



جامعة الملك فهد للبترول والمعادن
King Fahd University of Petroleum & Minerals

ICS 535: Design and Implementaion of
Programming Languages
Semester 111

Project Report

A CUDA C Implementation of a Parallel Parser for RNA Secondary Structures based on Bison-generated GLR Parser

By:
Omar Alzuhaibi

Supervisor:
Dr. M.S. Al-Mulhem

14 January, 2012

Pretext

This report is a record detailing my personal experience working with CUDA in implementing the algorithm in [2] in parallel. My aim of this project revolves totally around the need for a multiprocessing workbench for this specific type of problems, i.e., parsing RNA secondary structures.

Therefore, the reader's expectation of this report should be confined within the subject of feasibility of multiprocessing in RNA problems.

The source code can be found on sf.net/projects/cudaswallow or github.com/omarzd/CudaSwallow

To download a PDF version of this report, click [here](#).

1 Introduction

1.1 Parsing Problem

Parsing has been used to syntactically analyze programming languages. Usually, parsing programming languages is **deterministic**, i.e., there exists exactly one valid way to syntactically interpret the input. In this project, non-deterministic parsing is used as means to predicting RNA secondary structures. In **non-deterministic** parsing, more than one syntactic interpretation is valid. This is where this project comes in. The many different possible interpretations, called **parses**, each of which produces a **parse tree**, are computed in parallel using the CUDA environment.

2 . Algorithm

2.1 The Original Algorithm

Following is the algorithm exactly as in [2]:

Algorithm 1 The parsing algorithm

Input: An RNA sequence w and a parsing table with two parts ACTION and GOTO for a stochastic context-free grammar G

Output: If w is a valid RNA sequence, the parse tree with highest probability for w ; otherwise, error message. let a be the first symbol of w ;

```
while 1 do
    let  $s$  be the state on top of the stack;
    if  $ACTION[s, a] = \text{shift } t$  then
        push  $t$  onto the stack;
        let  $a$  be the next input symbol;
    else if  $ACTION[s, a] = \text{reduce } A \rightarrow \beta$  then
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push  $GOTO[t, A]$  onto the stack;
        output the production  $A \rightarrow \beta$ ;
    else if  $ACTION[s, a] = \text{shift } t$ , and  $\text{reduce } A \rightarrow \beta$  then
        create a thread to parse with the shift move;
        push  $t$  onto the stack;
        let  $a$  be the next input symbol;
        create another thread to parse with the reduce move;
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push  $GOTO[t, A]$  onto the stack;
        output the production  $A \rightarrow \beta$ ;
    else if  $ACTION[s, a] = \text{reduce } A \rightarrow \beta$ , and  $\text{reduce } C \rightarrow \gamma$  then
        create a thread to parse with the first reduce move;
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push  $GOTO[t, A]$  onto the stack;
        output the production  $A \rightarrow \beta$ ;
        create another thread to parse with the second reduce move;
        pop  $|\gamma|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push  $GOTO[t, C]$  onto the stack;
        output the production  $C \rightarrow \gamma$ ;
    else if  $ACTION[s, a] = \text{accept}$  then
        compute the probability;
        print parse tree and its probability;
        break;
    else
        error;
    end
end
```

print parse tree with max probability;

2.2 Relation to the GLR Parser

The algorithm in 2.1 is very similar to the well-known bottom-up GLR parser. This section will explain GLR based on LALR and why it is not quite suitable for our

application.

A Generalized LR (**GLR**) parser functions exactly like a Look-Ahead LR (**LALR**) parser. However, when it encounters a conflict due to ambiguity in the grammar, it essentially divides into multiple parses according to the possible options at that point. It continues with all options in tandem. Along the way, most parses will fail due to encountering an unexpected token, which simply means they took the wrong option earlier. Eventually, assuming the input is syntactically valid, only one parse should survive; and that would be the correct parse of the input.

GLR deals with ambiguity in a straightforward exact simple solution. However, even if a parse, upon conflict, happened to continue on with the right option, it would fail as soon as it faces a new conflict before the first is resolved. And that is perfect for parsing programming languages, because only one parse tree should be the correct one, i.e., the one intended by the programmer. On the other hand, for the case of RNA sequences, all successful parses are valid parse trees. This is why the algorithm described and implemented in [2] is slightly different from GLR in that a parse would divide and divide indefinitely every time it faces a conflict.

2.3 Parallelization of the Algorithm

There are many ways to write a parallel algorithm for this problem. Although this project implements the exact same algorithm in [2] without major modification, this subsection serves as an introduction to suggested solutions discussed in 5.3.

Generally for parallelization, the technique is to examine the sequential algorithm, identify a task that has one or more extra dimensions embedded in it, i.e., a task that can be done in parallel; then find the parameters that would define the extra dimension(s), and transform the algorithm from a 1D sequence of commands into 2D or more, where every dimension contains a sequence and parameters completely independent from the others. This may sound too abstract, but the parallel algorithm itself is very simple.

In fact, The algorithm proposed in [2] already utilizes a multi-threaded approach. For this project, the algorithm has been the same but the implementation added a multi-processing overhead caused by a very essential property of the CUDA architecture, i.e., CUDA cores cannot communicate with each other; this issue will be discussed in detail in 3.2-3.5.

Unfortunately, the algorithm as it is has only one degree/dimension/level of parallelization. Since it is based on LR, it must traverse the whole input string sequentially, token by token. An alternative approach will be discussed in 5.3.

2.4 Grammar

The grammar used in [2], also shown in 3.1 as *grammar 1*, is definitely nondeterministic. A grammar is **nondeterministic** when, at any point of an input, no fixed number of look-ahead tokens would suffice to determine the next grammar rule to reduce by [3]. And this is the case for *grammar 1*. Therefore, even a LR(infinity) parser cannot parse such grammar. it must be parsed using a GLR-based parser.

