 3 we describe the implementation of the neural network and the back-proagation algorithm, leveraging OpenMP. In Section 4 we describe the implementation of the neural network and the back-propagation algorithm, leveraging CUDA. In Section 5 we describe the implementation of the prefix-sum operation, leveraging CUDA. In Section 6 we conclude the paper summing up the work that has been done.

2 RANDOM NUMBER GENERATION

There exist many other techniques for pseudo-random number generation. The most notable are *multiple recursive generators*, *composite generators*, and *shift-register generators*. All of these produce periods wider than the one produced by a Lehmer generator, with the cost of computational complexity. However, we adopt Lehmer generator because it is effective, efficient and widely adopted, thus being perfect for our showcase.

Il progetto presentato si propone di testare il noto generatore pseudo-casuale di numeri random di Lehmer utilizzando, a tale scopo, uno dei sei test di casualità illustrati nella sezione 10.11. Per una migliore comprensione sull'importanza dell"uso di tale test, si è preferito implementare una propria versione del generatore lehmer; si è, quindi,scelto un moltiplicatore differente da quello proposta dal libro di testo che soddisfacesse dei requisiti particolari (spiegati nelle sezioni successive); una volta testato il generatore con tale moltiplicatore scelto effettuando i test preposti si sono confrontati i risultati riscontrati con quelli ottenuti dall'uso dei moltiplicatori e di altri parametri utilizzati dal libro di testo. Infatti da tale confronto si è compreso se le scelte effettuate a priori riguardo a una implementazione "nativa" possono considerarsi valide, e questo è possibile analizzando i grafici ottenuti in seguito alle simulazioni che forniscono indicazioni sul grado di casualità del generatore testato con i parametri di input scelti. Per tali test sono stati utilizzati le librerie offerte dal libro di testo, i quali forniscono una lista di funzioni API (scritti in linguaggio C) pronte per essere utilizzati in qualsiasi programma simulativo. Per sfruttare al maglio tali API, si è preferito implementare il progetto con lo stesso linguaggio di programmazione, mentre per la costruzione e visualizzazione dei grafici in seguito ai test simulativi è stato scelto di usare GNUPLOT per il linguaggio C.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Il primo passo necessario per una implementazione nativa del generatore Lehmer è stata la scelta di un moltiplicatore che godesse le proprietà di full-period e module-compatible rispetto al modulo m, dove m è il più grande numero intero primo rappresentabile in un calcolatore elettronico a 32 bit (m= 231 - 1). Si è dovuto, quindi, creare la lista di tutti i moltiplicatori che avessero simili proprietà e sceglierne uno tra questi per i nostri scopi. Il numero totale di moltiplicatori full-period e module-compatible calcolati in una architettura a 32 bit sono non più di 23093. Tra questi ci sono anche i moltiplicatori utilizzati dal libro di testo, ossia il moltiplicatore 16807, usato largamente in passato1 fino alla scoperta del moltiplicatore ad oggi più utilizzato e considerato migliore in termini di generazione di sequenze di numeri random, il cui valore intero è 48271. Di seguito verrà illustrato l'algoritmo utlizzato per il calcolo dei moltiplicatori full-period e module compatible rispetto ad m:

Dato il modulo primo m ed un qualsiasi moltiplicatore full-period e module -compatible, l'algoritmo riportato sopra genera tutti i moltiplicatori full period e module compatible a m. Per ogni iterazione dentro il while, viene controllato se il moltiplicatore x calcolato nella iterazione precedente soddisfa tali proprietà, in tal caso viene inserito in un'array, altrimenti si procede nel calcolo successivo di un nuovo intero. Tale iterazione è esprimibile come: ai mod m. Per evitare condizioni di overflow dovuti al prodotto a*Xi vengono calcolate resto e il quoziente della funzione, e in base al segno dell'operazione di sottrazione tra questi è possibile determinare il valore intero del moltiplicatore senza produrre alcun valore intermedio o finale più grande di m-1.

Dopo aver determinati tutti e 23093 moltiplicatori, questi vengono salvati in un file di testo per un facile recupero in un secondo momento. Per i nostri test abbiamo scelto infine il moltiplicatore di valore 50812 che gode delle proprietà di FULL-PERIOD e MODULE-COMPATIBLE rispetto al modulo m.

