Parallel CSV Reader using NVidia Cuda Logic

Introduction

The CSV Reader is a module of logic that takes advantage of the parallelism of running in NVIDIA Cuda on NVIDIA GPU boards. This logic has been implemented and deployed inside of several larger software programs, some of which are currently in production.

CSV stands for Comma Separated Values, and it is a simple specification for transferring (importing/exporting) data records in simple text files, called CSV files. The data in these text files are in row/column format. Each line of text represents one data record. Inside each record a separator character, or delimiter, is placed between each field, or column. For example, 6 fields would have 5 delimiters between adjacent fields. The comma is the standard delimiter character. However, the CSV Reader supports other delimiters. The record delimiter is normally Carriage Return followed by Linefeed (thus one line equals one record).

The purpose of the CSV Reader is really quite simple. It moves CSV data, as read in from a file, into a set of fixed width arrays, one for each column. The memory allocation for each of the arrays is based on the number of lines in the CSV file: in other words the total number of records, times the byte width of each array.

While in some ways this type of operation may seem trivial, it has a number of powerful advantages, which can lead to orders of magnitude increase in performance.

A CSV file is typically a very densely packed way of storing data. The values are all strings (text), but UTF-8 formatting is typically used to handle both simple ASCII and multi-byte characters very efficiently. Arrays are generally not compact, especially for strings, where array width must accommodate the *maximum* byte width expected and supported. Even with numeric values, arrays may not offer much advantage where the text versions of numbers are typically small (like “23”, “2.1”, “5”, etc.)

The advantage of arrays, however, is that GPU can very easily do many kinds of processing using them. What’s more, once this processing is complete:

1. The processing or calculations that use the arrays in GPU memory are extremely fast and take place in parallel.
2. The arrays may stay in GPU memory for an extended period supporting multiple types of processing, such as queries.
3. When results must be returned to the CPU, they are often very small in size.
4. The overall effect is that a great deal of processing and calculation can occur with a minimum of expensive CPU-GPU memory transfers, thus ensuring maximum performance.

The following figure shows the typical flow around the CSV Reader:

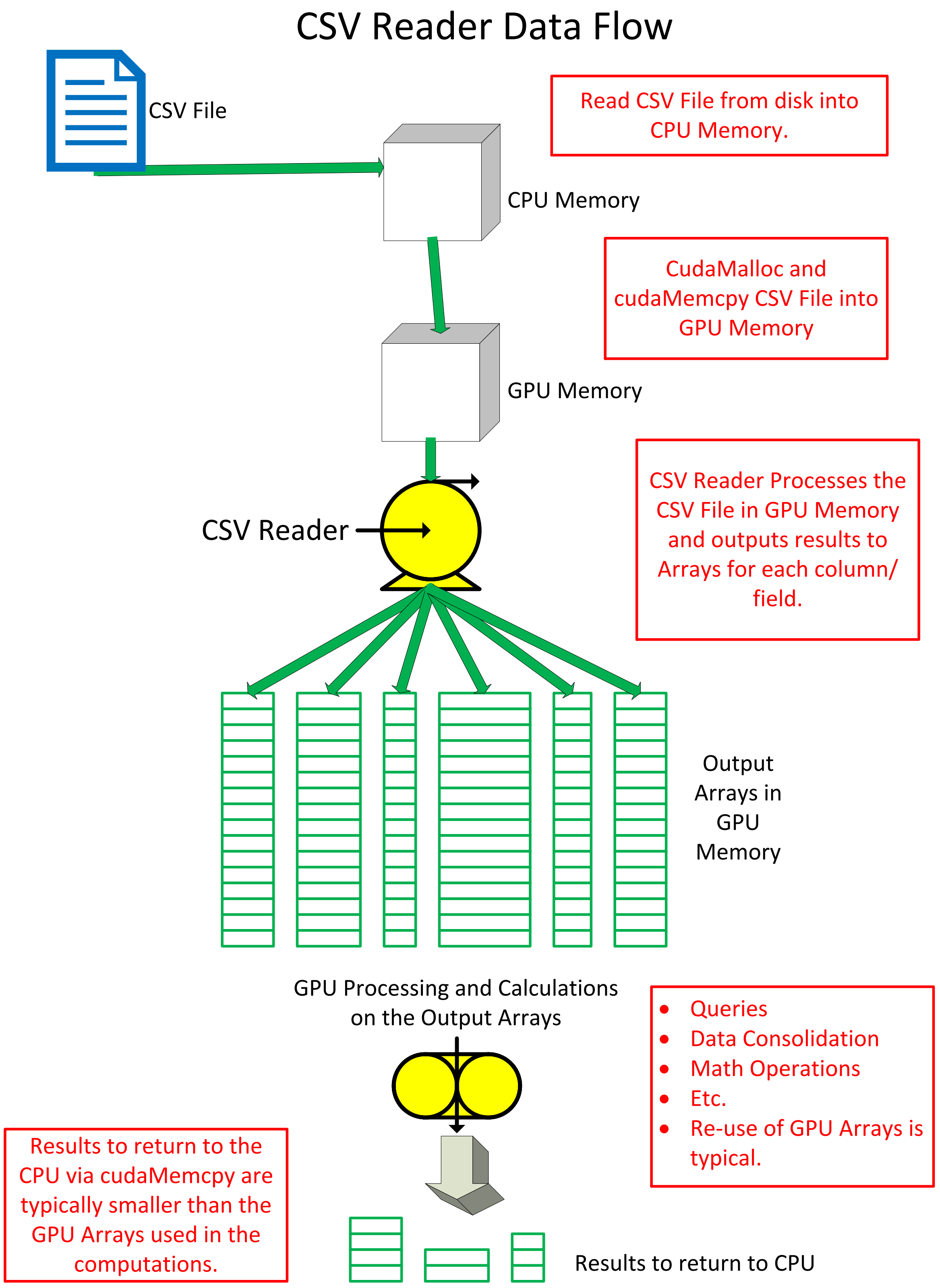

CSV Reader Data Flow
CSV File
Read CSV File from disk into CPU Memory.
CPU Memory
CudaMalloc and cudaMemcpy CSV File into GPU Memory
GPU Memory
CSV Reader
CSV Reader Processes the CSV File in GPU Memory and outputs results to Arrays for each column/ field.
Output Arrays in GPU Memory
GPU Processing and Calculations on the Output Arrays
• Queries
• Data Consolidation
• Math Operations
• Etc.
• Re-use of GPU Arrays is typical.
Results to return to the CPU via cudaMemcpy are typically smaller than the GPU Arrays used in the computations.
Results to return to CPU

Figure 1

CSV Reader Challenges

There are several challenges involved in the way the CSV Reader works that we highlight here.

Conversion from Unlimited Text Length to Fixed Array Sizes

A delimited text file does not limit the length of the text that comprises a field value. If these values must go into a Fixed Array, it is necessary to restrict the length of text for each field or column. For most purposes this is not a problem. Typical text fields like FirstName, LastName, Address1, SKU, UPC, etc. are normally limited in length in the systems in which they are stored. Even floating point numbers are typically limited to a certain decimal precision when stored as text. (Imagine trying to store the value of PI in text in full precision.)

The CSV Reader comes into its own when processing very large numbers of records, say 50 million or so. Memory availability and consumption, especially in the GPU, is always a concern. If Fixed Arrays must be very wide to accommodate long text fields, this can severely limit the number of records that can be processed. The CSV Reader might not be particularly useful in contexts where long text fields are involved.

Workarounds for Long Text Fields and Large Memory Requirements

There are two workarounds built into the CSV Reader to ameliorate these issues:

1. The CSV Reader is designed to *chunk* its input. If a very long CSV file must be processed, the CSV Reader can read a portion of the file at a time, breaking on complete record boundaries. Once this is done there are two possibilities. The GPU may return partial results to the CPU, which will in turn consolidate the results. Otherwise, the GPU may fully process each portion of the file, and discard the full output arrays and other temporary memory, but retain the smaller consolidated results in GPU memory, and then consolidate those results after all portions of the file have been processed, and then finally return those fully consolidated results to the CPU.

2. The CSV Reader can selectively convert Columns or Fields into GPU Fixed Arrays. Often a standard CSV file will have fields that are not needed for the specific processing required. Also, long text fields may not be needed for the specific processing or computations the GPU will do. The CSV Reader can be programmed to convert only specific fields to GPU Fixed Arrays. All other fields will be skipped over in the CSV Reader. What’s more, the lengths of skipped over fields do not need to be limited.

