Fall 2017 ME759 Final Project Report

University of Wisconsin-Madison

GPU Implementation of Order-Independent Pixel-Based Texture Synthesis

University of Wisconsin Madison

Jieru Hu

December 2017

**Abstract**

Recent years have witnessed significant progress in texture synthesis algorithms, an approach to reproduce large-sized texture based on a tiny input. However, the traditional texture synthesis approach implemented on the CPU architecture suffers from lack of efficiency and improvement potential. Therefore, more efforts are needed to develop an algorithm that is efficient and capable of producing high quality results.

This study aims to tackle this challenging task by designing and implementing a high-performance pixel-based algorithm for synthesis of images composed by textures. The idea of the order-independent algorithm improves the quality of the algorithm’s generated output by operating the synthesis through multiresolution pyramids. The multiresolution strategy allows a bottom-up, hierarchical representation of images, through which we are able to filter and extract features of convoluted images and apply them to the consecutive output. On each pyramid level, we choose pixels for the synthesis from the input pixels in the sample that share the highest neighborhood resemblance.

To further improve the efficiency of the algorithm, a GPU-based (Graphical Processing Unit) implementation is proposed and implemented on the CUDA platform. The mechanism behind to schedule the intensive computation work across the threads on the GPU streaming processors and execute them simultaneously. Techniques for optimizing the parallel computation were considered and implemented to guarantee the best performance of the algorithm on the CUDA architecture.

As a result, GPU-based implementation shows a remarkable improvement on the synthesis speed comparing with the traditional CPU-based approach. After benchmarking, a 30-times speedup on the synthesis process has been observed. Furthermore, a list of scaling analysis on GPU implementation is conducted based on different user-defined parameters in the synthesis process.

**Contents**

1. Introduction………………………………………………………………….4
2. Orderly-Independent Texture Synthesis…………………………………….4
   1. Related Work……………………………………………………..4
   2. Sequential CPU Implementation…………………………………4

2.3 MATLAB Multi-Core Acceleration…………..………………….6

1. GPU Implementation………………………………………………………..7

3.1 CUDA Background……………………………………………...7

3.2 Texture Synthesis Acceleration with CUDA…………………....7

3.3 GPU Parallel Implementation…………………………………...8

1. Result and Scaling Analysis………………………………………………...8
2. Conclusion………………………………………………………………….10
3. References………………………………………………………………….11
4. **Introduction**

Across various fields of scientific study, such as computational photography, material science, and so on, texture synthesis as an approach to reproduce a large output from a tiny sample input has regularly received attention. Traditionally, texture synthesis algorithm is implemented on the CPU and considered to be inefficient, because intensive computational work is only limited to the sequential execution on CPU. This study focuses on developing a GPU implementation based on an Orderly-Independent Pixel-based texture synthesis algorithm - a technique of great efficiency and capability, for the purpose of constructing large output images based on an arbitrary user-defined input. The GPU approach of the problem provides a significant speed boost on the synthesis speed and quality, compared with the CPU sequential computation. In reality, this accelerated technique has a wide range of applications. For example, it can be used to repair a broken image or to reconstruct damaged materials using textural fragments from the unimpaired parts. Moreover, we can apply this technique on different dimensions by mapping textures onto 2D- or 3D- object surfaces, which can be extremely useful in geometric modeling.

1. **Orderly-Independent Texture Synthesis**
   1. **Related Work**

In recent decades, there have been a great many studies on texture synthesis from a diverse range of subjects including statistics, computer modeling, psychology, image processing, and material science. Typically, the key of an algorithm is to enforce each local element to preserve the global patterns from the sample input and formulate an efficient way of sampling each unit.