2.5 Time and Space Complexity

The exact expression of time complexity depends upon the grammar and the input. Moreover, the approximate expression depends on the grammar. So, let us work out the worst case complexity. The worst case would be an infinitely ambiguous input, i.e., m valid actions can be taken in every state for every token. Call m the branching factor. Then, every parse would divide to m parses at every iteration. This means that the number of parses will be multiplied by m at every iteration. At the i -th iteration, there will be m^i parses. Also, in the worst case, the number of iterations can reach double the size of the input n . Therefore, the time needed would be $O(m^{2n})$. This case is not even realizable by any grammar because it does not account for error cases and failing parses! We need a more accurate analysis. But first, assume that we have a machine capable of running a maximum of m^{10} processes completely in parallel, i.e., its maximum capacity c of performing a linear-time run is an input of size 5, then for $n \leq 5$, it would need constant time per iteration, since processing a parse at any given iteration takes constant time. Then, the whole run would take $2Cn$, where C is a constant representing the time a single shift or reduce operation would take. Suppose now we choose $n = 6$. Let $m = 2$ for simplicity. That would raise the number of parallel processes in the last two iterations to 2^{11} and 2^{12} respectively, but the machine is capable of running only 2^{10} processes in parallel. So, it will sequentially run 2 times for the previous-to-last iteration and 4 times for the last iteration, i.e., the time for every iteration i is:

$$T(i) = 2^i \quad \forall \quad i > 10 \quad (1)$$

and Generally, the time for one iteration is:

$$T(i) = \begin{cases} C m^i & \forall \quad i > 2c \\ C & \forall \quad i \leq 2c \end{cases} \quad (2)$$

In big-O notation:

$$T(i) = \begin{cases} O(m^i) & \forall i > 2c \\ O(1) & \forall i \leq 2c \end{cases} \quad (3)$$

The machine's capacity c is calculated as follows:

$$\left\lceil \frac{\log_m N}{2} \right\rceil \quad \dots(4)$$

where N is the maximum number of processes that can run independently in parallel, in our case, the number of CUDA cores.

Generally, for a machine capable of handling a maximum input of size c completely in parallel, the total time it takes is:

$$T(n) = \begin{cases} \sum_{i=1}^{2c} C + \sum_{i=2c+1}^{2n} C m^{i-2c} & , n > c \\ \sum_{i=1}^{2n} C & , n \leq c \end{cases} \quad (5)$$

$$T(n) = \begin{cases} C(2c + m^{2n-2c+1} - 1) & , n > c \\ C 2c & , n \leq c \end{cases} \quad (6)$$

In big-O notation:

$$T(n) = \begin{cases} O(m^{n-c}) & , n > c \\ O(n) & , n \leq c \end{cases} \quad (7)$$

it is safe to assume that m , the branching factor, is around 2. The exact value is calculated by averaging the number of actions in every cell of the action table. Using the table used in this project, found in appendix B, the average value of m would evaluate to 2.318. So, for a machine with capacity $c = 0$, i.e., no parallel processing capability, the worst case complexity would be $O(2.3^n)$.

This result is not unexpected. In fact it is perfectly coherent with [5] where it says that for such non-deterministic grammar, the parser would need, in the worst case, exponential time.

The important thing to note here is the big jump from linear time to exponential time when the number of parses to process is more than the number of cores available; and that the number of cores needed to keep performance in linear time grows

exponentially with respect to the input size. Therefore, it is not feasible to try and keep up with this algorithm's requirement. Instead, as explained in 2.3, a new dimension of parallelization must be introduced for this problem. Suggested solutions are discussed in 5.3.

As for space complexity, the algorithm needs space to store stacks and parse trees. Again, in the worst case, $O(m^n)$ stacks are needed, each of which can hold up to $2n$ bytes. That is nearly $O(n 2^n)$ bytes in the worst case. The experimental results in 4.1 show that throughout a whole run, the maximum number of active stacks at any one time is linearly proportional to the total number of parse trees generated; and is exponential with respect to the input size.

The space needed to store parse trees is exactly the same number of times a shift or a reduce action has been taken, i.e., $O(m^n)$ in the worst case.

In [5], a similar parsing problem is solved in $O(n^3)$ using $O(n^2)$ space. However, such a solution depends on splitting and merging functions, which are properties essential to the sequential implementation of GLR parsers, defined in a certain way that would limit possibilities, thereby killing some potential parses early on. This project's implementation depends on parsing in a totally non-deterministic fashion, boastfully examining all possibilities backed by the power of parallel processing, which, alone, proved not to be enough for an efficient solution.

All the analysis done here is considering the worst case. The experimental results are in 4.1-4.2.

3. Implementation

3.1 Tweaking the Grammar

The grammar implemented in [2] is as follows:

Rule No.	Production	Probability
0	$S' \rightarrow S$	1
1	$S \rightarrow LS$	0.4
2	$S \rightarrow \epsilon$	0.6
3	$L \rightarrow a$	0.18
4	$L \rightarrow c$	0.09
5	$L \rightarrow g$	0.1
6	$L \rightarrow u$	0.17
7	$L \rightarrow aSu$	0.07
8	$L \rightarrow uSa$	0.09
9	$L \rightarrow cSg$	0.14
10	$L \rightarrow gSc$	0.11
11	$L \rightarrow gSu$	0.02
12	$L \rightarrow uSg$	0.03

Grammar 1: Grammar implemented in [2]

There is a number of observations regarding this grammar.