Character Set Considerations

The CSV Reader is designed to process UTF-8 Characters in its input. This is the standard in the internet world. The CSV Reader handles UTF-8 characters ranging from 1 byte to 3 bytes in length, thus covering the original Unicode specification. Its output into Fixed Arrays is also in UTF-8. UTF-8 is also consistent with ASCII, and will look the same for text that only uses the ASCII character set.

Character Size Limits versus Array Byte Widths

For any given column or field the CSV Reader permits defining both the Byte Width of the Array and the Character Limit. For example, let's say you have a field you want to limit to 20 characters. Allowing for up to 3 bytes per character in UTF-8, you may define a Fixed Array of 60 bytes width. Obviously you could put 60 ASCII characters in this Array, but this would be too many characters. This would be picked up as an Error in the CSV Reader. Also, if the field has more than 60 bytes this is also picked up as an Error in the CSV Reader.

Note: If your CSV file input only uses the ASCII range of characters, your byte width and character limit are the same: 20 characters maximum requires 20 bytes.

The CSV Reader will skip any Record where there is a character or byte length error in any Field. These errors will be collected and reported back to the CPU.

Number of Columns per Record

A well-formed CSV file should have a consistent number of columns or fields per record. For example, if that number is 6, 5 or 7 would be an undefined or error situation. The CSV Reader checks number of columns for each record. An incorrect number of columns is flagged as an Error.

The CSV Reader will skip any Record where there is a column count error in any Field. These errors will be collected and reported back to the CPU.

Special Characters in Comma-Delimited Format

The standard for comma-delimited format is what is found in Microsoft Excel. The comma alone works fine as long as there are no commas in text for any field.

For example:

```
babylon 5, 2777.00,Hello Kitty,Thomas
```

yields four text fields separated by the commas.

If you want a field with a comma in it, you must put quotes around the field:

babylon 5, 2777.00,"Hello, Kitty",Thomas

still yields 4 fields, but now *Hello, Kitty* contains a comma that is not a delimiter.

Thus now not only the comma, but the quotation marks are special characters.

What if you want to put a quote inside a field as part of the field? You must double the quote:

babylon 5, 2777.00,"Hello ""Our"" Kitty",Thomas

yields a field which is *Hello "Our" Kitty.*

These cases pose a challenge for the CSV Reader. It must filter out what we call *non-printing* characters. These are characters that do not go into the output Arrays. Since the arrays are fixed width and the individual values in an array do not require any apparatus to separate them from each other or from other columns, the non-printing characters are not needed. Moreover, they are not wanted, as they are not part of the original values. If you load a CSV file into Excel, you do not see the non-printing characters in the cells! (You do see them in a text editor, however.) The delimiter characters and the newline characters are obviously non-printing. However, CSV Reader must carefully filter out the others. Not all commas are delimiters, and not all quotes belong in the output text.

Another important factor here is that non-printing characters do not count toward either character or byte counts, when checking for too many characters or bytes.

CSV Reader Setup

Based on the material presented so far, we can define the Setup Data for the CSV Reader. The table below defines an Importer job.

c	Customer Import
Description	Customers with Status
CSV File	C:\customers.CSV
Chunk Size (bytes)	50000000
Record Boundary	CR LF
Delimiter Character (ASCII)	,
Total Number of Columns	16
Number of Header Rows	1
Schedule CRON	30 * * * * ?

Figure 2

The Chunk Size is set based on the GPU memory available, the number and size of the Output Arrays, and other factors. The chunk size should be set conservatively to allow for the very high memory requirements of GPU processing. Chunking is not feasible for all applications. Some require all the data in memory at the same time.

The Record Boundary is either CR LF (carriage return followed by linefeed) or LF (linefeed). The Delimiter Character separates the fields in a record. It is one ASCII character. The default is Comma. The total number of columns is the number in the imported file, *not* the number of Output Arrays. The CSV Reader needs to know the number of columns even if it discards a number of them.

The Number of Header Rows is the number of rows or full records in the file which serve as a header (typically 0 or 1). The CSV Reader will skip the header row(s) and not try to put the contents into Output Arrays.

The CSV Reader is also capable of reading Field Names from the columns in the Header Record. Since the main operation of the CSV Reader is simply to populate Output Arrays, the field names may not be important internally. However, this capability may prove useful in a future feature.

The Schedule CRON defines when and how often the processing will occur.

The table below defines the Columns processed by the CSV Reader. This is an expandable table, depending on the number of used columns.

Column #	**Output**		**Array Width**	**Maximum #**
(0-based)	**Column #**	**Field Name**	**Bytes**	**Characters**
0	0	FirstName	88	29
1	1	LastName	88	29
2	2	Address1	120	40
3	3	Address2	120	40
4	4	City	80	26
5	5	State	60	20
6	6	CountryCode	4	4
7	7	PostCode	8	8
8	8	Phone	16	16
12	9	Status	8	8
14	10	Active	2	2

Figure 3

Here you note that out of the 16 total columns we are only outputting 11 Arrays. We don't use input columns 9, 10, 11, 13, or 15. The Output Column Number should not have gaps. However, it does allow the Output Arrays to be "positioned" differently. The Output Column Number and the Field Name are used internally to define the individual Output Arrays.

Maximum number of characters is based on many systems (such as SQL Server) that define fields in terms of character width, not byte width. The maximum ensures that the number of characters does not exceed that number, even if you could squeeze more characters into in a row in the array (by using simple ASCII for example).

The Array Width in Bytes should be a multiple of 4 to ensure good memory alignment. Failure to observe this could cause alignment errors in the GPU. The Maximum Number of Characters should be based on 2 factors:

- The Array Width in Bytes
- The UTF-8 Characters expected

If only ASCII is expected, the maximum number of characters can equal the byte width. If some 2-byte UTF-8 characters are expected (common in European languages), the maximum number of characters should be half the byte width. If some 3-byte UTF-8 characters are expected, the maximum should be one-third of the byte width, rounded down, as shown in Columns 0 – 5 above.

Use of the Data

A complete User Interface for setting up the CSV Reader would have the equivalent of the tables above, but would also set up how the Output Arrays are to be consumed. Typically, such applications take great advantage of the speed and parallelism of the GPU. Examples include queries, data consolidation, date reorganization, creating output for other systems, etc.

Parallel Pre-Processing of CSV Data

Before the CSV Reader core logic can process CSV byte data into Output Arrays, it must perform a number of steps to provide additional data arrays which will allow the work to continue in parallel. In this section we will outline the simple version of this pre-processing. The simple version is based on the following:

- Field delimiters only appear in field boundaries. No commas within quotes or double quotes to escape a quote as discussed above.
- All data fit within their defined output array widths. There are no overruns.
- All data are ASCII text characters, so we are always dealing with 1 byte per character.
- All records or rows have the correct number of fields or columns. No column count errors.

It is worth noting that in some applications the CSV input data complies very well with all these criteria, so what we will show in this section is a realistic baseline case.

If you were to read through a CSV file in a purely sequential manner, columns per record, printing versus non-printing characters, character counts, and byte counts do not pose any special challenges. You would simply keep track of various states (conditions, such as inside a quoted field) and counters (such as column count in a record) as you read through the file. Simple!

In the GPU which processes in parallel, things are a bit more challenging. However, we do have some techniques at our disposal.

The GPU runs kernels which process in parallel. There are different ways to structure how the parallel-running threads in a kernel map, or read and write, data. In all the pre-processing outlined below, each thread will map to 1 byte in the CSV Input Array. Now the kernels that create Scans or Headers as explained below, will typically write out data into an Array of 32-bit integers (4 times the size of the input). However, each thread is still mapped to one byte of input, even where it is writing 4 bytes at a time.

The CSV files is copied into the GPU as an array of bytes. These bytes are UTF-8 which can mean 1 to 3 bytes per text character. Initially, we can identify record boundaries, as these are Carriage Return/Linefeed pairs. (At the end of the file there may or may not be a final Carriage Return/Linefeed.) The first task is to locate the record boundaries and then what record number any given byte finds itself in.

