CUDA based Parallel Derivation of Parametric L-system

Ji Liu, Sulan Zhang, Lingqiu Zeng, Qingsheng Zhu, Songyang Li


CPUA based Parallel Derivation of Parametric L-system

Ji Liu,1 Sulan Zhang,1 Lingqiu Zeng,1 Qingsheng Zhu,1 Songyang Li

*1College of Computer Science, Chongqing University, Chongqing 400044, China, liujiboy@gmail.com

doi:10.4156/jdcta.vol5.issue7.11

Abstract

This paper proposes an approach to derive a parametric L-system in parallel based on Compute Unified Device Architecture (CUDA). It consists of a host program running on CPU and a device program running on CUDA enabled GPU. The host program is used to transfer data between CPU and GPU, pre-allocate host and device memory, and launch the device program. The device program takes charge of deriving an L-system by computing module strings with kernel functions which represent productions of the L-system and are executed by CUDA threads in parallel. Unlike most traditional L-system derivation algorithms based on string operations, our algorithms, which use newly defined data structures to represent an L-system, can be implemented without using any string operations that are not supported by current CUDA compute capability. The given experiments illustrate that this method has better computing performance than the method using serial derivation. The parameters which affect the performance, such as the number of module strings computed by each CUDA thread and the size of each thread block, are also investigated through these experiments.

Keywords: L-system, CUDA, Parallel Computing

1. Introduction

L-system was introduced and developed in 1968 by the Hungarian theoretical biologist and botanist, Aristid Lindenmayer[1]. An L-system is a kind of string rewriting system, which is very similar to Chomsky grammars. It consists of a nonempty string representing system state, called state string in this paper, and a set of production rules. A fractal-like structure can be created by derivation. At each derivation step, production rules are applied to the state string in parallel, and then all letters in the state string are replaced by new symbols to produce a new system state.

Originally L-system was used to study the growth patterns of various types of algae, such as the blue or green bacteria Anabaena catenula. Most of recent researches, which include designing visual language framework for plant modeling[2], combining L-system and Ball B-spline curves to sketch 3D plant[3], generating plant with VRML and L-system[4] and root growth simulation[5], focused on how to use L-system to simulate plant growth[6] and to generate computer graphics which have fractal structure. Due to the recursive nature, applying L-systems to model large and detailed fractal structures often leads to high computation time requirement. In order to address this issue, the resent researches have also introduced some solutions to derive L-system on parallel architectures like GPUs[7] or multi-core CPUs, and indicated that it is possible to accelerate derivation of L-system on newly emerged Compute Unified Device Architecture (CUDA)[8].

CUDA is a GPGPU architecture specialized for highly parallel computation. A common CUDA program consists of functions called kernels which are executed N times in parallel by N different CUDA threads. Because thousands of threads can be executed by CUDA at the same time, CUDA is proved to be much faster than traditional parallel architectures at the same cost[9]. There have been already some compute-intensive problems solved by this architecture, such as computing particle swarm optimization[10], simulating 3D tissue deformation in real-time[11] and solving stochastic differential equations[12]. However, when designing CUDA based parallel derivation of L-system, most traditional L-system derivation algorithms based on string operations cannot be transferred to CUDA without special modification. The reason is that many string operations are not supported by current CUDA compute capability. There are also some other problems should be taken into consideration due to the differences between CUDA and other parallel architectures, such as how to allocate memory,
how to scatter and gather data between CPU and CUDA enabled GPU, how to determine the number of CUDA threads, and how to apply productions in parallel to rewrite the state string using CUDA supported operations. To our opinion, within the scope of our knowledge, these problems have been not well addressed.

This paper will present a method to derive the L-system in parallel using a host program and a device program. The host program running on CPU is responsible for allocating memory, transferring data and launching the device program while the device program running on CUDA enabled GPU derives an L-system by interpreting its productions with kernel functions. To fulfill the method, new data structures of a parametric L-system are also defined for representing L-system and data transferring. Several experiments are used to test our method. The results prove that the proposed method is more efficient than traditional serial L-system generation algorithms.