First, this grammar introduces a high degree of ambiguity and has no precedence rules. And as mentioned earlier, the goal is to output all possible parse trees. Therefore, it is not suitable to use traditional parser generators, such as Bison, that have been perfected for grammars of programming languages. The plan is to build a parser nearly from scratch and then generalize it into a parallel-parser generator that works with highly ambiguous grammars.

Second, Rule 1 is right-recursive making it less efficient for a LR implementation. To solve this, the rule was simply changed to left-recursive:

1. $S \rightarrow SL$

This does not affect what the grammar generates; It still generates the same set and carries the same probability values [1].

Finally, the most important observation is that it includes a deadly rule, i.e., rule number 2; considering the empty string production would make the implementation more complicated and reduce performance[1]. Moreover, this rule is dangerous because its LHS, i.e., the nonterminal “S”, occurs in 6 out of 8 RHS’s, which makes these six rules, without the “S”, effectively part of the grammar, doubling the number of rules and doubling the size of the state space.

In order to safely modify the grammar, consider the following. ‘S’ is a list of ‘L’s. Since

we have stated that we do not need to generate the empty string, substituting epsilon in rule 1 would yield:

$$S \rightarrow SL$$

$$S \rightarrow L$$

$$L \rightarrow a \mid c \mid g \mid u$$

$$L \rightarrow au \mid ua \mid cg \mid gc \mid gu \mid ug$$

$$L \rightarrow aSu \mid uSa \mid cSg \mid gSc \mid gSu \mid uSg$$

But since L is now completely equivalent to S, S can replace L in the rest of the rules. After applying all the modifications mentioned above and recalculating the probabilities according to equivalence, the resulting grammar will be as follows:

Rule No.	Production	Probability
0	$S' \rightarrow S$	1
1	$S \rightarrow SS$	0.6829
2	$S \rightarrow a$	0.0432
3	$S \rightarrow c$	0.0216
4	$S \rightarrow g$	0.024
5	$S \rightarrow u$	0.0408
6	$S \rightarrow au$	0.0130176
7	$S \rightarrow ua$	0.0158976
8	$S \rightarrow cg$	0.021024
9	$S \rightarrow gc$	0.016704
10	$S \rightarrow gu$	0.004512
11	$S \rightarrow ug$	0.005952
12	$S \rightarrow aSu$	0.0168
13	$S \rightarrow uSa$	0.0216
14	$S \rightarrow cSg$	0.0336
15	$S \rightarrow gSc$	0.0264
16	$S \rightarrow gSu$	0.0048
17	$S \rightarrow uSg$	0.0072

Grammar 2: Modification of Grammar 1. Implemented in this project.

This was transformed from the original grammar in [2] and the probabilities were recalculated by writing every rule of the new grammar in terms of rules of the original grammar as logical expressions; detailed expressions for every transition can be found in Appendix A. This new grammar is equivalent to the original one except that it does not generate the empty string, which is practically not necessary as explained before.

3.2 Relevant information about CUDA

Here we present information about CUDA relevant to 3.3-3.6. Everything here is from [6] and [7].

CUDA organizes all parallel computations using abstractions, which are threads, blocks and grids. They represent a straightforward application of the single-program multiple-data parallel-processing paradigm. Here are definitions of the abstractions:

Thread: Simply, it is just an index for the execution of a kernel. The index of each thread can be used to access elements in an array. Threads are not used in the implementation of this project.

Block: A group of threads. Unfortunately, we are not sure of the details of executing threads within a block. They might be executing concurrently or serially at any given time. But calling the `_syncthreads()` function in the kernel would stop a thread until all others in the block finish. The same can be said for blocks running in parallel. They can be synchronized but only from the host by calling `CudaThreadSynchronize()`.

Grid: A group of blocks, can only be one-dimensional or two-dimensional.

Regarding the distribution of computation of these components:

Grid → GPU: A single GPU handles a single grid.

Block → Core: A GPU chip contains multiple cores, each of which can handle one or more blocks in a grid. The opposite obviously is not the case, i.e., a single block would never be handled by multiple cores.

Thread → SP: Each Core further divides into stream processors (SPs), each of which handles one or more threads in a block.

There is also a hierarchy of memory:

Global memory: It consists of a number of SDRAM chips on the graphics card, so that all blocks and all threads can access it.

Texture cache: This memory is available inside all cores; it is like a cache; and it can copy data from the global memory. But threads running in the block assigned to a certain core can only read and not write to this memory.

Constant cache: This is also a read-only memory inside each core. It is limited and must be hard-coded in the code.

Shared memory: This is an even smaller memory inside each core. It is a fast read/write memory for all threads in the block assigned to that core.

Registers: Registers are also inside cores and they are shared between threads.

Of course, as you go higher in the memory hierarchy, you get larger, slower storage; and as you get lower, you get smaller faster storage.

3.3 Tweaking the Algorithm for CUDA

This project's aim was to implement the same algorithm in [2] using CUDA. The problem is that parallel subprocesses, whether blocks or threads, in CUDA cannot communicate with each other, nor can they give commands to the CUDA device. So splitting up a subprocess into subprocesses upon detecting a conflict, as in step "create another thread" of the algorithm, is not possible with CUDA. All the thread management and memory management must be done on the CPU.