For this purpose we use standard GPU techniques. These are memory intensive, but extremely fast. We allocate two arrays of 32-bit integers that are the same number of 32-bit integers as there are bytes in the CSV byte array. One array is the Header Array, and the other is the Prefix Sum (otherwise known as Scan) Array. First we run a GPU kernel, where each thread maps to a byte in the CSV byte array. If this happens to be a Line Feed, the kernel writes a 1 to the Header Array, otherwise a 0, at the corresponding position. As this is embarrassingly parallel, it is extremely fast. Next we run the Prefix Sum (in this case an Exclusive Scan) on the Headers. Now the Scan Array will have the 0-based record number that corresponds to every byte in the CSV Array.

What is a Prefix Sum or Scan? It is simply a running sum of values in an Array (typically integers). An Exclusive Scan starts at 0 and for subsequent Scan values adds the prior Array value to the current Scan value. (An Inclusive Scan is similar, but uses the current Array Value rather than the prior. It is essentially shifted left compared to an Exclusive Scan. Inclusive Scans are not used very often.)

While Prefix Sum is *not* embarrassingly parallel, there are some excellent GPU libraries with extremely fast Scan routines, on the order of at least 150 times faster than would be achieved on a very fast CPU!

Here is a simple non-parallel code snippet in C++ for an Exclusive Scan:

```
#define ARRAYLENGTH 24
int VALUE[ARRAYLENGTH] = { 2, 5, 0, 9, 6, 1, 8, 4, 7, 5, 1, 3, 5, 2, 1, 9, 2, 6, 7, 1, 3, 3, 1, 6 };

// exclusive scan
int XSCAN[ARRAYLENGTH + 1];
for (int idx = 0; idx < ARRAYLENGTH + 1; idx++)
```

```
{
        if (idx == 0)  XSCAN[idx] = 0;
        else  XSCAN[idx] = XSCAN[idx - 1] + VALUE[idx - 1];
}
```

Headers and Scans are used many times in setting up the CSV Reader, and are therefore very important.

The following figure illustrates the start of a simple 3-column CSV file.

- The first row is the array index for the next three rows (a byte index for CSV, and a 32-bit index for the subsequent rows).
- The second row is the array of bytes in the start of the CSV.
- The third row is the Linefeed Header array created by the GPU kernel. The Linefeeds separate the records in the CSV.
- The fourth row is the Exclusive Scan. The value is the 0-based record number that the CSV byte is in.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3

Figure 4

In the above figure each square represents an element of an Array, either a byte array or a 32-bit integer array. The figure below shows the first portions of the same arrays vertically instead of horizontally, with each square representing 1 byte. For the 4-byte array elements, the least significant byte is on the left side. This way of looking at the data is closer to what you would see in a memory dump.

- The Array Index for all arrays is the first (left) column.
- The CSV is the second column.
- The Linefeed Header array is the third column
- The Scan is the fourth column.

0	H	0	0	0	0	0	0	0	0
1	e	0	0	0	0	0	0	0	0
2	l	0	0	0	0	0	0	0	0
3	l	0	0	0	0	0	0	0	0
4	o	0	0	0	0	0	0	0	0
5	,	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0
7	2	0	0	0	0	0	0	0	0
8	.	0	0	0	0	0	0	0	0
9	2	0	0	0	0	0	0	0	0
10	,	0	0	0	0	0	0	0	0
11	W	0	0	0	0	0	0	0	0
12	o	0	0	0	0	0	0	0	0
13	r	0	0	0	0	0	0	0	0
14	l	0	0	0	0	0	0	0	0
15	d	0	0	0	0	0	0	0	0
16	CR	0	0	0	0	0	0	0	0
17	LF	1	0	0	0	0	0	0	0
18	M	0	0	0	0	1	0	0	0
19	a	0	0	0	0	1	0	0	0

Figure 5

Next, consider very simple CSV files, files with no commas or quotes inside fields. These are actually fairly typical, as often data are fairly consistent, such as zip codes, state and country abbreviations, or SKUs composed of letters, numbers, and hyphens. Also, a character which will never be found in a field value may be chosen as a delimiter, such as a vertical bar. Under these circumstances the comma, or other delimiter, can be used definitively as the column delimiter.

In this case our next task will be to identify the fields within the records. We already have Linefeed Headers. We will now run a kernel that populates a Header Array for commas, the field delimiters. Then we run a different variety of Prefix Sum or Scan – the Segmented Scan. The Segmented Scan works like a regular Scan except that the count is reset at various points, creating separate segments. In this case our Segment Boundaries are the Record Boundaries, namely the Linefeeds. The Scan is the Scan of the comma Headers within those Segment Boundaries.

The figure below shows the Segmented Exclusive Scan.

- The first 4 rows are the same as in the figure above.
- The fifth row is the Columns (Commas or Delimiters) Header Array.
- The sixth row is the Segmented Exclusive Scan, which resets to 0 after every Linefeed Header. The value is the 0-based column number within the record (row).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e	,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	0	0	0	0	1	1	1	2	2	2	2	2	0	0

Figure 6

Thus for every byte in the CSV Array we know which record it is in and which column it is in.

The Records and Columns Tables, and Segmented Scans

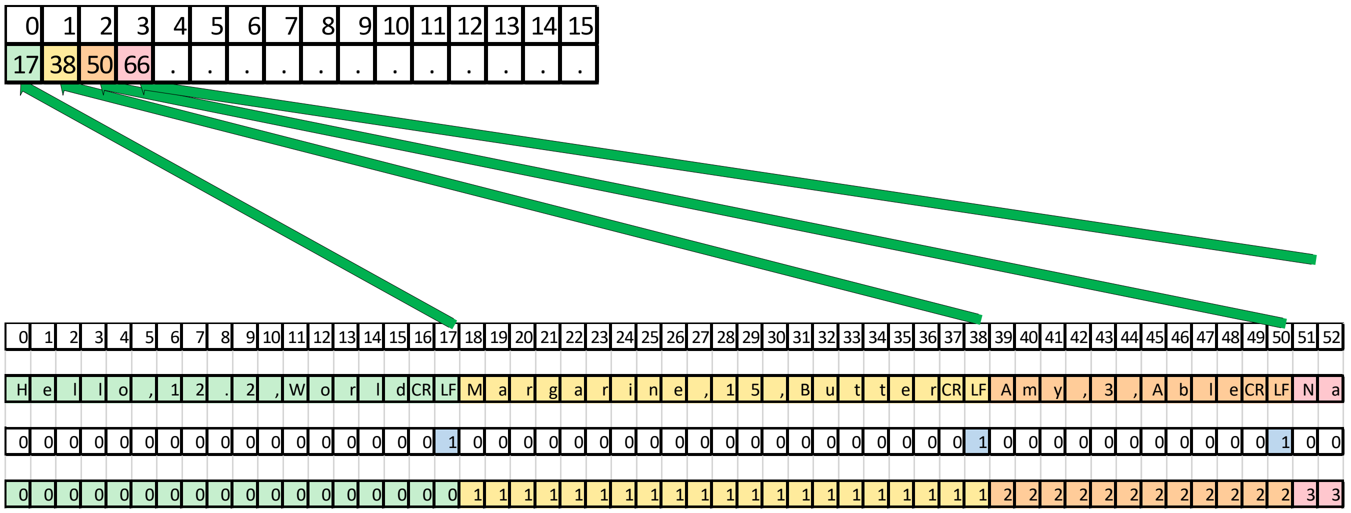
We have referred to and explained Segmented Scans. Segmented Scan is available is some of the standard GPU libraries we use. However, we create our own Segmented Scans. One of the reasons we do this is because some of the major Tables we need bring us one simple step away from having a Segmented Scan. Here we will outline three of the tables we need throughout the CSV Reader.

First we need a Records Table. This is an array which maps the start of each Record in the CSV. From this it is very easy to find where a record starts and ends by a simple array lookup into this table. Fortunately, based on our Scan of Record Headers we know where each record starts and ends.

To build the Records Table, we make use of another GPU operation that runs in a simple kernel, making use of a Header array and the Exclusive Scan of that array, namely Stream Compact, also known as Scatter. Stream Compact is an efficient parallel compression of result values, which appear across a very large array, into a small array of just the results. The results buffer can be allocated as a 32-bit array, the length of which is the final Scan Value plus the final Header value (0 or 1).

The Records Table is a Stream Compacted array. The Index of the array is the 0-based Record Number, and the Value in the Array is the Index of the Linefeed at the *end* of the record in the CSV Array. Thus this is actually a Linefeeds Table, as illustrated in the figure below.