A prevailing approach among all search-based synthesis algorithms is using a pixel-based unit for synthesizing. As the name implies, one single pixel is selected to synthesis at a time. Wei and Levoy [1] developed an algorithm on pixel synthesis operated on a frame composed by multi-resolution pyramids. Markov random Field is used to model the texture as a random selection process. A specially designed technique called the Tree-structured vector quantization is used to accelerate the selection of the best-matched pixel. The pixel-based algorithm can produce favorable synthesis results, but requires significant computational cost to carry out. The architecture of Wei and Levoy’s algorithm provides a solid foundation for my study. I developed the texture synthesis algorithm between levels of pyramids with different resolutions, which work similarly as the Gaussian pyramids. By synthesizing on a filled graph, we are no longer restricted to the scanline order since the neighborhood of each pixel can be determined. Once the texture synthesis is processed with an arbitrary order, we could make use of GPU parallel computing to break the computational work into parts in order to accelerate the synthesis process.

* 1. **Sequential CPU Implementation**

The definition of “texture” comes from a computer graphics concept called “texture mapping,” which involves applying color or bitmap onto a computer-generated model. However, the definition of “texture” in our technique could be much broader. The term can be used to describe all kinds of surface quality characteristics from either nature- or computer-generated images with repeated pattern properties [2]. In this study, textures are images composed by repeated patterns. Texture synthesis, as the name suggests, is an approach to creating more textures based on the given input. The basic idea of the synthesis algorithm is to begin with a random noise image. Each time we choose the best-fitted unit from the input image and stitch it onto the output graph, eventually filling the output canvas in the scanline order.

Pixel as the smallest controllable element in an image, is usually stored as matrix of multiple number types, most commonly (64-bit) double floating numbers in MATLAB. The color characteristic for each pixel is determined by the combination of each numeric component in the color channel. For this study, we used uin8 integer with each pixel ranging from 0 to 256. RGB images with three color channels are converted into grayscale images to reduce the computing cost.

To evaluate each subsequent candidate unit, we employ Markov Random Field, which is a model widely used in physics and probability for describing a set of variables sharing the Markov properties in a network. In texture synthesis, we use it to describe the correlation between pixels. Each pixel belongs to the local neighborhood based on its location on the output graph. Therefore, every pixel is characterized by all the pixels within its neighborhood. In our algorithm, we use a squared neighborhood N (p) to describe the locality of each pixel and the sum of difference (SSD) to measure the distance between each neighborhood.

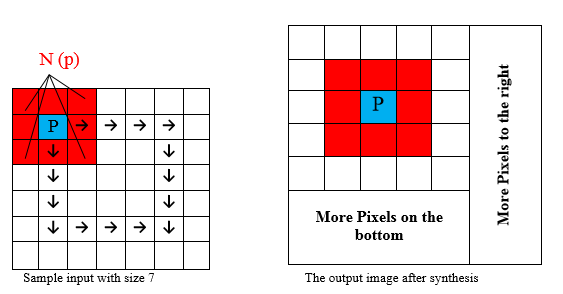


Figure . Pixel sampling from the input with sample size of 7 and neighborhood size of 3.

Although the technique described above lends itself well to generating a valid algorithm for performing texture synthesis, the process can be time-consuming since each candidate pixel requires thousands of times of calculation using the sample pixel pool from the input image. Nevertheless, we could simply improve the efficiency and quality of the original technique by applying the texture synthesis on levels of multiresolution pyramids. The idea is to downgrade the input image in terms of size and operate texture synthesis on levels with increasing size. In each pyramid level i, each input image Pin,i performs texture synthesis to determine the corresponding output image Pout,i . We start from the basis-level with images of the smallest size and proceed to the original-sized level. In each iteration, after finishing synthesizing a level of pyramid, we enlarge the image into the next level with larger size and lower resolution and then continue the texture synthesis process on the enlarged image. The algorithm could significantly improve the running time of the algorithm. This is because at each level, instead of selecting candidate pixels from a relatively large input image, we look through the pixels in a comparatively small size sublevel image after convolution and perform calculation to select the best candidate. Besides, Wei and Levoy [3] introduced a way to remove cyclic neighborhood dependency: in the process of texture synthesizing, instead of overwriting the old pixels with new ones, the new technique creates a new blank output graph and keeps the old one at the same time. This technique could also be applied to my algorithm when synthesizing at each level to enhance the quality of the image.