This paper is organized as the following: we will define the parametric L-systems and its data structures which will be used in our algorithms in section 2; Section 3 will describe the parallel algorithms in detail; Section 4 will show experiment results; Section 5 will give a conclusion and discuss some future works.

2. Parametric L-systems and Data Structures

2.1. Parametric L-systems

To simplify discussion, this paper discusses L-systems which are context-free, parametric and determined. A parametric L-system can be defined as an ordered quadruplet:

\[ G = (V, \Sigma, \omega, P) \]  

Where

- \( V \) is the alphabet containing a set of letters.
- \( \Sigma \) is the set of formal parameters.
- \( \omega \in (V \times \mathbb{R}^*) \) is state string of the L-system. The initial state string is called axiom. \( R \) is the set of real numbers. Letter \( A \in V \) and parameters \( a_1, a_2, ..., a_n \in R \) are denoted by \( A(a_1, a_2, ..., a_n) \) called module string. A state string is a set of model strings consisting of letters with associated parameters.
- \( P \subset (V \times \Sigma \times \mathbb{R}^*) \times C(\Sigma) \times (V \times E(\Sigma))^* \) is a finite set of productions. \( C(\Sigma) \) denotes a logical expression with parameters from \( \Sigma \), and \( E(\Sigma) \) is an arithmetic expression with parameters from the same set.

A production \( (A(t), C(t), \chi(t)) \in P \) is written as \( A(t):C(t) \rightarrow \chi(t) \), where \( A(t) \in V \), \( t \in \Sigma^* \), \( A(t) \in (V \times \Sigma^*) \), \( C(t) \in C(\Sigma) \) and \( \chi(t) \in (V \times E(\Sigma))^* \). \( A(t), C(t) \) and \( \chi(t) \) are called the predecessor, the condition and the successor of this production, respectively. If \( A_1(a_1)A_2(a_2)\ldots A_n(a_n) \) is the axiom, and for each \( A_i(a_i) \in \omega \) there is a production \( A_i(t):C_i(t) \rightarrow \chi_i(t) \). If \( C_i(t) \) is true \( (t=a_i) \), then after one step derivation, a string \( \chi_1(a_1)\chi_2(a_2)\ldots \chi_n(a_n) \) will be generated as the new state.

If \( p_1, p_2, ..., p_k \in P \) and \( P \subset P \), where

- \( p_1: A(t):C_1(t) \rightarrow \chi_1(t) \)
- \( p_2: A(t):C_2(t) \rightarrow \chi_2(t) \)
- ...
- \( p_k: A(t):C_k(t) \rightarrow \chi_k(t) \)

Productions \( p_1, p_2, ..., p_k \) have the same predecessor, but different conditions and successors. These productions can be written as:

\[ p = (A(t) \rightarrow \{C_1(t) \rightarrow \chi_1(t)\}, \{C_2(t) \rightarrow \chi_2(t)\}, ..., \{C_k(t) \rightarrow \chi_k(t)\}) \]  

The above notation will be used in the following sections, and \( A(t), C(t) \) and \( \chi(t) \) are defined to denote the predecessor, \( i \)-th condition and \( i \)-th successor of the production \( p \), respectively. This paper aims to design a method which can derive the following kind of L-system in parallel:
An example of parametric L-system using the above definition is given as below:

\[
\omega: B(2)A(4, 4)
\]

\[
p_1: A(x, y) \rightarrow \{y \leq 3: A(x*2, x+y)\} \{y > 3: B(x) A(x/y, 0)\}
\]

\[
p_2: B(x) \rightarrow \{x < 1: C(x)\} \{x \geq 1: B(x-1)\}
\]

The parametric L-system (2.3) generates sequence of strings:

- \(B(2)A(4,4)\)
- \(B(1)B(4)A(1,0)\)
- \(B(0)B(3)A(2,1)\)
- \(C(0)B(2)A(4,3)\)
- \(C(0)B(1)A(8,7)\)

...  