A simple kernel (again, 1 thread per 1 byte of input) writes this table. If the current thread finds a Linefeed Header at its current position, it writes its index into the Records Table at the position (in the Records Table) of the Exclusive Scan value at the thread's index. For example, in the figure below, the thread at index 17 will see a Linefeed Header, and will map its index (17) into the Records Table at 0, which is the value of the Exclusive Scan at 17. The thread at index 38 sees a Linefeed Header at that position. It then writes its index, 38, into the Records Table. The position it writes in the Records Table equals the Scan value at its index of 38, namely 1.



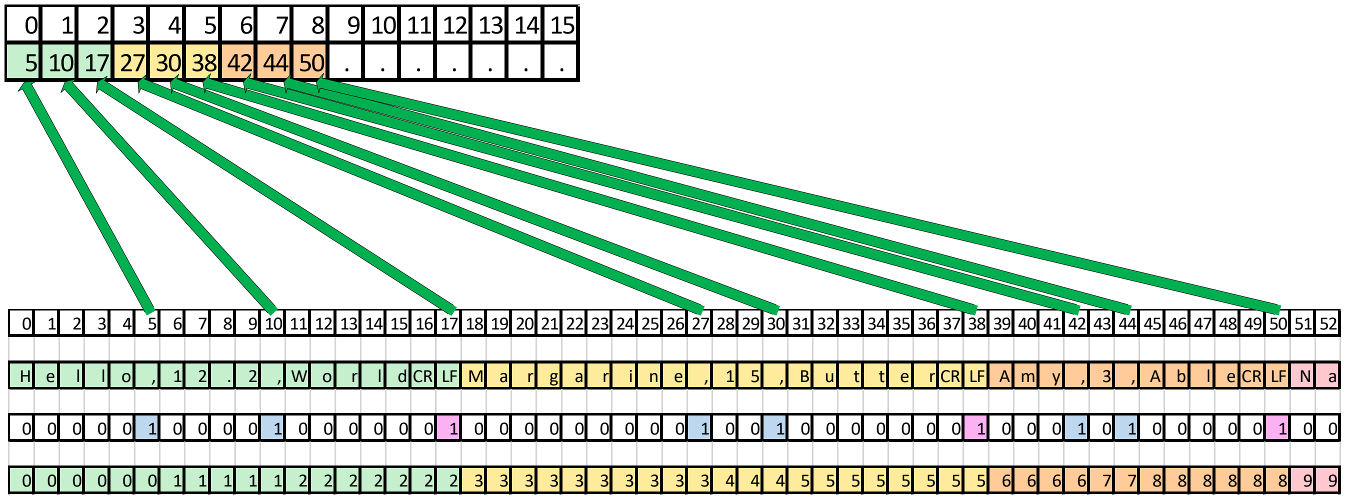
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
17	38	50	66	.	.	.	.	.	.	.	.	.	.	.	.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e	,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3

Figure 7

We also need a Columns Table. This will work like the Records Table except that it will be based on Column Headers and Linefeed Headers. It will use the Exclusive Scan and the Stream Compact. However, in this case the table will reflect the column delimiter positions themselves. The figure below illustrates the formation of the Columns Table.

- The third row are headers for both the column delimiters (commas), shown in blue, and the linefeeds, shown in pink.
- The fourth row is the Exclusive Scan which provides the index offsets into the Columns Table filled in by the Stream Compact, shown on top.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	10	17	27	30	38	42	44	50	.	.	.	.	.	.	.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e	,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	7	7	8	8	8	8	8	8	9	9

Figure 8

Note that there is a lot of flexibility when it comes to how Headers are built and how Stream Compacts are carried out. There may be slightly different formats, each of which are useful. For example, in the Records Table you might include Record 0 at 0, or you might start with Record 1, as we know that Record 0 will always start of CSV Position 0. Typically, we pick a way of doing it, and then make minor adjustments in the code for the format as needed.

In the Records Table above, you know that Record 0 *starts* at 0 and effectively ends at the linefeed, at 17. Record 1 starts at the position just *after* the value in position 0 of the Records Table, namely 18. Record 2 starts at the position just *after* the value in position 1 of the Records Table, namely 39

Similarly in the Columns Table when you are figuring out where the *content* of each column starts. The content of column 0 starts at 0, column 1 starts at 6, column 2 at 18, column 3 at 28, and so on.

We also create another reference table, the Records-To-Columns Table, which is used in the Core processing. This Table has the same as the Records Table. However, it maps the position in the Columns Table that corresponds to the Linefeed Headers. Thus if the CSV is 3-columns, the Columns Table is about 3 times larger than the Records Table. The Records-To-Columns Table would simply reference every third entry in the Columns Table. The Records-To-Columns Table would therefore seem to be redundant since it would always give a simple multiple. However, we need this information since we are allowing for catching and filtering out records with column-count errors, which means that you can't simply use a multiple.

The figure below illustrates the Records-To-Columns Table, with the Records Table on the bottom, the Columns Table in the middle, and the Records-To-Columns Table on top.

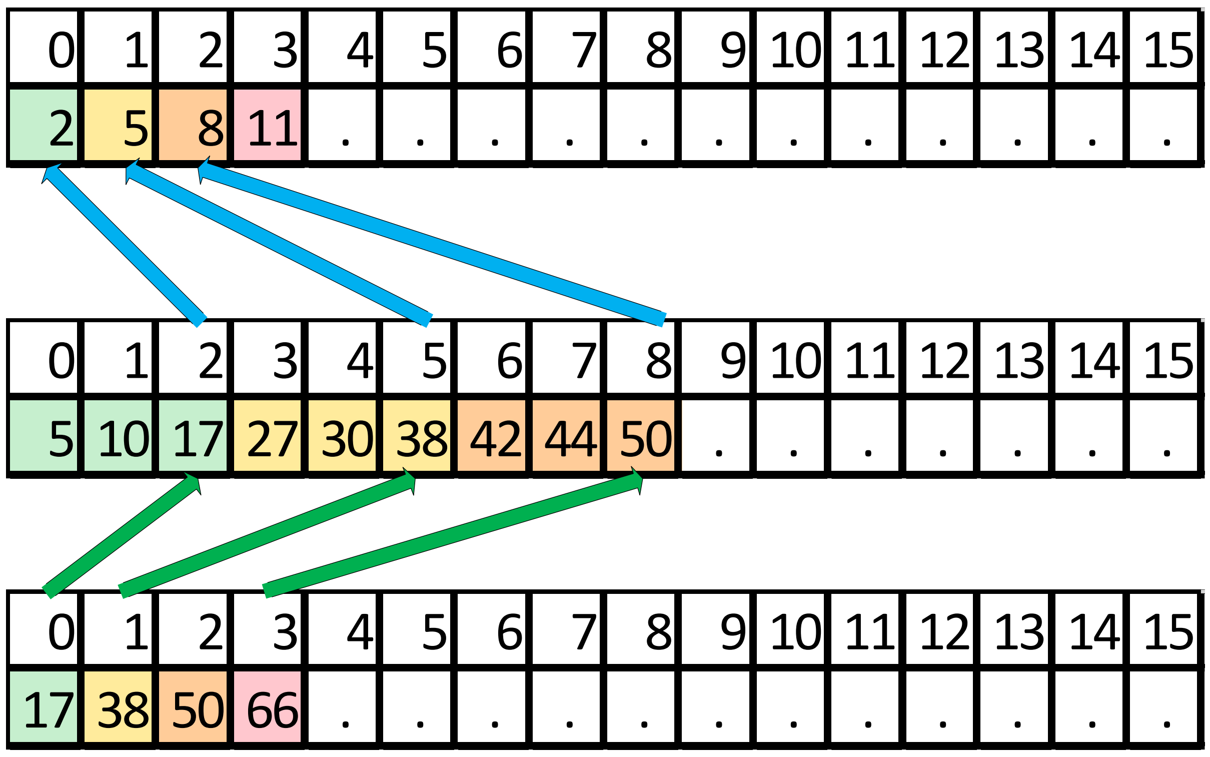


Figure 9

With the three tables above, we can see the simple step required to create a Segmented Scan of Column Headers within each record. The logic is processed in a simple kernel, operating on one CSV byte per thread in parallel. Each thread reads the Exclusive Scans of the Linefeeds (for records) and of the Column Delimiters. For the first thread, we know we are in the first column in the first record. For the rest, we know our record number and column number from the Scans. Based on the record number as the index, we then look up the current value in the Record-To-Columns Table. This tells us what the column number is at the *start* of the current record. We then subtract this value from the current thread's column number. The result is written into an output array – which *is* the Segmented Scan of the Columns within the Record.