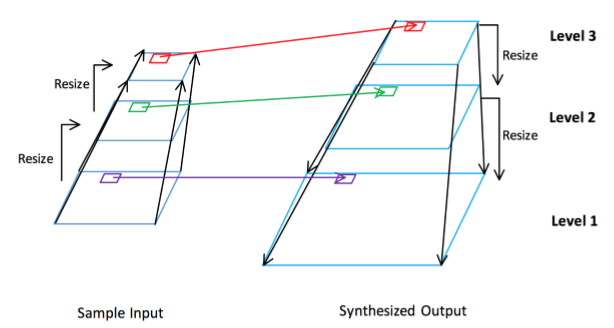


Figure . A three-level pyramid level synthesis, with the synthesis proceeding with decreasing order.

Assume that we are synthesizing an image with size L from a sample input with size A, operated on a neighborhood size of b. Each pixel is determined through calculating the sum of squared differences of A\*A neighborhoods, where each SSD requires B\*B matrix arithmetic calculation. Eventually, we traverse the SSD array and select the corresponding pixel within a proper error range of the minimum value. As the neighborhood size is usually small, we consider each SSD calculation as constant time operation. Therefore, the total runtime is approximately O (L2\*A2). In some real cases, when the size of the input image is extremely smaller than the output, the algorithm ends up with a quadratic running time O (L2).

* 1. **MATLAB Multi-Core Acceleration**

The CPU sequentail implementation could be accelerated through MATLAB’s Parallel Computing Toolbox, which is widely used for computationally intensive problems. The Toolbox works in a similar manner as OpenMP we learned in ME 759 class. The computational tasks are divided into smaller unit tasks, called “jobs”. At first, the client sends the job to the scheduler for evaluation, after which the separated tasks are distributed to the workers in the parallel working pool. During their lifetime, all workers execute the assigned task simultaneously and they are able to communicate with each other at the same time. The user could set the preferred number of workers for the parallel pool, but the actual working number of workers depends on the available cores in the operating system. [4]

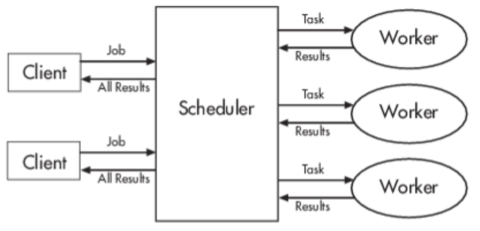


Figure . A diagram describing how MATLAB multi-core computing works.

As the most computationally intensive part of our synthesis algorithm, the scanline-order pixel selection is distributed to a pool of works for separate execution. As a result, the texture synthesis with parallel computing support exhibits a   times speedup compared to a plain sequential implementation.

1. **GPU Implementation**
   1. **CUDA Background**

Recent year, fields of high performance computing has grown significantly with increasing needs from industry and research. Graphics Processing Unit (GPU) is widely used for single instruction multiple data computation. In companion, CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA to provide APIs for developers to directly utilize the CPU computational elements for intensive computing problems. [5]

GPU has massively parallel architecture consisting of thousands smaller cores occupied by lighter weighted threads, which makes it specialized for highly data parallel computing use. However, to squeeze the best performance from GPU, some important factors need to be implemented:

1. Each separate thread need to execute tasks solely independent from each other.
2. Data copying between the host and device needs to be minimized to reduce the system memory latency.
3. Use of shared memory and coalesced memory access could improve the speed.
4. Appropriate number of threads need to be maintained to keep high occupancy of hardware.

* 1. **Texture Synthesis Acceleration with CUDA**

In our study, CUDA could be a perfect parallel computing tool to speed up the texture synthesis mainly because performing the evaluation and selection from a pool of pixels for a single target pixel is a single instruction multiple data operation. The computation involved for picking each target pixel is independent of picking the others. This important property of texture synthesis guarantees itself to be a suitable beneficiary from CUDA parallel computing. Therefore, the major implementation with CUDA support is to implement a kernel, which designates each thread to calculate SSD measure and pixel selection from the current input pool of candidates, and finally stitch the chosen pixel onto the final texture output, on each level.