2.2. Data structures

Based on the above definition, we give data structures used in the parallel algorithms.

1. Module String (MS)

It is used to denote a module string, such as \(A(1,2,3)\), and defined as:

\[MS = (L, P, S)\]

Where

- \(L\) is the letter of a module string.
- \(P\) is an array of associated parameters. \(P[i]\) means the i-th parameter.
- \(S\) is the size of parameters.

The module string \(A(1,2,3)\) is stored as figure 1(a).

2. Module String List (MSL)

It is used to denote a list of module string, such as \(A(1,2,3)B(4,5)\), and defined as:

\[MSL = (M, S)\]

Where

- \(M\) is an array of module strings. \(M[i]\) means the i-th module string.
- \(S\) is size of module string list.

The list of module string \(A(1,2,3)B(4,5)\) is stored as figure 1(b).

3. State String (SS)

State string represents the state of an L-system. As state string is a list of module strings, it can be defined as:

\[SS = MSL\]

4. Generated Successor (GS)

\(A(a)\) is a module string, and \(A(t): C(t) \rightarrow \chi(t)\) is a production. When \(t=a\) and \(C(t)\) is true, then \(A(a)\) will be replaced by successor \(\chi(a)\) which is called generated successor in this article. \(\chi(a)\) is a list of module strings, so generated successor can be defined as:

\[GS = MSL\]
(5) Generated Successor List (GSL)

In the first derivation step of parametric L-system (2.3), state string \( B(2)A(4, 4) \) is replaced by \( B(1)B(4)A(1,0) \). \( B(1) \) and \( B(4)A(1,0) \) are generated successors, so \( B(1)B(4)A(1,0) \) is a list of generated successors. Generated successor list can be defined as:

\[
GSL = (G, S)
\]

Where:
- \( G \) is an array of generated successors. \( G[i] \) means the \( i \)-th generated successor.
- \( S \) is size of generated successor list.

The state string \( B(2)A(4, 4) \) and its generated successor list \( B(1)B(4)A(1,0) \) are stored as figure 1(c).

![Figure 1. Data structures and their storage](image)

(a) module string \( A(1,2,3) \) (b) module string list \( A(1,2,3)B(4,5) \) (c) state string \( B(2)A(4, 4) \) and its generated successor list \( B(1)B(4)A(1,0) \)

3. Parallel Algorithms

3.1. Overview

As illustrated in figure 2, the parallel algorithms contain two parts: a host program running in CPU to execute serial codes and a device program running in GPU with \( N \) CUDA threads to do parallel computing. Both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. The derivation of an L-system begins with executing serial codes in host side. After allocating host and device memory, data needed for device program are copied from host memory to device memory. Then the device program is launched by host program. The device program consists of kernel functions which are executed by CUDA threads. When threads executing complete, results produced by kernels should be gathered and transferred back to host. The following execution sequence is a brief overview of parallel derivation. In section 3.2 and section 3.3, we will give more details of the host and device program.

1. State string is stored in host memory as a module string list. The initial state string is axiom \( \omega \).
2. Allocate memory in device and copy state string from host memory to device memory. The allocated memory size is the same as the size of state string in host memory.
3. Allocate device memory for generated successor list. Generated successor list is used to store the derivation result in device program.
4. Start kernel to derive the L-system. The kernel functions are built form productions of L-system.
5. Parallel derive in GPU and store results in generated successor list.
Allocate memory in host and copy generated successor list from device memory to host memory. The allocated memory size is the same as the size of generated successor list in device memory.

Use generated successor list to create a new state string.

Go to step (2) to begin next derivation.