Finora i test spiegati in precedenza sia per determinare i moltiplicatori full-period e module-compatible con m che per i test spettrali si è generato un unico stream di valori pseudo-casuali. Tuttavia nei programmi di simulazione discrete-event in cui si vuole simulare un sistema composto da varie componenti stocastiche, è conveniente utilizzare un generatore multi-stream e, a tale scopo, viene fornita la libreria rngs.c. Tale libreria implementa la funzione di lehmer estesa al multi-stream, introducendo il moltiplicatore di salto: aj mod m che viene computata solo una volta; grazie a questo, infatti la funzione di salto gj (x) = (aj mod m)*x mod m fornisce un modo per separare diverse sequenze di numeri random, e se il parametro di salto j viene scelto in modo appropriato, viene garantita anche la disgiunzione degli stream creati, ognuno con un proprio seme iniziale determinato dal primo seme iniziale impostato. in questo

progetto si è scelto di mantenere 256 streams di numeri random previsti dalla libreria rngs.c. Tuttavia, si deve considerare il nuovo moltiplicatore scelto, a = 50812, per cui è necessario determinare nuovamente la variabile di salto j tale che il moltiplicatore di salto aj mod m sia modulo compatibile con m1. La variabile di salto j, che determina la partizione della sequenza in 256 streams disgiunti di uguale lunghezza, dovrà quindi essere il più grande intero minore di 231 / 28 =8388608 per garantire tale proprietà. L'implementazione per determinare il nuovo moltiplicatore di salto è riportato di seguito:

La funzione prende in input il moltiplicatore di base a e imposta le variabili locali a zero. Per ogni iterazione viene computato il valore del moltiplicatore di salto utilizzando il confronto con il resto e il quoziente rispetto ad m in modo simile alla computazione dei moltiplicatori full-period e module-compatible visto nella sezione precedente (per evitare l'overflow del prodotto). Viene quindi controllato se il moltiplicatore di salto è module compatible con m tramite la chiamata alla funzione check():

una volta determinato il moltiplicatore di salto, è stato quindi reso necessario modificare la libreria rngs.c in modo da essere adatta ai scopi del progetto. Con i nuovi valori del moltiplicatore a = 50812 e il moltiplicatore di salto aj mod m = 29872 integrati nella libreria è possibile utilizzare il generatore multi-stream di Lehmer per i randomness test riportati nel capitolo 10 del libro di testo.

La prima parte di questo progetto dimostra come la scelta di implementare un proprio generatore random comporta tutta una serie di considerazioni (e riflessioni) necessarie se si vuole avere un grado di bontà accettabile riguardo alla generazione di sequenze pseudocasuali. Infatti pur scegliendo un proprio moltiplicatore che soddisfa le proprietà discusse nei capitoli precedenti e il proprio generatore riesca a passare i test statistici introdotti, tale generatore random potrebbe risultare persino peggiore tra quelli conosciuti (basti vedere il confronto fatto dal libro con i moltiplicatori a=16807 con a=48271). Per questo motivo esiste la differenza tra generatori ideali e quelli reali

3 MODELING

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Figure 1: The conceptual model.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4 EVALUATION

The experiments have been conducted on an Amazon EC2 c3.8xlarge instance, which is really indicated for high performance science

Figure 2: The Spectral Test to evaluate randomness of the pseudo-random numbers generator.

and engineering applications[1]. The instance is equipped with 32 vCPU based on an Intel Xeon E5-2680 v2 (Ivy Bridge) processor, 30 GB of RAM and SSD with 900 IOPS. It runs Debian 8.3 (Jessie), Python 3.5.2, and the Python-ported version of the official Leemis library for discrete-event simulation, indicated in [2].

Our solution has been developed in Python, following the de-facto standard best-practices, stated in [1, 3]. For the production of plots Matplotlib 1.5.3 has been used.

The simulations are replicated several times to clear out the weight of unrepresentative subsequences. Statisticians discard outliers as being unrepresentative, but it is not appropriate to do it in simulation.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

L'ultima parte di questo progetto riguarda l'applicazione dei test di casualità per il generatore di numeri random. Il generatore Lehmer usato è preimpostato con (a,m) = (50812, 231 -1) nella libreria rngs.c spiegato nella sezione precedente, mentre il test utilizzato è stato il test degli estremi (extremes test). Per la simulazione di tale test si può riassumere il processo in tre passi: Generazione di un campione di valori con chiamate ripetute al generatore. Computazione di un test statistico la cui distribuzione (pdf o funzione di densità di probabilità) è nota su variabili random uniformi in (0,1) indipendenti e identicamente distribuiti (iid). Valutare la verosimiglianza del valore computato del test statistico con la relativa distribuzione teorica da cui è stato assunto adottando una metrica basato sulla distanza lineare.

Il test degli estremi si basa sulla considerazione che se si ha una sequenza iid di variabili random uniformi in (0,1) e se una variabile R = maxU0, U1,..., Ud-1 allora la variabile random U = Rd è anch'essa uniforme in (0,1)1.

l'algoritmo del test degli estremi effettua un raggruppamento (batching) dei valori estratti dal generatore in gruppi di uguale lunghezza (determinato dal parametro d), trovando il massimo di ogni batch, elevandolo tale massimo all d-esima potenza e conteggiando tutti i massimi generati in un array come illustrato di seguito:

[1]https://aws.amazon.com/ec2/instance-types/

Figure 3: The Test of Extremes to evaluate randomness of the pseudo-random numbers generator.