When we refer to other Segmented Scans used in the CSV Reader, the principle follows the same logic outlined here for the Columns Segmented Scans. For example, to add a Segmented Scan of UTF-8 Characters within Columns, we create the UTF-8 Headers, the Exclusive Scan of those, and a Columns-To-UTF8Characters Table. From these we easily derive the Segmented Scan in the same way as we did for the Columns.

Close to Parallel Output (But No Cigar)

Next we add one more Array, which is simply a Header Array for Printing Bytes as defined above. This does not necessarily reflect the order in which operations are done. However, it begins to give a picture of the information we have available in the CSV Reader. This data is generated by a kernel (or set of kernels) that figures which bytes should be copied to output. In our sample above this is very simple, all bytes except commas, carriage returns, or linefeeds.

In the figure below, we have added two rows.

- Row 7 shows Printing Headers.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e	,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	0	0	0	0	1	1	2	2	2	2	2	2	0	0
1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	0	1	0	1	1	1	1	0	0	1	1

Figure 10

There is one more step required before we are ready to write the Output Arrays. We need to get the byte position of each character in the input. (Again, we assume here one byte per character, so byte and character positions are the same.) To do this we do another Segmented Exclusive Scan. This is a Scan of the Printing Headers, and the Segment reset on both Column Headers and Record Headers (thus at every new column).

- Row 8 shows the new Segmented Scan. The values give the byte or character position of each input character.
- Row 9 is informational only, and does not represent another array that is generated. It simply maps the input to show the actual bytes that will be written.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
H	e	l	l	o	,	1	2	.	2	,	W	o	r	l	d	CR	LF	M	a	r	g	a	r	i	n	e
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	0	1	2	3	4	0	1	2	3	4	5	5	0	1	2	3	4	5	6	7	8
H	e	l	l	o		1	2	.	2		W	o	r	l	d			M	a	r	g	a	r	i	n	e

27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
,	1	5	,	B	u	t	t	e	r	CR	LF	A	m	y	,	3	,	A	b	l	e	CR	LF	N	a
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	3	3
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	1	1	1	2	2	2	2	2	2	2	2	0	0	0	0	1	1	2	2	2	2	2	2	0	0
0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	0	1	0	1	1	1	1	0	0	1	1
9	0	1	2	0	1	2	3	4	5	6	6	0	1	2	3	0	1	0	1	2	3	4	4	0	1
	1	5		B	u	t	t	e	r			A	m	y		3		A	b	l	e			N	a

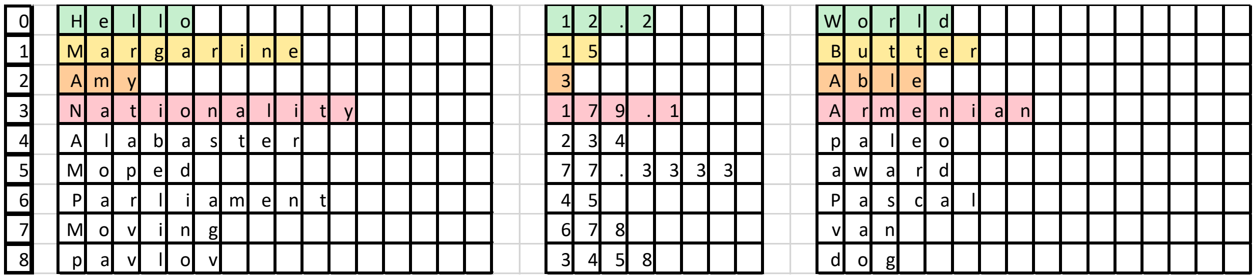
Figure 11

We are now at a point to see the parallelism we have made possible. All the Scan and Header Arrays show are 32-integers versus the single byte of input. However, thinking about 1 thread per input byte executing in parallel, we see how a final kernel could create the Output Arrays. Each thread could check the Printing Headers to see if its byte should be written. If so, it checks the Scans and Segmented Scans to identify exactly where this byte goes in the Output:

- Which Record.
- Which Column (or Output Array in this case).
- Which byte or character position within the Record within the Output Array.

At this point the operation is embarrassingly parallel.

Below we show the first few records in our sample CSV in the Output Arrays. We have selected 16-byte arrays for the first and third columns, and an 8-byte array for the second column. On the left is the record index for all Output Arrays.



0	H	e	l	l	o											
1	M	a	r	g	a	r	i	n	e							
2	A	m	y													
3	N	a	t	i	o	n	a	l	i	t	y					
4	A	l	a	b	a	s	t	e	r							
5	M	o	p	e	d											
6	P	a	r	l	i	a	m	e	n	t						
7	M	o	v	i	n	g										
8	p	a	v	l	o	v										

1	2	.	2				
1	5						
3							
1	7	9	.	1			
2	3	4					
7	7	.	3	3	3	3	
4	5						
6	7	8					
3	4	5	8				

W	o	r	l	d											
B	u	t	t	e	r										
A	b	l	e												
A	r	m	e	n	i	a	n								
p	a	l	e	o											
a	w	a	r	d											
P	a	s	c	a	l										
v	a	n													
d	o	g													

Figure 12

It is important to see how we are in a position to write Output Arrays at this point. However, the CSV Reader does not operate this way. There are two more basic operations involved.

- We have omitted the pre-processing of complex comma-delimited data, with quoted columns, commas within columns, and escaped quotes within columns. This occurs in a separate pre-processor which is skipped if we are not dealing with such CSV input.
- The actual writing out of CSV data to Output Arrays is handled in a Core Processing kernel, which is optimized for memory alignment and performance. If we simply wrote data as we were suggesting above, one thread per one byte, we would be using memory inefficiently.

Nevertheless, this simple perspective shown above has value to illustrate techniques used throughout the CSV Reader that make possible a maximum of parallelism. We will see some of these techniques used in the next section covering complex CSV pre-processing.

Parallel Pre-Processing of Complex CSV Data

The pre-processing of CSV where there is a delimiter character *only* used as a delimiter is relatively simple, as we have outlined above. For example, the CSV Reader has been used with a vertical bar delimiter that never appears elsewhere in the data. However, normal comma-delimited rules present additional challenges for parallel processing.

The CSV Reader handles comma-delimited input as a special case, and does additional pre-processing as we will describe in this section – which is not needed for a vertical bar delimiter. The result of this pre-processing will be to rewrite the CSV buffer so that it can be processed more simply, as outlined in the previous section.

Although we are working in parallel, we cannot look at an individual byte in the CSV and know exactly what is going on. Is a quote the start or end of a quoted field, or a quote inside a field, or an escape character? Thus look-backs and look-aheads are necessary. Fortunately, there are practical limits to how far these look-aheads and look-backs have to go with any normal content.

CSV Reader has a special section of code to handle comma-delimited formatting. This is an either/or between the simple delimiter formatting (as outlined above) and the more complex comma-delimited formatting.

Here are the basic steps to handle complex comma-delimited formatting. These are all based on one thread per byte of CSV or per Array Element for Header or Scan Arrays.

The First Kernel: MarkCommas

This kernel first checks for five conditions: at a comma, at a carriage return, at a linefeed, at the first CSV byte, at the last CSV byte. The thread exits if none of these are found. If this is a comma, the thread sets Headers for Commas and Column Headers (tentative). If this is a linefeed, the thread sets Headers for Columns, Linefeeds, and Quote Boundaries.

Next this kernel does a look forward and/or backward to count quotation marks. At a comma, a linefeed, or at the start of the CSV, the thread looks forward counting quotes. At a comma, a carriage return, or at the end of the CSV, thread looks backwards counting quotes. These looks always start one byte away at a comma (since we know it is not a quote), and at the current byte at the start or end of the CSV (since the current byte could be a quote).