For these reasons, the implementation does not exactly correspond to the algorithm, yet it does not deviate from it. Such case is not uncommon; it is especially evident when algorithms are implemented in multiple platforms. This can be seen more clearly in the following abstract pseudo-code:

Function: `main()`

```
    Prepare Action[state][token] and Goto[state][non-terminal] tables
```

```
    Get #Actions; // (This is the Max # possible Actions per state per token, which
    is also equal to 1 + the Max # conflicts per state per token. It is extracted
    from the Action table: equals 3 for our grammar)
```

```
    Make initial Parse; // This is element 0 in an Array_of_Input_Parses
```

```
    Allocate Stack space
```

```
    Let N = 1;
```

```
    Let #Successful_Blocks = 1;
```

```
    while ( #Successful_Blocks > 0 )
```

```
        N = #Successful_Blocks * #Actions;
```

```
        Allocate space on the device for N Parses;
```

```
        call N instances of Device_Kernel();
```

```
        Get #Successful_Blocks from the last device call;
```

```
    end while
```

end `main()`

Function: `Device_Kernel()` // Called with multiple instances running in parallel

```
    Let x be a unique number referring to this instance;
```

```
    Let Input_Parse = Array_of_Input_Parses [ x/#Actions ];
```

```
    Given the state of Input_Parse, Get all possible actions from the Action table.
```

```
    Do Action #( x MOD #Actions )
```

```
    If the action is Shift or Reduce, copy the stack of Input_Parse into
    Output_Parse and update it accordingly, then set the status flag of
    Output_Parse to "successful".
```

```
    Else, if the action is Error or Accept, set the flag of Output_Parse to "failed"
    or "accepted" respectively.
```

end `Device_Kernel()`

Code 1: Pseudo-code of implementation

Since the `Device_Kernel()` function cannot split itself further, an alternative approach was taken: The host function allocates enough memory and calls the `Device_Kernel()` with multiple instances. Every three instances---“three” being the `#Actions`, which is also the branching factor m mentioned in Section 2.5 and calculated in appendix B---are given a parse as input and an empty space for the output parse. This is effectively splitting each parse into three.

Every output parse takes an action: either *Shift*, *Reduce*, *Error*, or *Accept*, and updates its status flag to either *successful*, *failed*, or *accepted*. Every successful parse needs a copy of its parent's stack to make its own changes on it according to the action; and this stack operation is what causes 90% of the overhead.

3.4 Memory Management Overhead

Naturally, before copying a parse's stack, a segment of the global stack space needs to be allocated for the output parse's stack. Keep in mind that multiple processes would want to allocate stack segments for themselves simultaneously. So, if this operation was not controlled by atomic locks, perhaps one segment might be allocated to two or more processes. This means that all parses that are successful, which basically are the ones that matter, need to go through the process of stack allocation in a sequential manner.

This sequential allocation procedure consumes more than 90% of the execution time of the `Device_Kernel()` function. The only alternative is to throw away the idea of dynamic memory allocation altogether, which leaves us with two options:

1. We pre-allocate all the memory that is going to be needed in the worst case; that amount of memory is close to $O(2.3^n)$ bytes as shown in 2.5. Such approach is not at all feasible, as it would need 8 Gigabytes of memory for an input size of 20 tokens.
2. Instead of providing every parse with its own stack space, We treat all parses working in parallel, at a single `Device_Kernel()` call, as a whole, i.e., for every iteration in `main()`, we do the following:
 - Before calling `Device_Kernel()`, We allocate enough space on the device for all parses' stacks, regardless of whether they will succeed or fail.
 - After the call, we deallocate the chunk of memory that was, in the previous iteration, allocated to what are now the stacks of `Input_Parses`.
 - Then we assign the next iteration's `Input_Parses`' stack chunk to what are now the `Output_Parses`' stacks

The second option would've been the best except that CUDA has been showing unexpected behavior whenever multiple Cuda Blocks try to write in different places of a space allocated by one `CudaMalloc()`. Therefore, if implemented, it had to be controlled by atomic locks, which means it had to be sequential.

Throughout the project, in order to make up for this low-performing sequential part of the code, I had been optimizing memory management operations through the following guidelines:

- using C structs instead of objects,
- performing low-level byte-by-byte copying,
- limiting the data types used to consume minimal size of memory,
- and most importantly, doing all data manipulation from the device on device memory, thus by, totally avoiding passing data between host and device except for a limited number of device pointers and primitive-type variables.

Note that the `main()` function which resides in the host and runs on the CPU, does the

device memory allocation; and it does it every iteration. This resembles the necessity imposed by the CUDA Architecture for the host (CPU) to interfere for managing memory. The Cuda Architecture imposes a segregation between device and host in that sense. That is why it was important, in order to gain better performance, to resist the suggested paradigm, i.e., perform all memory management on the host, and instead, do it all on the device.

3.5 Issues and Difficulties with CUDA

As 3.4 concluded, the CUDA Architecture imposes a segregation between device and host. This architecture represents a side of CUDA's poor memory management: the library is poor, in that it provides no functions that run from the device; and the implementation is poor for the evident and frequent resource hogs and memory leaks experienced when running a medium number of instances of the kernel; the numbers below illustrate.

According to [7], even if your hardware has only four cores, for example, you can still call the kernel with 10,000 parallel blocks, since the device will do all the management for you. However, my experience with CUDA says otherwise: there definitely is a limit to how many blocks you can instantiate, after which messy things start to happen; and that limit is related to the following factors:

- The number of CUDA cores on the device. For a device with 16 cores, the limit was around 1500 blocks; and for 336 cores, the limit was around 6000 blocks.
- The size of memory on the device, given that the memory allocated by the host and by the variables initialized in the kernel function never exceeded the size of global memory.
- Whether the kernel is called with a 1D, or a 2D grid of blocks; the limit increases respectively. In other words, the limit seems to depend on maximum number of blocks in any one dimension.

The reason is that CUDA's auto-management of blocks and threads is done using a local stack. And when too many blocks, with respect to available cores, are initiated, too many local stacks are activated. This might lead to corruption of the local stacks which might be the reason for the unexpected results described above.