### Input L-system

\[ \omega : A \rightarrow A A \rightarrow A A \rightarrow \cdots \]

\[ p_1 : A \rightarrow C C \rightarrow C C \rightarrow \cdots \]

\[ p_2 : A \rightarrow C C \rightarrow C C \rightarrow \cdots \]

\[ \vdots \]

\[ p_n : A \rightarrow C C \rightarrow C C \rightarrow \cdots \]

### Figure 2.
The process of parallel derivation of L-system based on CUDA

### 3.2. Host Program

For easy reference, variables and functions used in the host program are listed in Table 1.

<table>
<thead>
<tr>
<th>Table 1.</th>
<th>Table of variables and functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_{SS} )</td>
<td>State string stored in host memory</td>
</tr>
<tr>
<td>( D_{SS} )</td>
<td>State string stored in device memory</td>
</tr>
<tr>
<td>( H_{GSL} )</td>
<td>Generated successor list stored in host memory</td>
</tr>
<tr>
<td>( D_{GSL} )</td>
<td>Generated successor list stored in device memory</td>
</tr>
<tr>
<td>( \text{SIZE OF SS} )</td>
<td>Size of state string</td>
</tr>
<tr>
<td>( \text{SIZE OF GSL} )</td>
<td>Size of generated successor list</td>
</tr>
<tr>
<td>( \text{TOTAL THREAD} )</td>
<td>The number of CUDA threads</td>
</tr>
<tr>
<td>( \text{MS_PER_THREAD} )</td>
<td>The number of module strings computed by each CUDA thread</td>
</tr>
<tr>
<td>( DB )</td>
<td>The size of thread block</td>
</tr>
<tr>
<td>( DG )</td>
<td>The size of thread grid</td>
</tr>
<tr>
<td>( h_{\text{allocate}}(V, S) )</td>
<td>Function used to allocate host memory for parameter ( V ), where parameter ( S ) represents the size of allocated space</td>
</tr>
<tr>
<td>( d_{\text{allocate}}(V, S) )</td>
<td>Function used to allocate device memory for parameter ( V ), where parameter ( S ) represents the size of allocated space</td>
</tr>
</tbody>
</table>

The host program is defined as Algorithm 1 whose input parameters are the axiom \( \omega \), the number of module strings per thread \( \text{MS\_PER\_THREAD} \), the thread block size \( DB \) and the output is a new...
state string which will be the initial string for the next iteration.

**Algorithm 1:** Host (ω, MS_PER_THREAD, DB)
1. \( \text{SIZE_OF_SS} \leftarrow \text{size of state string } \omega \)
2. \( h \_\text{allocate}(H \_SS, \text{SIZE_OF_SS}) \)
3. \( H \_SS \leftarrow \omega \)
4. \( d \_\text{allocate}(D \_SS, \text{SIZE_OF_SS}) \)
5. \( D \_SS \leftarrow H \_SS \)
6. \( \text{SIZE_OF_GSL} \leftarrow \text{size of generated successor list} \)
7. \( d \_\text{allocate}(D \_GSL, \text{SIZE_OF_GSL}) \)
8. \( \text{TOTAL_THREAD} \leftarrow \left\lfloor \frac{\text{SIZE_OF_SS}}{\text{MS_PER_THREAD}} \right\rfloor \)
9. \( DG \leftarrow \frac{\text{TOTAL_THREAD}}{DB} \)
10. \( \text{Derive}(<DG, DB>>(D \_SS, D \_GSL, \text{SIZE_OF_SS}, \text{MS_PER_THREAD})) \)
11. \( h \_\text{allocate}(H \_GSL, \text{SIZE_OF_GSL}) \)
12. \( H \_GSL \leftarrow D \_GSL \)
13. \( H \_SS \leftarrow \text{BuildNewSS}(H \_GSL) \)
14. \( \text{SIZE_OF_SS} \leftarrow H \_SS \_S \)
15. \( \text{output } H \_SS \)
16. goto 4

