ultra fast SOM using CUDA

SOM (Self-Organizing Map) is one of the most popular artificial neural network algorithms in the unsupervised learning category.
### contents

<table>
<thead>
<tr>
<th>0.1</th>
<th>Abstract</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>Introduction</td>
<td>01</td>
</tr>
<tr>
<td>0.3</td>
<td>CUDA Programming Paradigm</td>
<td>01</td>
</tr>
<tr>
<td>0.4</td>
<td>Optimization Strategies</td>
<td>02</td>
</tr>
<tr>
<td>0.5</td>
<td>Results and Discussion</td>
<td>03</td>
</tr>
<tr>
<td>0.6</td>
<td>Conclusion and Future Work</td>
<td>04</td>
</tr>
<tr>
<td>0.7</td>
<td>Contact</td>
<td>04</td>
</tr>
<tr>
<td>0.8</td>
<td>Reference</td>
<td>04</td>
</tr>
</tbody>
</table>
Ultra Fast SOM using CUDA

Abstract

SOM (Self-Organizing Map) is one of the most popular artificial neural network algorithms in the unsupervised learning category. For efficient construction of large maps searching the best-matching unit is usually the computationally heaviest operation in the SOM. The parallel nature of the algorithm and the huge computations involved makes it a good target for GPU based parallel implementation. This paper presents an overall idea of the optimization strategies used for the parallel implementation of Basic-SOM on GPU using CUDA programming paradigm.

Keywords - Basic-SOM, Data Mining, CUDA.

I. INTRODUCTION

Self Organizing Map or SOM is a data visualization technique which reduces the dimensions of data through the use of self-organizing neural networks. The way SOMs go about reducing dimensions is by producing a map of usually 1 or 2 dimensions (in our case it is 2 dimensions) which plot the similarities of the data by grouping similar data items together. So SOMs accomplish two things, they reduce dimensions and display similarities.

The first part of a SOM is the data. The self-organizing maps will project this n-dimensional data into something that can be better understood visually. In this step the input vector will search for the map node that makes the minimum Euclidian distance and is selected as the winner node. And the circular area around winner node is defined as proximity area. All the nodes in the proximity area do the study of input vector, and move a little to the input direction in accordance to formula:

\[ m_i(t+1) = m_i(t) + h_o(t)[ x(t) - m_i(t) ] \]

At the beginning of the learning process the radius of the proximity area is fairly large, but it is made to shrink during learning. This ensures that the global order is obtained already at the beginning, whereas towards the end, as the radius gets smaller, the local corrections of the model vectors in the map will be more specific. There is a function to decease the proximity area and this can be explained by the following equation:

\[ h(d, t) = \exp\left(-\frac{d^2}{k(r(t))^2}\right), \text{ and} \]

\[ h_o(t) = \alpha(t). h(d, t) \]

where,

\[ d = (x - a_i)^2 + (y - b_i)^2, \text{ and} \]

\[ 0 < \alpha(t) < 1 \text{ with, } \alpha(t) = \alpha_0(1 - t/T) \]

Here, \( \alpha_0 \) is in the initial value of \( \alpha \), and the value is normally 0.2 to 0.5. \( T \) is a given study frequency.

Briefing the SOM steps:

1. Initialize Map
2. for \( t \) from 0 to (number of iterations)
   1. Randomly select a sample
   2. Get best matching unit
   3. Scale neighbors
   4. Increase \( t \)
3. End for

2 CUDA Programming Paradigm

NVIDIA’s CUDA architecture is a parallel computing architecture that delivers the graphics processor technology to general purpose GPU computing. It is organized as a set of multiprocessors, each having 8 CUDA cores and an on-chip memory shared all the 8 CUDA cores within the multiprocessor. Besides this a global memory space is also available which is shared by all the multiprocessors. The overall architecture is shown in Figure1. This architecture allows the GPU to be viewed as a data-parallel
computing device that operates as a coprocessor to the main CPU (the host).

At the hardware level, the GPU is a collection of multiprocessors, with several processing elements in each. Each processor in the multiprocessor executes the same instruction in every cycle. Each can operate on its own data, which makes the multiprocessor a SIMD processor. Communication between multiprocessors and between the GPU and the CPU is possible only through the global memory space, which can be accessed and modified by all the cores of the multiprocessors and also from the CPU thread. The processing elements of a multiprocessor can synchronize with each other, but there is no direct synchronization mechanism between the multiprocessors.

For the programmer, CUDA provides a C extension, which makes it easy to explore the parallel processing power of GPUs for compute-intensive general purpose applications. This hides the graphical behavior of GPUs from developers and gives them the comfort of using it as a many-core platform which is computationally much faster.

The CUDA APIs hides all the complexities involved in thread creation and will generate light-weight threads for execution on CUDA cores, all at the ease of a function call. However, the total number of CUDA threads and its occupancy in multiprocessors is fully dependent on the parameters specified at the API invocation. There lies the need for significant programmer effort to utilize the GPUs at its peak and obtain the best performance. Efficient memory access patterns and the efficient utilization of available GPU memories like texture, shared and constant memories is also required. This may require the serial logic to be polished or rewritten to best suite the GPU's parallel architecture.