With such great prospect on GPU implementation, there are some possible improvements we could made to keep our solution at high efficiency. A good improvement would be to takes advantage of the shared memory during the synthesis. It is known that while calculating SSD between the current pixel and the candidate pixels in the input candidate pools, pixels from the candidate pool array will keep repeatedly getting assessed and loaded for arithmetic operations by each thread; assessing global memory in such pattern could result in high latency. A good solution would be to load the input pool of candidates into the shared memory and keep it for the entire SSD calculation section to avoid the expensive assess to the global memory. However, one thing we must keep in mind is that the GPU stream multiprocessor might run out of space for storing candidate pixels depending on the size of the input pixel pool and hardware configuration. For example, a (compute capability 2.0) GPU card has 48 KB memory per multiprocessor. If we have a total of N blocks on the same multiprocessor, the maximum size of shared memory to hold candidate pixels is 48/N KB. Besides, a heavy use of on-chip memory as shared memory means less memory is spared for the L1 cache, which might affect the performance the program.

Another lesson from CUDA is to perform GPU computing without moving data back and forth from the host memory and device memory. Therefore, the synthesized texture needs to be kept on the device memory through different levels of synthesis. A set of different operations needs to be implemented in kernel, including initial pixel randomization, resizing texture size, noise filtering and so on. By keeping all texture operations on the kernel, we could successfully avoid creating memory overhead from unnecessary data movement.

* 1. **GPU Parallel Implementation**

The implementation is mainly divided into two parts, CPU part and GPU part. The CPU part includes a pre-processing and post-processing of texture image, and setting up the correct parameters for the CUDA kernels. As a powerful image processing library, OpenCV is used to load and output images, as well as rescaling input texture image to a texture pyramids with different level of sizes.

The GPU portion of the synthesis follows the order-independent pixel-based algorithm described above. A list of kernel functions is implemented for different purposes:

1. A random initialization kernel uses the CURAND() function to directly initialize the random pixels on device, given the ratio measured from input texture.
2. An upscale texture kernel is implemented using bilinear interpolation to resize the texture directly on device, between different levels of texture pyramids.
3. The main kernel performs the SSD calculation for each pixel with all the candidate pixels from the input pool, and randomly selects a target pixel from the best five pixels on neighborhood resemblance.
4. A box linear filter is applied on the resulting texture to average each pixel based on its neighboring pixels, in order to remove noises.

An overall working flow of the GPU implementation was visually described as following:

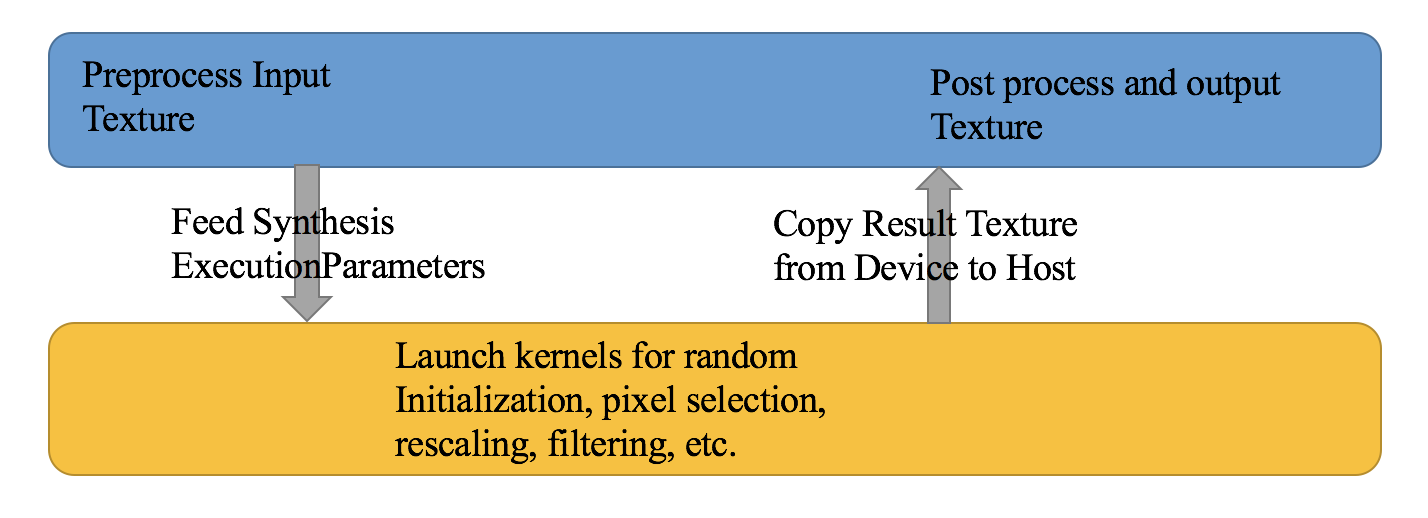


Figure . Working flow of the GPU synthesis implementation

1. **Result and Scaling Analysis**

So far, two version of orderly-independent pixel based texture synthesis has been implemented, CPU-based sequential computing with multi-core parallel acceleration and GPU-based parallel implementation. In this section, timings results are provided for comparison between two approaches and texture synthesis outputs are shown for different pyramids levels. Scaling analysis is performed specifically for GPU implementation. For all the results, the GPU Implementation is executed and benchmarked on the GeForce GTX 1060 Graphics card.

The algorithm runtime comparison between CPU and GPU implementation is included as following, an approximately 30-times speedup was observed while using GPU implementation to perform the texture synthesis. The advantage of using GPU for computation becomes more obvious as expected output texture size increases.

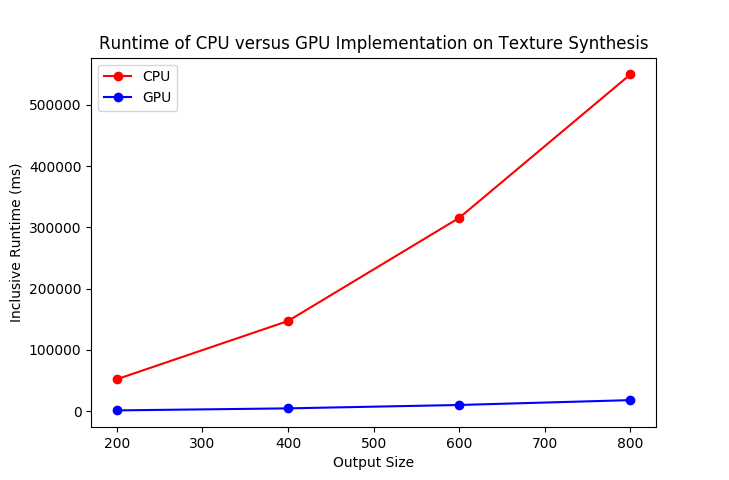


Figure . Runtime comparison between GPU and CPU implementation of synthesis

The GPU implementation of our texture synthesis not only provides incredible speedup on the synthesis process, but also ensure good quality of the end product. An array of output textures could be found as following after each iterations of synthesis:

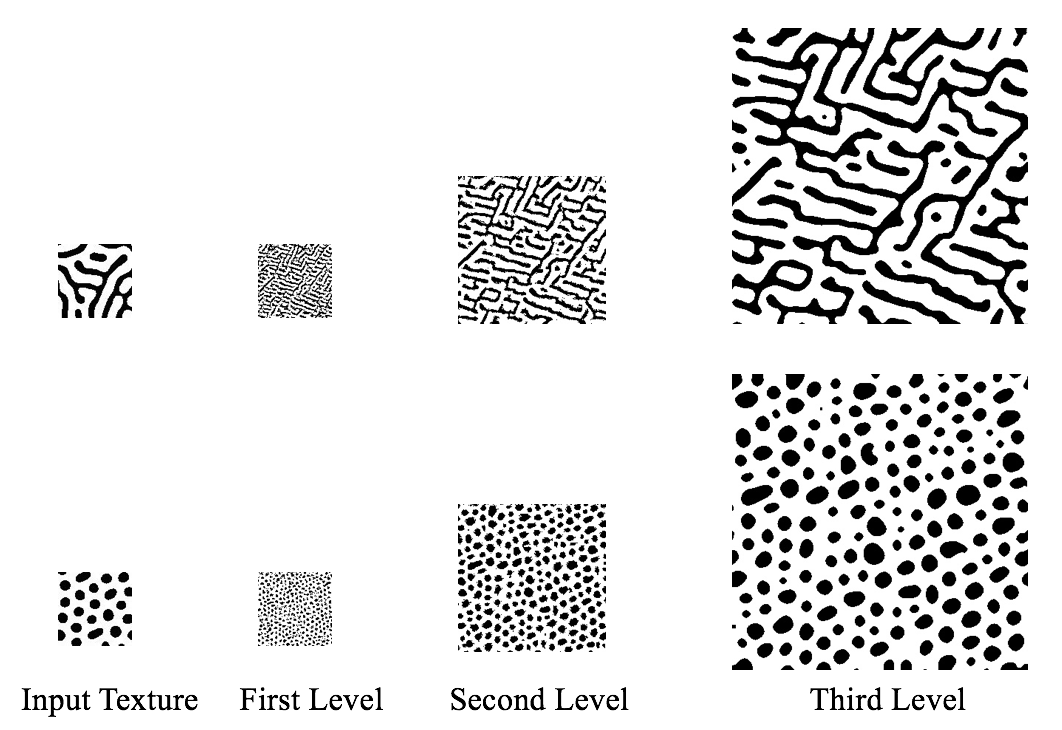


Figure . Texture synthesis results after three levels of GPU implementation

Besides, a set of scaling analysis are performed on the GPU-based implementation texture synthesis, with different synthesis parameters: output texture size, neighborhood size and iterations per level, respectively. As a result, a linearly increasing trend of GPU execution time was observed as we increase these three factors.

|  |  |
| --- | --- |
| Figure . Scaling Analysis of GPU Runtime versus Output Size | Figure . Scaling Analysis of Runtime versus neighborhood Size |
| Figure . Scaling Analysis of Runtime versus Iterations per Level |  |

1. **Conclusion**

In summary, GPU-based implementation moves the texture synthesis a great step forward as it applies the state-of-art hardware and software architecture to speed up the computationally intensive. This study, as a proof of concept, implies the huge potential of the GPU parallel computing on this area. In coming decade, with huge potential revolution on GPU, the texture synthesis could keep benefiting from it by taking advantage of its computing power. By breaking the limitation of the CPU sequential execution, the future research could gain more flexibility on developing more complex synthesis algorithms for different applications such as photography, material science, medical research, etc.

1. **References**

[1] Wei, L., & Levoy, M. (2000). Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques in SIGGRAPH ’00: Fast Texture Synthesis

Using Tree-structured Vector Quantization. New York, NY: ACM Press/Addison-

Wesley Publishing Co.

[2] Wei, L., Lefebvre, S., Kwatra, V, & Turk, G. (2009). Eurographics 2009, State of the Art Report, EG-START: State of the Art in Example-based Texture Synthesis. Munich: Eurographics Association.

[3] Wei, L., & Levoy, M. (2002). Order-Independent texture synthesis. Technical Report TR-

2002-01.

[4] Reese, J., & Zaranek, S. (2011). GPU Programming in MATLAB. Mathworks.

[5] CUDA. (2017, December 16). Retrieved December 19, 2017, from <https://en.wikipedia.org/wiki/CUDA>