Because of the CUDA programming model’s limitation that device memory cannot be dynamically allocated in kernels, the host is responsible for allocating, freeing device memory and transferring data between host and device memory. According to the production of L-system grammar, there is only one generated successor for each preceding module string, so the size of a generated successor list (\( \text{SIZE_OF_GSL} \)) is determinate and equals to the number of predecessors (\( \text{SIZE_OF_SS} \)). However, the device memory used to store generated successor list (\( D \_GSL \)) should be allocated before the number of parameters in a module string and the number of module strings in a generated successor are certain. In order to solve this problem, we assume that the size of module string and the size of generated successor are constants which are large enough to store the max number of parameters in a module string and the max number of module strings in a generated successor respectively. To avoiding the waste of memory, the former allocated memory should be freed before next iteration.

The device program is started by the following execution configuration syntax:

\[ \text{Derive}(<DG, DB>>(D \_SS, D \_GSL, \text{SIZE_OF_SS}, \text{MS_PER_THREAD})) \]

Derive is a kernel function defined using the \_global\_ declaration specifier and can be called from the host only. This function will be generated by Algorithm 3 described in section 3.3. Because the CUDA programming model organizes threads as a grid of thread blocks, the parameter \( DB \) and \( DG \) specify the size of a thread block and the size of a thread grid. \( DG \) is determined by \( \text{TOTAL_THREAD} \) and \( DB \). \( \text{TOTAL_THREAD} \) depends on the size of state string (\( \text{SIZE_OF_SS} \)) and the number of module strings computed by each CUDA thread (\( \text{MS_PER_THREAD} \)). So the performance of kernel execution will be affected by different \( DB \) and \( \text{MS_PER_THREAD} \) value. We will discuss this problem in section 4.

When the given kernel call has been returned from the former algorithm, Algorithm 2 is recommended for collecting module strings from generated successor list and building a new state string for next iteration or rendering on the screen.

**Algorithm 2:** BuildNewSS (\( H \_GSL \))
1. \( \text{SIZE_OF_SS} \leftarrow 0 \)
2. for \( i \) from 1 to \( H \_GSL \_S \)
3. \( \text{SIZE_OF_SS} \leftarrow \text{H \_GSL \_G[i].S} \)
4. \( h \_\text{allocate}(H \_SS, \text{SIZE_OF_SS}) \)
5. \( H \_SS \_S \leftarrow \text{SIZE_OF_SS} \)
6. \( j \leftarrow 0 \)
3.3. Device Program

For a parametric L-system with following productions:

\[ p_1 : A_1(t) \rightarrow [C^1_1(t) : X^1_1(t)] [C^1_2(t) : X^2_1(t)] \ldots [C^n_m(t) : X^n_m(t)] \]

\[ p_2 : A_2(t) \rightarrow [C^2_1(t) : X^1_2(t)] [C^2_2(t) : X^2_2(t)] \ldots [C^n_m(t) : X^n_m(t)] \]

\[ \ldots \]

\[ p_n : A_n(t) \rightarrow [C^n_1(t) : X^1_n(t)] [C^n_2(t) : X^2_n(t)] \ldots [C^n_m(t) : X^n_m(t)] \]

Kernel functions Derive and KA_1, KA_2, ..., KA_n, which are executed by CUDA threads in parallel can be built from Algorithm 3 and Algorithm 4. Kernel function Derive is defined using __global__ declaration specifier and used to start parallel derivation. Kernel functions KA_1, KA_2, ..., KA_n defined using __device__ declaration specifier represent the production \( p_1, p_2, \ldots, p_n \) respectively and are called by Derive to produce generated successors according to predecessor \( A_1, A_2, \ldots, A_n \).