So, there is a limit to the number of blocks that can be seamlessly managed. What makes it worse is that no option for micro-managing parallel blocks and threads is provided. This is specially bad because it shows ambiguity in NVIDIA's approach. Is CUDA a high-level or low-level programming environment?!

The boast that CUDA automatically manages the distribution of virtual blocks and threads to physical CUDA cores gives the impression that CUDA is for high-level programming while it lacks the most important high-level features. This will drive a programmer to low-level programming to recreate these important high-level features. For example, CUDA lacks one of the most important data structures in both basic and featured libraries: The Stack. The easiest way to create the stack structure in CUDA is to use a library that implements a dynamic array or Vector for CUDA, and then create your own push and pop functions. The low-level way is to use C-native arrays and pointers to implement the stack functions.

Therefore, CUDA does not provide full functionality neither for high-level

programmers nor for low-level programmers.

Another issue is debugging. Debugging is hard with CUDA because you simply cannot print messages from the device. You can only examine the output after all instances of the `Device_Kernel()` have finished execution. So, it is like a black box. To solve this issue, the programmer has to account for every single run-time error or logical error that could occur and check that every value used is in the expected range; if an error is detected, the kernel exits with a specific code for each case.

3.6 Source Code Features

For the reasons explained in 3.4 and 3.5, I decided to take matters by my own hand and create my own device-functions to over-bare the deficiency in CUDA's library:

- A Stack structure that can be manipulated from the device by the device. Naturally it includes device-functions: `Push()`, `Pop()`, and `Peek()`
- A Dynamic Array used for storing parse trees as they grow.
- A Buffer used for printing debugging messages.

All of these run from the device on data residing in device memory. Also, they are totally independent, i.e., they can be used for any other project simply by using `#include`. This means they represent a framework for future CUDA development.

4. Results

4.1 Raw Data

All CUDA experiments were conducted on a PC with 8 GB RAM, Intel Xeon X5680 3.33GHz processor running Ubuntu 11.04 with one Nvidia GeForce 460GTX with the following specs:

- 336 CUDA cores.
- 1GB RAM on board.
- Memory Interface: 256 bit
- Bus: PCI Express x16 Gen1
- Software: Nvidia developer driver version 280.13, CUDA Toolkit 3.2

Input String	Input Size	CPU Running Time (ms)	GPU Running Time (ms)	# Parse Trees	Max Stacks Needed	#
au	2	235.35	0.46	2	4	
acu	3	236.71	0.73	3	8	
acug	4	238.77	1.15	9	22	
acugg	5	241.70	2.95	27	71	
acugga	6	251.41	17.29	77	233	
acuggau	7	281.98	188.10	361	817	
acuggacu	8	372.20	1788.6	1009	2530	
acuggaug	8	409.80	3102.8	1346	3374	
acuggacuu	9	706.28	23,444	3526	9052	
acuggacua	9	781.87	28,029	3909	9949	
acuggauga	9	777.66	42,634	4524	12,318	
acuggacuaa	10	1211.3	402,581	13696	36,220	

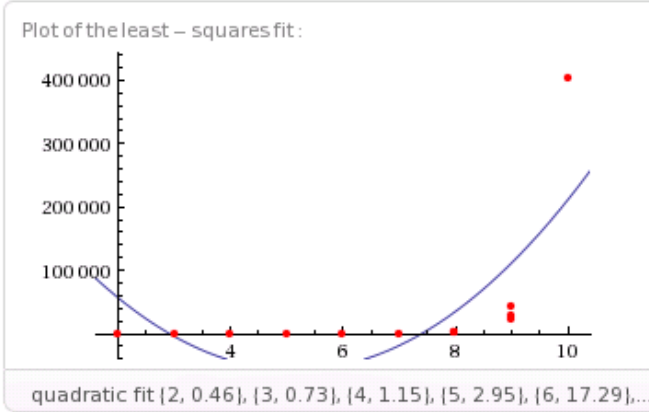
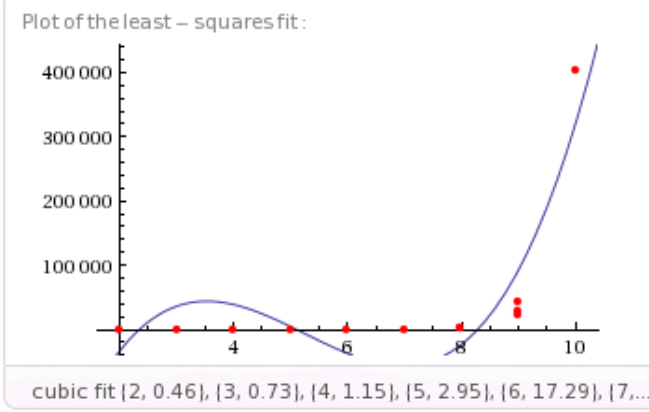
Table 1: Results of running the code on Nvidia GTX 460