If either the forward or backward looks count an odd number of quotes, the quote closest to the comma (in that direction) or the current byte at the start or end of the CSV, is a quote boundary – meaning it is a quote used to surround a field value. In these cases the QuoteBoundary Header is set at the boundary quote position. (At a comma you could possibly set a QuoteBoundary Header on both the byte before and the byte after the current byte.) At the same positions, the Printing Characters Headers are set to 1 (though these will be reversed later since boundary quotes are non-printing).

The Second Kernel: DoubleQuotes

The second kernel isolates first printing quotes. These are not boundary header quotes. Within a field a first printing quote is one that should go to the Output Array. Since these quotes must be escaped by being doubled up, we say that the first quote (or all odd position quotes) are printing, and the second quote (or all even positioned quotes) are non-printing. Care is taken to make sure that each thread will handling an exclusive set of bytes, so that parallel threads cannot step on each other.

The thread exits if the current CSV byte is not a quote or if the Printing Header is already set to 1 by the MarkCommas kernel – which would be the case for a Boundary Quote. Then the thread looks at the prior byte. If this is a quote with Printing Header set to 0, the thread determines that this is not the first non-boundary quote and it exits. If the thread thus determines that it is the first of a non-boundary quote, then it has work to do. It looks forward, and sets a Second Quotes Header at even position it finds that has a quote. (Since normally you don't have double quotation marks in actual text, this Second Quotes Header will probably only be set for one position up. Later these Second Quotes will be merged with Non-Printing, but a separate array is used here to prevent the parallel threads from stepping on each other.

The Third Kernel: Merge2ndQuotesAndNonprinting

This is a simple kernel. It merges Printing Headers and Second Quotes. Either Header set to 1 means non-printing. Then it rewrites Printing Headers, reversing the prior meaning, so that printing is 1 and non-printing is 0. Since this is done on per CSV byte basis (no forward or backward looks), the threads do not step on each other. The result is that Boundary Quotes and Second Quote is escaped quotes, are marked as non-printing.

Exclusive Scan of Quote Boundary Headers

The next step is an Exclusive Scan of the Quote Boundary Headers. This scan is based on the fact that Quote Boundaries occur in pairs. The Exclusive Scan value will be odd inside a Quote Boundary pair and even outside one.

The Fourth Kernel: FixColumnHeaderCommas

This is a simple kernel to fix up the Column Headers. If the thread is at a CSV comma, it checks the Quote Boundary Scan at that position. If the Scan value is odd, the Column Header is reset to 0 from 1. That particular comma will then simply be part of the output text, and not a column boundary.

Exclusive Scan of Printing Characters

This Scan of the Printing Characters will pave the way for a rewrite of the CSV buffer. Here, the only "non-printing" characters are the Quote Boundaries and the Second Quotes in escaped quotes. At this point, commas that are column boundaries and carriage returns and linefeeds are still treated as "printing." These will be filtered out later, as they do not actually get "printed" into the output arrays.

The Fifth Kernel: BufferPrinting_StreamCompact

This kernel uses the above Scan to rewrite the CSV buffer array to remove the non-printing characters described above. At the same time, it stream compacts the Record Headers array and the Column

Headers array based on the same Scan. If there are Quote Boundaries and Escaped Quotes, each of these three arrays will get shorter.

Parallel Processing of Column Count and Character Count Checks

The CSV Reader has a certain degree of fault-tolerance. It will skip records which have the wrong number of columns, or where any column destined for an Output Array contains too many characters or bytes. When this happens the CSV Reader will report these errors back to the CPU, with a report as to which records have errors and were skipped.

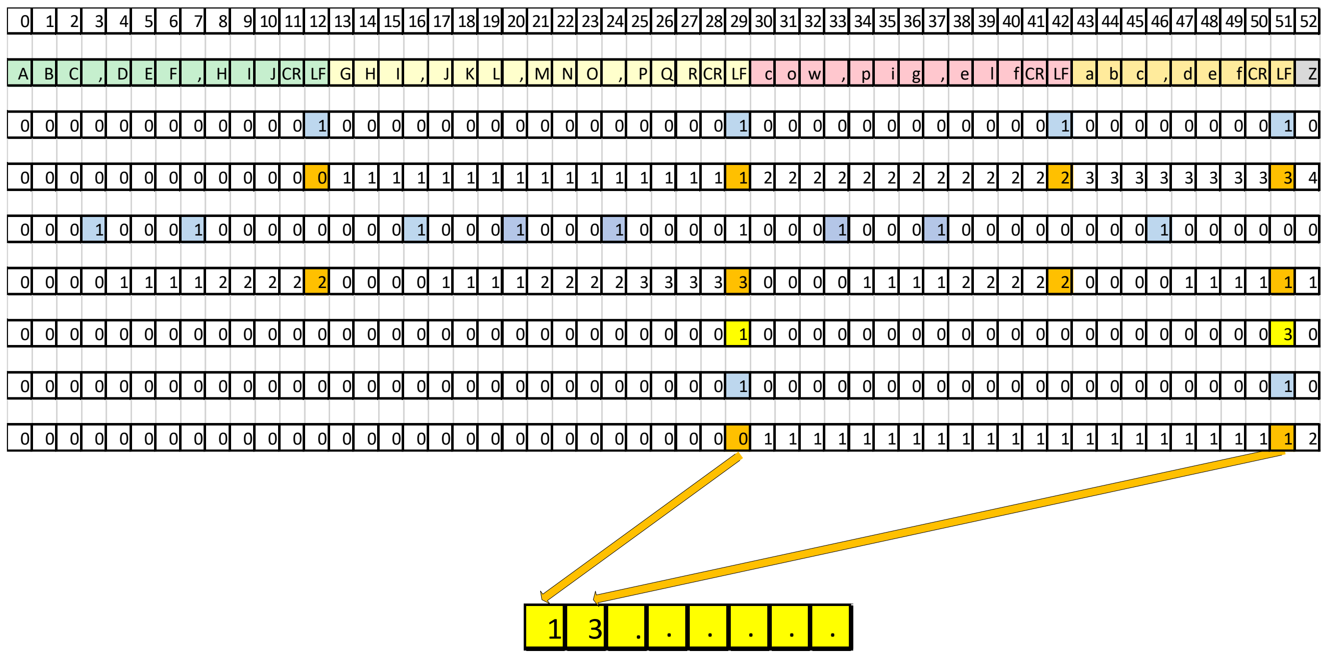
Here, also, our kernels and Scans are all based on one thread per byte of CSV or per Array Element (32-bit) for Header or Scan Arrays – and the indices shown reflect that.

Identifying Column Count Errors

The Segmented Scan of Column Headers, along with the Linefeed (Record) Headers and the Linefeed (Record) Scan, provide all the information we need for picking up and marking erroneous records.

The figure below gives a simple example. Here we are expecting three columns in each record. Anything else is an error.

- The first row is, as usual, the array index, and the second is the start of a CSV Byte Array.
- The third row is the Linefeed (Record) Headers, and the fourth is the Exclusive Scan of those headers. At each Linefeed Header position in the Scan you have the 0-based Record Number, highlighted in orange.
- The fifth row is the Column Header Array, and the sixth is the Segmented Scan of that that array.
- In the sixth row, note that the Linefeed Header Positions, highlighted in orange, show the 0-based number of columns in that record. You can see that the second record has four columns and the fourth record has two – both column count errors.
- The seventh row now copies the 0-based Record Number from the fourth row. At the same time an Error Header is flagged, shown in the blue highlight in the eighth row.
- The ninth row does an Exclusive Scan of the eighth row.
- We now do a Stream Compact. Wherever there is an Error Header (eighth row), we copy the Record Number (seventh row) into an Error Table (shown below in all yellow), at the index offset of the Scan (ninth row). This is now a compact Error Table that can be returned to the user.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
A	B	C	,	D	E	F	,	H	I	J	CR	LF	G	H	I	,	J	K	L	,	M	N	O	,	P	Q
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	1	1	1	2	2	2	2	2	0	0	0	0	1	1	1	1	2	2	2	2	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
R	CR	LF	c	o	w	,	p	i	g	,	e	l	f	CR	LF	a	b	c	,	d	e	f	CR	LF	Z
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
3	3	3	0	0	0	0	1	1	1	1	2	2	2	2	2	0	0	0	0	1	1	1	1	1	1
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2

Figure 13

Meanwhile the erroneous records are skipped. Here is what the Output Arrays might look like (assuming three at 8-byte width each).