3 Optimization Strategies

Most real world problems using the SOM logic require efficient construction of large maps. This involves processing of large data buffers and huge computations are involved in the matching process. Once the map got initialized with some random values, the rest of the computations which is compute intensive are mostly data parallel and this makes it a good target of GPU based parallel implementation.

Our implementation splits the entire SOM implementation logic into 3 major CUDA kernels. (a) Distance finding, (b) Reduction operation for minimum finding and (c) Adjust weight kernel.

The CUDA kernels are optimized to achieve their best performance by the efficient utilization of the GPU memory, i.e. by using constant, shared and texture memories and by avoiding non-coalesced global memory accesses.

Some constant parameters which remains the same for a given input data and used within CUDA kernel for various calculations, is placed inside the GPU’s constant memory (which is the fastest of all the available GPU memories; but is very much limited in availability).

The data buffer given as input to the SOM logic is used for various calculations inside the above CUDA kernels. So this input data buffer is copied to the device’s global memory prior to the kernel invocations. To avoid too many accesses to global memory (which is much slower), we tried with texture binding the input buffer (thereby
making it cached), and the cached texture is used for all the computations within the kernels.

All the other buffers allocated in the GPU’s global memory and used within the CUDA kernels are either cached using texture binding or copied to shared memory (which is the on-chip memory) to achieve the best performance.

The kernel invocations are also checked for better utilizations of multiprocessor cores by making the block size (i.e. the number threads within a block and this will be considered as a single unit to be active in a multiprocessor) optimal based on register usage, shared memory usage and other parameters.

4 Results and Discussion

Table 1 gives the performance comparisons in figures for various NVIDIA GPUs by changing the map size and attribute count. The performance is measured in terms of speedup with respect to the CPU (Intel Core(TM) 2 Quad CPU Q8400 @ 2.66GHz (2666 MHz), using only single core) implementation and is taken for GTX260 and Tesla C1060. Figure 2 and Figure 3 plots the time measured (in seconds) on these platforms.

According to the current GPU implementation, increasing the map size will increase the parallel paths in GPU and increasing the attribute count will increase the computation within each CUDA thread (since GPUs are good at computation this will not add much to the CUDA kernel execution time). Analyzing the results from table 1, it is obvious that as the map size or attribute count goes higher, the CUDA implementation will be much faster compared to the corresponding CPU implementation.

** Table 1: Performance Comparison Table (in terms of speed-up)**

<table>
<thead>
<tr>
<th>Map Size**</th>
<th>Attributes</th>
<th>CPU</th>
<th>Tesla C1060</th>
<th>GTX 260</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 x 2000</td>
<td>16</td>
<td>1x</td>
<td>57x</td>
<td>55x</td>
</tr>
<tr>
<td>1500 x 1500</td>
<td>32</td>
<td>1x</td>
<td>72x</td>
<td>66x</td>
</tr>
<tr>
<td>2000 x 2000</td>
<td>32</td>
<td>1x</td>
<td>74x</td>
<td>69x</td>
</tr>
<tr>
<td>2500 x 2500</td>
<td>32</td>
<td>1x</td>
<td>76x</td>
<td>NA*</td>
</tr>
<tr>
<td>3000 x 3000</td>
<td>32</td>
<td>1x</td>
<td>70x</td>
<td>NA*</td>
</tr>
<tr>
<td>1000 x 1000</td>
<td>64</td>
<td>1x</td>
<td>76x</td>
<td>73x</td>
</tr>
<tr>
<td>1500 x 1500</td>
<td>64</td>
<td>1x</td>
<td>84x</td>
<td>77x</td>
</tr>
<tr>
<td>2000 x 2000</td>
<td>64</td>
<td>1x</td>
<td>84x</td>
<td>NA*</td>
</tr>
</tbody>
</table>

* GTX260 is not having enough GPU memory to handle this data.

** With iterations and data items fixed to 20 and 16 resp.
5 Conclusions and Future Work

We have shown that the SOM implementation can benefit greatly from using CUDA-capable GPU. Overall the best CUDA implementation provides a speed of 84x (compared to CPU implementation) for 2000x2000 map size and with 64 attribute on a Tesla C1060 GPU.

When the map size or the attribute count is too large, there is not enough space in the GPU to hold the entire data at the same time. Of the above discussed test platforms, GTX260 is having only 896MB memory size, while Tesla C1060 is having 4GB memory size. The maximum limit for map size and attribute count is much lower for GTX260 compared to the Tesla, which makes some map sizes not supported in GTX260 as given in the results table. To avoid this we have to split the data according to the device memory availability and do the computations separately for various blocks and for this we have to rewrite the logic accordingly, which is planned to be our next milestone.

The straightforward CUDA implementations can achieve substantial benefits. For further performance tuning, significant programmer effort can be required to fully utilize GPU's potential when irregular memory access patterns or small kernels are present. Despite this extra effort required to realize the potential of the GPU, the benefits can be dramatic. Our experience with CUDA shows the power of the GPU as a parallel platform, and is a proof to the fact that, many-core platforms are having the potential to improve the performance of various compute intensive applications.

6 Contact

QuEST-NVIDIA Center for GPU computing.

7 References