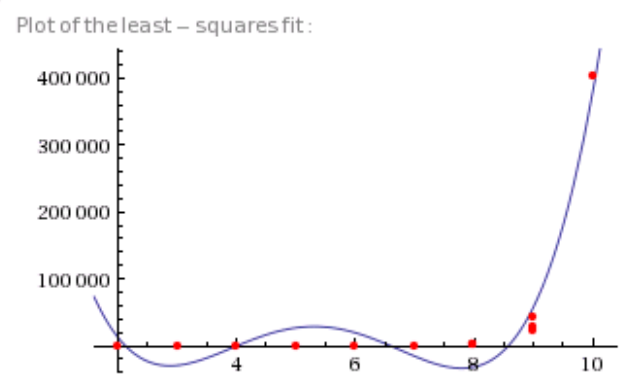
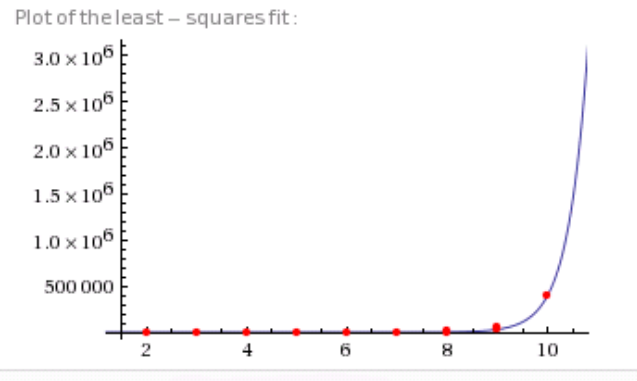
4.2 Data Analysis

Using Wolfram Alpha to fit the data of GPU time vs. Input Size to functions yielded the following:

Input to Wolfram Alpha: *typeOfFitting* fit {2, 0.46}, {3, 0.73}, {4, 1.15}, {5, 2.95}, {6, 17.29}, {7, 188.1}, {8, 1788.6}, {8, 3102.8}, {9, 23444}, {9, 28029}, {9, 42634}, {10, 402581}

typeOfFitting can be any of : quadratic, cubic, exponential.

Type of Fitting	Least-squares Fit	Plot
quadratic	$11475.5 n^2 - 118351. n + 247481.$	 <p>quadratic fit {2, 0.46}, {3, 0.73}, {4, 1.15}, {5, 2.95}, {6, 17.29},...</p>
cubic	$4967.66 n^3 - 78149.4 n^2 + 365930. n - 492497.$	 <p>cubic fit {2, 0.46}, {3, 0.73}, {4, 1.15}, {5, 2.95}, {6, 17.29}, {7, ...}</p>

Polynomial of degree 4	$1655.51 x^4 - 35264.6 x^3 + 261693. x^2 - 787820. x + 797002.$	 <p>Plot of the least – squares fit:</p> <p>polynomial of degree 4 fit {2, 0.46}, {3, 0.73}, {4, 1.15}, {5, 2</p>
exponential	$3.32159 \times 10^{-6} e^{(2.55207 n)}$	 <p>Plot of the least – squares fit:</p> <p>exponential fit { 2 , 0.46 }</p>

As the derivation in 2.5 predicts, there is a sharp performance drop after the input exceeds a capacity c which, by equation 3, in this case equals 4. Therefore an input of size $n = 6$, for example, would take as much time as an input of size $(2n - 2c) = 4$ running completely sequentially; similarly, an input of size 100 would take as much time running in parallel as would an input of size 192 running sequentially. Ultimately, the parallel programming aspect is not utilized at all since it does not perform much better than the sequential program. On top of that, there is the overhead of managing parse stacks and the overhead that comes from the CUDA device itself managing blocks and their local stacks. This multiprocessing overhead coming from CUDA cannot be determined because of the ambiguity of how CUDA manages cores.

5. Conclusion

5.1 Cuda is not the best choice

The difficulties posed by CUDA's architecture, described in 3.4-3.5, and CUDA's ambiguity about the management of cores has been constantly slowing development of this project and driving attention towards low-level problems. CUDA needs a fully featured more stable library that would keep the programmer focused on the main problem. Perhaps with future CUDA toolkit versions, and with libraries such as, Thrust, CaCuda, and Hydrazine, developing with CUDA would be more efficient rather than time consuming.

Moreover, the type of problems CUDA was built for and can very efficiently handle consists of massive computations, minimal data manipulation and memory operations, and zero inter-process communication. And given these factors, *parsing* is the total opposite of where CUDA shines.

In fact, any LR-based parser is not suitable for CUDA because it needs a stack. Here, it is even worse because the stack has to be dynamic.

5.2 GLR approach is not suitable

As explained in 2.2, the algorithm is based on GLR parsing. But GLR is usually used with techniques that reduce its time complexity from exponential to polynomial. Such techniques, splitting and merging, must be discarded to complete the objective of our algorithm, i.e., to find all possible parse trees. But if no other techniques, suitable to our goal, can replace them, then the implementation will end up running in exponential time.

The algorithm has offered a solution not deep enough as to completely parallelize the problem nor traditional enough as to use sequential approximation.

5.3 Future Work on Algorithm

The algorithm has to introduce at least one new dimension of parallelization In order to completely utilize any parallel programming environment. The most obvious sequential procedure in the current algorithm is traversing the algorithm from left to right. If this can be done in parallel, then this tedious problem would definitely be solved in polynomial time. This is not impossible. Consider the problem of Fibonacci or prefix-sum. These have been done in parallel. In fact the latter was brilliantly parallelized and implemented in [4].

To parallelize our problem, let us forget parsing for a moment and focus on the goal of why [2] used parsing in the first place. The goal is predicting RNA secondary structures guided by finding the pairs a-u, c-g, and g-u. Suppose we have the following string

“ACCCUCUC”. As soon as “A” is read from the left, all “U”s should be found in parallel that would form an A-U pair; and so the problem would be divided into “A{CCC}U” and “A{CCCUC}U” with the substring in braces to be divided next. When no more division is possible, because no pairs are found, we can deterministically parse small substrings completely in parallel. This divide-and-conquer approach is essential for finding new solutions and hidden dimensions of parallelization. This idea can be generalized to apply to other similar grammars.

From where this project stands, it is still premature to try and transform this idea into an algorithm. More research should be done on this idea before transforming it into an algorithm.

5.4 Future Work on Implementation

The source code has been carefully implemented as to be a standalone framework for future CUDA development. More details, comments, code clean-up, and documentation can be added to make this portion of the code an independent efficient library.