0	A	B	C						D	E	F						H	I	J					
1	c	o	w						p	i	g						e	l	f					
2	Z	a	n	d	e	r			P	e	r	c	h				f	i	s	h				

Figure 14

Character Count Error Checking

Character counts checks start differently from column count checks, but they finish the same way.

Kernel: BuildCharsHeadersOnly

This kernel actually exists in two different versions depending on context. Regardless of which, the purpose is set to 1 (or clear, set to 0) the UTF8charHeaders Array. This kernel makes sure the current

CSV byte is not on a Column Header, a Record Header (linefeed), or a Carriage Return. If not, the current CSV byte is examined. If the bit format of the byte (from most to least significant bit) is 0xxx xxxx, or else is *not* 10xx xxxx, this is the first byte of a UTF-8 character which can be one, two, or three bytes long. If so, the header is set. (Note: It is the user's responsibility to set the appropriate storage multiplier (1, 2, or 3) for the UTF-8 characters that will be used in each field. If in doubt, the value should be set to 3.) This kernel is the first step in being able to count the characters.

Note that *all* the bytes of any UTF-8 character are "printing": they are all sent to the respective Output Array in sequence.

Segmented Scan of UTF-8 Character Headers

A Segmented Scan of the UTF-8 Character Headers, which resets at every Column or Record Header, will provide a quick means of identifying a Too Many Characters situation.

Suppose we are reading a 2-column CSV file destined for two Output Arrays. The first Array is limited to 5 characters, and we expecting to handle up to 3-byte UTF-8. The byte width of the array is 16 bytes, as we round up to the nearest 4-byte width. The second Array is limited to 8 characters, and we expecting to handle up to 2-byte UTF-8. The byte width is 16 bytes.

The figure below shows how this might work:

- In the first record, we see five 3-byte characters in the first column and eight 2-byte characters in the second column. In the second record, we see six 1-byte characters (over the limit) and three 2-byte characters.
- The third row shows the Column Headers, and the fourth row shows the Segmented Scan of the Column Headers, breaking on each record. The orange bytes indicate where the 0-based column numbers are to be identified (so we see columns 0 and 1 twice).
- The fifth row shows the UTF-8 Character Headers as described above.
- The sixth row shows where the Character Counts are found for each column. These are identified by the orange bytes and are 1-based. The positions where these are found are where there are delimiter commas and linefeeds.
- Note that the first column in the second record shows 6 characters. Even though these are ASCII and the 6 bytes would fit easily into the Array, the 5-character limit is exceeded. This record is an error and would be skipped.
- The next figure shows the two 16-byte Output Arrays with the first record copied in.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
色			は			匂			へ			ど			,	ε		λ		λ		η		ν		ι		κ		ή		CR	LF	A	B	C	D	E	F	,	Π		ά		ν		CR	LF	色			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1	1	1	0	1	0	1	0	1	0	0	0	1	0	0	0
0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	8	0	1	2	3	4	5	6	0	1	1	2	2	3	3	3	0	1	1	1

Figure 15

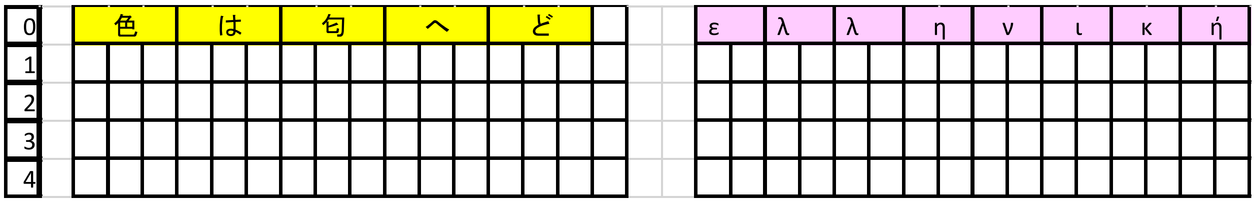


Figure 16

Character Count Error Output

In the figures above we see the mapping of character count errors to Record Numbers – in much the same way as we saw above for column count errors. This could be used to provide detailed information on output:

- Which record contains the error.
- Which column contains the error.
- What is the character count of the erroneous column.
- With some column-centric logic we could report on multiple character count errors in different columns in the same record.

In fact, to make things a bit simpler, we simply merge the column count and character count errors, and only report on which records contain one or more errors, without differentiating.

A Note on Memory

The reader may note how memory intensive the CSV Reader is. Going from a single byte array to several 4-byte arrays, multiplies the size of the original CSV many times over.

It is important to note that, as long as GPU memory is available, these operations involve allocating and freeing GPU memory. The processing of the memory is all done in parallel, and there is no DMA copying of memory inside the GPU context, or between CPU and GPU except to load the CSV initially and to return results as processing completes. Thus these operations are incredibly fast.

The CSV Reader is structured to use GPU memory conscientiously. As the processing proceeds, arrays are freed as soon as they are no longer needed.

As the GPU evolves there is more on-board memory. Unified Memory is providing faster access to other pools of memory, and oversubscription of GPU memory is now possible with performance benefits.

The CSV Reader Core Processing

The Core Processing follows a different model than what we were led to above, where each thread writes one byte into an Output Array.

While that model works, it is not efficient in how it uses GPU processing or memory. Unlike the pre-processing above, the Core is based on 4 bytes per thread. Inside a GPU threads are grouped into Warps, and within a warp there is a great deal of thread-to-thread transparency. There are 32 threads per warp, so each warp handles 128 bytes at a time. This corresponds to the minimum fetch block size GPU global memory aligned on even 128-byte boundaries.

Memory alignment makes a difference on the CPU. However, it is especially important on the GPU. GPU memory should always be read or stored on even boundaries for the size of the object. For example, if you are accessing 32-bit integers, they should always be configured on even 4-byte boundaries relative to their allocations. Thus you would never want to access a 32-bit integer at byte offsets 1, 2, or 3, but only at 0 or 4 (or multiples of 4).

We always use minimally 4-byte multiples for memory offsets in the Core. We also take advantage of casts of what are byte arrays to 32-bit or even 64-bit integers to move 4 or 8 bytes in single instructions.

The CSV Data Array is a sequence of bytes comprised of rows (records) and columns (fields). However, the sizes of these are in no way fixed. A field might be empty or it might have 1000 bytes in it.

In the Core, each GPU Warp handles one full record or row, regardless of the length of that row. The CSV byte array starts on an even 128-byte boundary, so the first record will start on an even 128-byte boundary. However, subsequent records are most likely not to do so.

In the Core, each Warp will calculate its Warp Index, which is the Index of Thread 0 in the Warp divided by 32. The Warp Index will correspond to the 0-based Record Number which it will process.

Core CSV Buffer Read Alignment

In the Core, each Warp will grab the information for its corresponding Record from the Records Table based on its Warp Index, which is the Index of Thread 0 in the Warp divided by 32. It will also grab the next value in the Records Table to find where its record ends.

Each Warp handling one record will load one or more 128-byte chunks of memory which encompass one entire record. The first chunk will contain the first byte of the record. If the last byte of the record is not included in the first chunk, more chunks will be loaded as needed until the last byte has been read. This may seem inefficient as often time more bytes will be read than needed, and adjacent records will often read the same 128 bytes. However, the fact that 128 bytes are always read from memory, and caching may make the duplicate reads available in cache, means that this is a very efficient system.

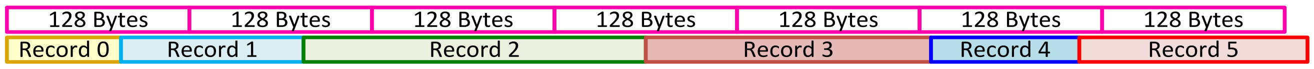


Figure 17

Figure 10 above shows how 128-byte chunks may map unevenly to records, whose lengths can vary. In this situation, we read the first chunk for Record 0, the first and second for Record 1, the second through fourth for Record 2, and so on.

While the Warp Index should normally match the Record Number, there is an adjustment made for records containing errors. Thus if Record 3 has an error and is skipped, Warp 3 would work with Record 4.

Shared Memory and Output Alignment

The Core does a number of look-backs and look-aheads within the record. Since these are within the warp, they take advantage of inter-thread communication techniques within a warp. When the record is longer than one chunk, the next chunk is immediately loaded into Shared Memory. This allows look-aheads to cross the even 128 boundaries. Shared Memory is very fast.