Figure 4: The Kolmogorov-Smirnov Analysis to evaluate randomness of the pseudo-random numbers generator.

I valori critici v*1 e v*2 vengono calcolate utiizzando la funzione inversa idfChisquare(long n, double u) fornita dalla libreria rvms.c del libro di testo. Bisogna precisare che per il calcolo di tali variabili statistiche è stato scelto un livello di confidenza con parametro alpha = 0.05, mentre i parametri N e K sono rispettivamente N = 10000 e K = N/10 = 1000. Successivamente si confronta la statistica chi quadro v, determinata al passo precedente, con i valori critici v*1 e v*2 , per ogni stream(in totale sono 256 variabili chi-quadro v). Se v < v*1 o v > v*2 il test fallisce (per quello stream) con probabilità 1 - alpha. Il grafico risultante di questo test empirico è illustrato di seguito:

I valori critici sono visualizzati come linee rosse orizzontali: quella inferiore rappresenta v*1 = 913.3 mentre quella superiore è v*2 = 1088.5

Dalla simulazione effettuata si è notato che il numero di test statistici v > v*2 sono stati 6 mentre il numero di test v < v*1 sono stati 10. Di seguito è riportato l'output del programma:

Considerando il numero totale di test falliti (upper e lower bound)1 pari a 16, si nota che non ci si discosta molto rispetto al valore atteso approssimato; infatti in 256 test con un livello di confidenza del 95% il valore aspettato è circa 256 * 0.05 = 13 fallimenti. Questo valore può essere una indicazione della bontà del generatore di Lehmer implementato. dai risultati visti in precedenza si è notato che il numero totale di test falliti è pari a 16 non lontano dal valore di riferimento pari a 13.

Il test statistico di Kolmogorov-Smirnov misura la più grande distanza verticale tra la funzione di distribuzione cumulativa calcolato da un dataset e una funzione di distribuzione cumulativa teorica1. tale statistica è indicata con d. Il primo passo per effettuare questo tipo di test è di ordinare la sequenza dalle variabili del dataset in senso crescente applicando un algoritmo di ordinamento Se n=256 e alpha=0.05 si ha quindi d*=0.084. Perciò la probabilità che una singola variabile KS d256 sia minore di 0.084 è di 0.95. Il test KS può considerarsi fallito se la statistica computata d256 supera il valore critico d*=0.084. La simulazione di questo test è stato effettuato con il generatore Lehmer settato con i parametri (a,m) = (50812, 231 -1) n=256 ed alpha=0.05. Dal risultato della simulazione il valore d256 computato ha il valore pari a d256 = 0.043433 minore del valore critico d*=0.084. pertanto si può considerare il test KS superato con successo. Il grafico ottenuto viene riportato di seguito:

la linea verticale tratteggiata indica il valore della statistica chi-quadro a cui è stato determinato la distanza massima.

Figure 5: Response Time Analysis.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5 USAGE

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6 CONCLUSIONS

In this work we propose implementations of parallel algorithms for deep learning and prefix-sum, leveraging OpenMP and CUDA.

The OpenMP-based deep-learning solution exploits the amends made to the serial code to avoid inefficiencies from the point of view of memory. It then leverages parallel for loops, task-based parallelism and parallel reductions to achieve a reasonable speedup. We show a satisfying speedup scaling using increasing number of threads on a desktop computer and a powerful cluster node.

The CUDA-based deep learning solution mainly leverages memory linearization, data transfer overlapping and streams. We demonstrate that the response-time of the CUDA-based solution converges to a speed-up of $\sim 1380\%$ from block sizes bigger than 128 threads and it is not much sensible to compiler optimizations.

With respect to prefix-sum, we implemented a $O(nlogn)$ solution leveraging shuffle operations within warps. We demonstrate that, when considering a maximum array size of 65536 elements, the response time of such a solution is pretty insensible to block size variations.

Although experimental results are pretty satisfactory, the proposed solutions could certainly be improved and be subjected to a more in-depth analysis. With respect to the deep learning CUDA algorithm, the most important improvements that should be addressed concern the (i) avoidance/reduction of atomic operations by the use of shuffle operations and shared memory and (ii) the use of streams synchronization leveraging events. With respect to the prefix-sum CUDA algorithm, the most important aspects to address are the implementation of solution with less computational complexity, e.g. solutions with $O(n)$ exist, and the study of response time when considering bigger input sizes.

REFERENCES

[1] Google. 2016. The Google's Python Styleguide. (sep 2016). http://bit.ly/2d4M9UN
[2] Lawrence M Leemis and Stephen Keith Park. 2006. *Discrete-event simulation: A first course.* Pearson Prentice Hall Upper Saddle River.
[3] Kenneth Reitz and Tanya Schlusser. 2016. *The Hitchhiker's Guide to Python: Best Practices for Development.* O'Reilly Media. http://amzn.to/2bYCvBD