References

1. Aho, Sethi, Ullman, “*Compilers: Principles, Techniques, and Tools*”, Addison-Wesley, 1986. ISBN 0-201-10088-6
2. M.S. Al-Mulhem, “Multithreaded parsing for predicting RNA secondary structures”, *Int. J. Bioinformatics Research and Applications*, Vol. 6, No. 6, 2010
3. Bison Documentation, <<http://www.delorie.com/gnu/docs/bison/>>
4. Harris M., “Parallel Prefix Sum (Scan) with CUDA”, NVIDIA Corporation 2007
5. Leermakers, R., L. Augusteijn and F.E.J. Kruseman Aretz, “A functional LR parser”, *Theoretical Computer Science* 104 (1992) 313-323.
6. NVIDIA Corporation. “NVIDIA CUDA Programming Guide”. 2007
7. J. Sanders, E. Kandrot, “CUDA by example : an introduction to general-purpose GPU programming”, Addison-Wesley, 2011. ISBN 0-13-138768-5

Appendix A: Transition of Grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow SS \\ S \rightarrow a &\equiv (S \rightarrow LS) \wedge (L \rightarrow a) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow c &\equiv (S \rightarrow LS) \wedge (L \rightarrow c) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow g &\equiv (S \rightarrow LS) \wedge (L \rightarrow g) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow u &\equiv (S \rightarrow LS) \wedge (L \rightarrow u) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow au &\equiv ((S \rightarrow LS) \wedge (L \rightarrow aSu) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow a) \wedge (S \rightarrow LS) \wedge (L \rightarrow u) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow ua &\equiv ((S \rightarrow LS) \wedge (L \rightarrow uSa) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow u) \wedge (S \rightarrow LS) \wedge (L \rightarrow a) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow cg &\equiv ((S \rightarrow LS) \wedge (L \rightarrow cSg) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow c) \wedge (S \rightarrow LS) \wedge (L \rightarrow g) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow gc &\equiv ((S \rightarrow LS) \wedge (L \rightarrow gSc) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow g) \wedge (S \rightarrow LS) \wedge (L \rightarrow c) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow gu &\equiv ((S \rightarrow LS) \wedge (L \rightarrow gSu) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow g) \wedge (S \rightarrow LS) \wedge (L \rightarrow u) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow ug &\equiv ((S \rightarrow LS) \wedge (L \rightarrow uSg) \wedge (S \rightarrow \varepsilon)) \vee ((S \rightarrow LS) \wedge (L \rightarrow u) \wedge (S \rightarrow LS) \wedge (L \rightarrow g) \wedge (S \rightarrow \varepsilon)) \\ S \rightarrow aSu &\equiv (S \rightarrow LS) \wedge (L \rightarrow aSu) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow uSa &\equiv (S \rightarrow LS) \wedge (L \rightarrow uSa) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow cSg &\equiv (S \rightarrow LS) \wedge (L \rightarrow cSg) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow gSc &\equiv (S \rightarrow LS) \wedge (L \rightarrow gSc) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow gSu &\equiv (S \rightarrow LS) \wedge (L \rightarrow gSu) \wedge (S \rightarrow \varepsilon) \\ S \rightarrow uSg &\equiv (S \rightarrow LS) \wedge (L \rightarrow uSg) \wedge (S \rightarrow \varepsilon) \end{aligned}$$

The rules or productions on the left-hand side represent the suggested grammar. They are expressed in terms of productions from the original grammar in the form of clauses related by AND (\wedge) and OR (\vee) operators.

The method used to calculate the probability of left-hand side productions is:

- 1- Substitute each clause on the RHS with its probability from the original grammar
- 2- Substitute each AND with a multiplication and each OR with an addition.
- 3- Evaluate the expression.

The results are shown in *Grammar 2*.