Algorithm 3: Derive (D_SS, D_GSL SIZE_OF_SS, MS_PER_THREAD)

1. threadID ← blockDim.x * blockIdx.x + threadIdx.x
2. begin ← threadID*MS_PER_THREAD
3. end ← min{begin + MS_PER_THREAD, SIZE_OF_SS}
4. for i from begin to end
5.  MS ← D_SS[i]
6.  for j from 1 to n
7.    if MS.L == 'A_j' 
8.     D_GSL[i] ← KA_j(MS.P)
9.  goto 4
10. end if
11. end for
12. end for

When kernel function Derive is started, multi-threads are launched at the same time. As the blocks and grid in device are arranged in one dimension in our algorithms, the total number of CUDA threads equals to DB*DG. Each CUDA thread executes the same kernel function Derive, so the first step of the Algorithm 4 is to calculate threadID based on built-in variables blockDim, blockIdx and threadIdx. With threadID and variable MS_PER_THREAD, module strings stored in D_SS will be scattered to different CUDA threads which can be indexed by threadID and have the responsibility to compute the assigned module strings. The parameter MS_PER_THREAD has a significant impact on the efficiency of the algorithm. In the experimental part (section 4), we will discuss about how this parameter affect performance.

CUDA thread uses kernel functions KA_1, KA_2, ..., KA_n, which are selected according to the letter of a module string to derive assigned module strings. Function KA_j corresponds to predecessor \( A_j \) and production \( p_j \), KA can be generated by Algorithm 4.

Algorithm 4: KA_j (MS.P)

1. create a new GS
2. for k from 1 to m
3.  if C_k(MS.P) is true
4.    GS ← X_k^j(MS.P)
5.  goto 8
6. end if
7. end for
8. return GS
As described in the above algorithms, the device program uses device memory which is pre-allocated by host program. The only exception is the first step of Algorithm 4 which creates a new GS in thread’s private local memory. This operation is different from dynamically allocating device memory and is supported by CUDA. Unlike traditional L-system derivation algorithms, the device program is not based on string rewriting mechanism. Because many of the string rewriting operations cannot be supported by current CUDA compute capability. In our algorithms, L-system is not represented as a string, but stored in the newly defined data structures. With these data structures, Algorithm 3 and Algorithm 4 can be easily implemented using basic arithmetic, logical and condition operations without using any string operations that are not supported.

4. Experiments and Discussions

Although L-system is parallel in nature, the traditional derivation algorithms are often implemented in serial. That means an L-system with \( k \) letters in state string needs \( k \) replacements during a derivation process. If each letter in the state string is replaced by a new string whose length is at least \( m \) in each derivation step, it needs \( km^i \) replacements in the \( n \)-th derivation. So the computation time required by the serial ones may grow exponentially when L-system becomes complicated. However, what is new in our method is that we replace letters in parallel. With new method, an L-system with \( k \) letters in state string needs \( k/\text{TOTAL\_THREAD} \) replacements during a derivation process. We used six experiments to test this proposed parallel algorithm. The first and second experiments are used to verify how the parameters \( \text{MS\_PER\_THREAD} \) and \( \text{DB} \) affect the efficiency of the algorithms. The other experiments are used to investigate whether there are some computing performance advantages over the traditional serial derivation algorithms. And at each test, we measured the derivation time in microsecond (\( \mu s \)) and the number of computed module strings during iteration.

The serial algorithms are implemented on Intel Pentium 4 3.0GHz and Core 2 Due 2.4GHz CPU. And CUDA based parallel derivation is implemented by Geforce GTS 250 GPU, which has 128 stream processors. All the parametric L-systems, which can also be found in the book *The Algorithmic Beauty of Plants* [1], used in our experiments are listed below:

Experiment 1, 2 and 4 used the L-system which generated Koch island, see Figure 3(a); Experiment 3 used the Parametric L-system (2.3) in this paper; Experiment 5 used the L-system which generated Fractal tree, Figure 3(b); Experiment 6 used the L-system which generated 3D Hilbert curve, see Figure 3(c).

![Figure 3. L-system used in experiments. (a) Koch island (b) Fractal tree (c) 3D Hilbert curve](image)

**Experiment 1:** This experiment tested the efficiency of the algorithms with different \( \text{MS\_PER\_THREAD} \) value and \( \text{DB}=256 \). The experiment results are showed in Figure 4(a). The results conclude that the more threads are utilized by the parallel algorithms, the higher efficiency will be got, and especially when a thread just takes charge of one module string’s derivation the efficiency will be the best. This is very different from parallel derivation on multi-core CPUs, where the performance will decrease rapidly when the number of threads exceeding a threshold.

**Experiment 2:** This experiment was implemented to find the relationship between the parameter \( \text{DB} \) and the efficiency of the algorithms proposed in this paper. In this experiment,
MS_PER_THREAD=1 and block size was increasing by 2 times each time. The curves of computing performance with different block size almost overlap with each other in Figure 4(b), so the block size (DB) has little effect on the computing performance of the proposed method.

**Experiment 3, 4, 5, and 6**: In order to compare the performance of the proposed algorithms with the serial algorithms on single-core CPU (Intel Pentium 4 3.0GHz) or multi-core CPU (Core 2 Due 2.4GHz), the experiments were designed to derive four common L-systems. In this part MS_PER_THREAD and DB were assigned as constants value of 1 and 256. As the results showed in the Figure 4 (c), (d), (e) and (f), our algorithms have better performance over the serial algorithms when the number of module strings is large, but there is no significant advantage in efficiency when the number is very small. And serial algorithms running on multi-core CPU still have better computing performance than single-core CPU.

**Figure 4.** Experiments results. (a) Derivation times with different MS_PER_THREAD value (b) Derivation times with different DB value (c) Derivation times of parametric L-system (2.3) (d) Derivation times of L-system which generates Koch island (e) Derivation times of L-system which generates Fractal tree (f) Derivation times of L-system which generates 3D Hilbert curve

In the process of implementing our algorithms, we also noticed two problems which need further
discussion. To use CUDA, state string must be transferred from the host to the device along the PCI Express bus which is costly in terms of performance, especially when the length of state string is very long. In order to minimize the data transferring time, our algorithms can be improved by keeping the intermediate module strings for the next iteration on the device until the expected result is achieved. Another problem is that the device memory which can be used by GPU is much smaller than the host memory which can be used by CPU. For example, the current graphics cards mostly have 512MB memory while the memory size of CPU can be up to 4G or more. In the proposed algorithms, there is also some waste of memory in order to pre-allocate device memory in host program. So when state string is too long, there will be not enough device memory to run the device program. To solve this issue, a state string can be split into several smaller parts computed by the device program separately. Then different derivation results can be combined as a whole state string for next iteration or output. Obviously, this modified version decreases some performance, but it is still faster than serial algorithms.

5. Conclusions and Future works

In this paper, we proposed new parallel algorithms for the derivation of parametric L-system under CUDA programming model. The presented algorithms involve two parts of code running on different platforms concurrently: a host program with single-core or multi-core CPUs and a device program with CUDA-enabled GPU. In host program, we designed algorithms to transfer data between CPU and GPU, allocate host and device memory, and start device program. In device program, we designed data structures for representing an L-system and convert productions of the L-system into a set of kernel functions which are executed in the GPU using operations supported by CUDA compute capability to derive the L-system in parallel and generate a new state string. A set of experiments demonstrated that our method has better computing performance than the serial L-system generation algorithms and the more threads are launched, the higher performance can be got.

Future works: Next we will extend our algorithms to a broad set of L-systems including stochastic L-systems, and context-sensitive L-systems. At the same time we would like to integrate the parallel derivation of L-system in a rendering engine to render a complex L-system graphically.

Acknowledgements: This work has been supported by Natural Science Foundation Project of CQ CSTC 2010BB2213 and Fundamental Research Funds for the Central Universities CDJRC10180010.

6. References

CUDA based Parallel Derivation of Parametric L-system
Ji Liu, Sulan Zhang, Lingqiu Zeng, Qingsheng Zhu, Songyang Li