Appendix B: Action Table and the calculation of the branching factor m

Following is the action table copied verbatim from the source code

```
table[NSTATES][NTOKENS][MAX_ACTIONS] =
{
//
/* state 0 */ { { 0 , 0 , 0 }, { 1 , 0 , 0 }, { 2 , 0 , 0 }, { 3 , 0 , 0 }, { 4 , 0 , 0
},
/* state 1 */ { {-3 , 0 , 0 }, { 1 ,-3 , 0 }, { 2 ,-3 , 0 }, { 3 ,-3 , 0 }, { 7 ,-3 , 0
},
/* state 2 */ { {-4 , 0 , 0 }, { 1 ,-4 , 0 }, { 2 ,-4 , 0 }, { 9 ,-4 , 0 }, { 4 ,-4 , 0
},
/* state 3 */ { {-5 , 0 , 0 }, { 1 ,-5 , 0 }, {11 ,-5 , 0 }, { 3 ,-5 , 0 }, {12 ,-5 , 0
},
/* state 4 */ { {-6 , 0 , 0 }, {14 ,-6 , 0 }, { 2 ,-6 , 0 }, {15 ,-6 , 0 }, { 4 ,-6 , 0
},
/* state 5 */ { {127, 0 , 0 }, { 0 , 0 , 0 }, { 0 , 0 , 0 }, { 0 , 0 , 0 }, { 0 , 0 , 0
},
/* state 6 */ { {-1 , 0 , 0 }, { 1 , 0 , 0 }, { 2 , 0 , 0 }, { 3 , 0 , 0 }, { 4 , 0 , 0
},
/* state 7 */ { {-7 , 0 , 0 }, {14 ,-6 ,-7 }, { 2 ,-6 ,-7 }, {15 ,-6 ,-7 }, { 4 ,-6 ,-7
},
/* state 8 */ { { 0 , 0 , 0 }, { 1 , 0 , 0 }, { 2 , 0 , 0 }, { 3 , 0 , 0 }, {18 , 0 , 0
},
/* state 9 */ { {-9 , 0 , 0 }, { 1 ,-5 ,-9 }, {11 ,-5 ,-9 }, { 3 ,-5 ,-9 }, {12 ,-5 ,-9
},
/* state 10*/ { { 0 , 0 , 0 }, { 1 , 0 , 0 }, { 2 , 0 , 0 }, {19 , 0 , 0 }, { 4 , 0 , 0
},
/* state 11*/ { {-10, 0 , 0 }, { 1 ,-4 ,-10}, { 2 ,-4 ,-10}, { 9 ,-4 ,-10}, { 4 ,-4
,-10} },
/* state 12*/ { {-11, 0 , 0 }, {14 ,-6 ,-11}, { 2 ,-6 ,-11}, {15 ,-6 ,-11}, { 4 ,-6
,-11} },
/* state 13*/ { { 0 , 0 , 0 }, { 1 , 0 , 0 }, {20 , 0 , 0 }, { 3 , 0 , 0 }, {21 , 0 , 0
},
/* state 14*/ { {-8 , 0 , 0 }, { 1 ,-3 ,-8 }, { 2 ,-3 ,-8 }, { 3 ,-3 ,-8 }, { 7 ,-3 ,-8
},
/* state 15*/ { {-12, 0 , 0 }, { 1 ,-5 ,-12}, {11 ,-5 ,-12}, { 3 ,-5 ,-12}, {12 ,-5
,-12} },
/* state 16*/ { { 0 , 0 , 0 }, {22 , 0 , 0 }, { 2 , 0 , 0 }, {23 , 0 , 0 }, { 4 , 0 , 0
},
/* state 17*/ { {-2 , 0 , 0 }, { 1 ,-2 , 0 }, { 2 ,-2 , 0 }, { 3 ,-2 , 0 }, { 4 ,-2 , 0
},
/* state 18*/ { {-13, 0 , 0 }, {14 ,-6 ,-13}, { 2 ,-6 ,-13}, {15 ,-6 ,-13}, { 4 ,-6
,-13} },
/* state 19*/ { {-15, 0 , 0 }, { 1 ,-5 ,-15}, {11 ,-5 ,-15}, { 3 ,-5 ,-15}, {12 ,-5
,-15} },
/* state 20*/ { {-16, 0 , 0 }, { 1 ,-4 ,-16}, { 2 ,-4 ,-16}, { 9 ,-4 ,-16}, { 4 ,-4
,-16} },
/* state 21*/ { {-17, 0 , 0 }, {14 ,-6 ,-17}, { 2 ,-6 ,-17}, {15 ,-6 ,-17}, { 4 ,-6
,-17} },
/* state 22*/ { {-14, 0 , 0 }, { 1 ,-3 ,-14}, { 2 ,-3 ,-14}, { 3 ,-3 ,-14}, { 7 ,-3
,-14} },
/* state 23*/ { {-18, 0 , 0 }, { 1 ,-5 ,-18}, {11 ,-5 ,-18}, { 3 ,-5 ,-18}, {12 ,-5
,-18} },
};
```

A positive number is a shift action; a negative number is a reduce action. The number of actions at every cell will be included in calculating the average of m , the branching

factor explained in 2.5, except for one column and two rows: the column of the token <EOF>, end of file, because it's a special token that occurs only once in the input and is obviously negligible; the row of the initial state number 0; and the row of the terminal state number 5, because these two states, at least the former, are only visited once during the lifetime of a parse.

The following table has been extracted from the action table. It carries the number of possible reductions per state per token. It is also available in the source code:

```
{
//
/* state 0 */ { eof, 0, 0, 0, 0 },
/* state 1 */ { 1, 1, 1, 1, 1 },
/* state 2 */ { 1, 1, 1, 1, 1 },
/* state 3 */ { 1, 1, 1, 1, 1 },
/* state 4 */ { 1, 1, 1, 1, 1 },
/* state 5 */ { 0, 0, 0, 0, 0 },
/* state 6 */ { 1, 0, 0, 0, 0 },
/* state 7 */ { 1, 2, 2, 2, 2 },
/* state 8 */ { 0, 0, 0, 0, 0 },
/* state 9 */ { 1, 2, 2, 2, 2 },
/* state 10 */ { 0, 0, 0, 0, 0 },
/* state 11 */ { 1, 2, 2, 2, 2 },
/* state 12 */ { 1, 2, 2, 2, 2 },
/* state 13 */ { 0, 0, 0, 0, 0 },
/* state 14 */ { 1, 2, 2, 2, 2 },
/* state 15 */ { 1, 2, 2, 2, 2 },
/* state 16 */ { 0, 0, 0, 0, 0 },
/* state 17 */ { 1, 1, 1, 1, 1 },
/* state 18 */ { 1, 2, 2, 2, 2 },
/* state 19 */ { 1, 2, 2, 2, 2 },
/* state 20 */ { 1, 2, 2, 2, 2 },
/* state 21 */ { 1, 2, 2, 2, 2 },
/* state 22 */ { 1, 2, 2, 2, 2 },
/* state 23 */ { 1, 2, 2, 2, 2 },
};
```

Again, state 0 and 5, and token <EOF> must be excluded from the calculation. Additionally, since the Shift operation is always valid, then the branching factor = 1+ number of possible reductions.

The calculation of m , the average branching factor, has been done using a spreadsheet software.

$$m = 1 + \text{Average of possible reductions} = 1 + 1.318 = \mathbf{2.318}$$