The first chunk is read into local GPU variables. If there is a second chunk it is read into Shared Memory. When the first chunk is finished, the second chunk in Shared Memory is copied into locals. At that point if there is a third chunk, it is read into Share Memory. This process continues until all chunks for the record are processed. The only overlapping access of global memory occurs when there are multiple records that are each partially contained in the same 128-byte range. This is mitigated by the fact that the 128-byte accesses are optimum, and the memory cache will help to some extent.

There is one Output Array for each column in the CSV. To ensure memory alignment, we require that the widths of all output arrays be defined in multiples of 4 or even (preferably) 8. If you used array widths such as 5 or 6, this would be problematic for the Core's logic.

Constant Memory

The Core starts by copying several “metadata” values into GPU Constant Memory. In short, constant memory “broadcasts” to all threads in a warp at once. Thus it is very good for storing standard more-or-less constant values that all threads use. In contrast, using global memory for these common values can lead to per-thread replays of memory reads, which can seriously reduce performance. Constant memory is not for large data arrays, where memory accesses will range across those arrays.

The following values used by all threads are copied into Constant Memory:

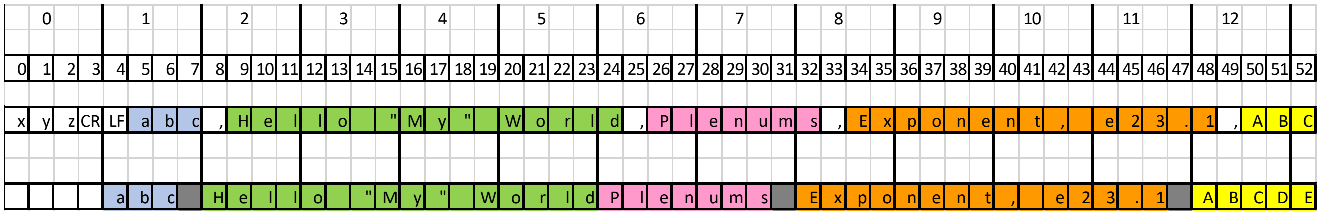
- Field Character Widths
- Field Byte Widths
- The Pointers to each of the Output Arrays

These values are used by all warps for all records.

Core Overview

The primary work of the Core is to read in bytes that are not in an aligned structure. At this point we assume that all non-printing characters resulting from comma-delimited-specific features have been removed from the CSV buffer. Thus we are dealing only with actual “printing” bytes, and non-printing column delimiter characters, carriage returns, and linefeeds. (Note: In a comma-delimited situation, some commas may appear that are printing. However, they will not be designated as column boundaries.) The printing bytes of each column are “shuffled down” so that the column aligns with the start of a 4-byte boundary. As the process completes, threads that have 4 bytes ready to write do write out their 4 bytes to an Output Array. If the column ends before the end of a 4-byte boundary, the unused bytes are masked off to 0 to get rid of any extraneous bytes that are there from subsequent delimiters or columns.

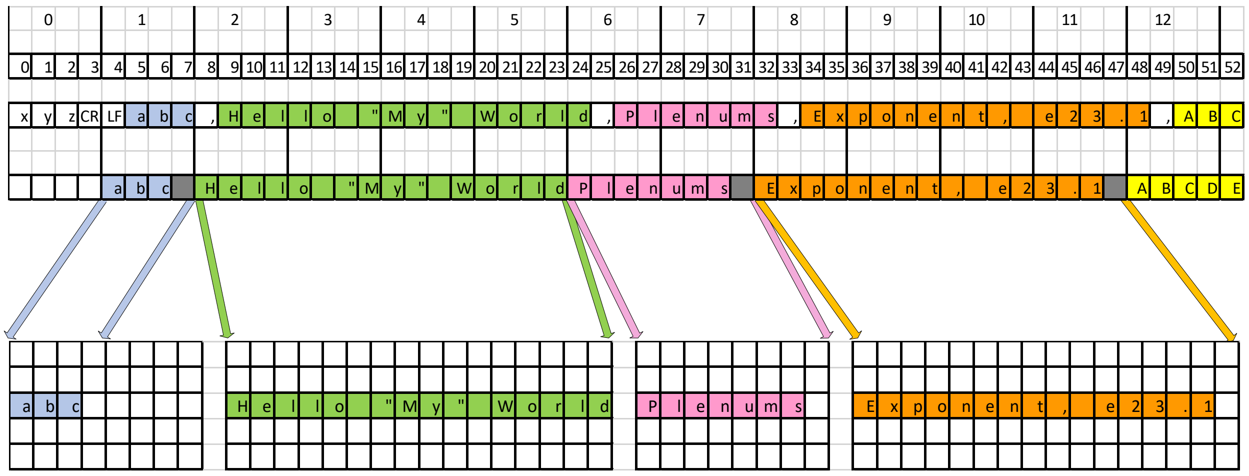
The figure below shows the first part of a sample chunk. The tall vertical bars simply demark the 4-byte boundaries representing individual threads.



	0				1				2				3				4				5				6				7				8				9				10				11				12			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
x	y	z	CR	LF	a	b	c	,	H	e	l	l	o		"	M	y	"		W	o	r	l	d	,	P	l	e	n	u	m	s	,	E	x	p	o	n	e	n	t	,		e	2	3	.	1	,	A	B	C
				a	b	c	⬛	H	e	l	l	o		"	M	y	"		W	o	r	l	d	P	l	e	n	u	m	s	⬛	E	x	p	o	n	e	n	t	,		e	2	3	.	1	⬛	A	B	C	D	E

Figure 18

You see the printing characters of each column in colors in the middle row. The bottom row represents the positions of the printing characters after the shuffling is complete. Each column now starts at the beginning of a 4-byte boundary. Note that the quotes which surround two of the fields, as well as the escape quotes inside the second field, are gone. The dark gray bytes represent the masking that is done to allow full 4-byte writes each time, but eliminate extraneous bytes from the shuffle operations.



0 1 2 3 4 5 6 7 8 9 10 11 12
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
x y z CR LF a b c , H e l l o " M y " W o r l d , P l e n u m s , E x p o n e n t , e 2 3 . 1 , A B C
a b c H e l l o " M y " W o r l d P l e n u m s E x p o n e n t , e 2 3 . 1 A B C D E
a b c
H e l l o " M y " W o r l d
P l e n u m s
E x p o n e n t , e 2 3 . 1

Figure 19

Note that what is shown in the figures above is an oversimplification. It gives a general idea, but it overlooks the fact that the shuffling may “reuse” some of the “same” bytes. For example, suppose Column A comes in aligned at the beginning of a 4-byte boundary, and its last byte is byte 0 of fifth thread in the warp. Column B comes just after Column A. Its beginning is picked up in the fifth thread at byte 2 (byte 1 is the delimiter). After shuffling, the first byte of Column B will be in the byte 0 position of the fifth thread – the same position as occupied by the last byte of Column A! This does not cause a problem, because the shuffle and write operations are done sequentially by the 4 byte positions in a thread, so that Column A’s last byte is written out before Column B’s first byte.

The Core Kernel Logic

The Core Kernel (WriteCSVRecord2) begins as follows:

```
// pass in:
// the "CSV" buffer, the records table, the col count errors headers, the col count
errors scan,
// the columns table, the records to columns table,
// the fields count (# of columns), the records count, and the character multipler
(convert char sizes to byte sizes -- normally 3, but can be 1 for ease of debugging).
__global__ void WriteCSVRecord2(uint8_t *  d_Buffer, uint32_t *  d_RecsTabl,
       uint32_t * d_ColumnCountErrors, uint32_t *d_ColumnCountErrorsSCAN,
       uint32_t *  d_ColsTabl, uint32_t *  d_RecsToColsTabl,
       int fieldcount, uint32_t recordscount)
{
```

Remember that we already have three highly reused values in Constant Memory when this kernel runs:

- Field Character Widths
- Field Byte Widths
- The Pointers to each of the Output Arrays

The Core Kernel goes an outer and an inner loop.

- The outer loop is a loop through all the 128-byte chunks containing the record.
- The inner loop processes through each column that is contained within that chunk.

One of the techniques we use to maximize memory performance involves unions which allow us to take advantage of 4 or 8 byte memory accesses, rather than bytewise. This is a standard GPU optimization, sometimes referred to as Load-As. It involves casting.

```
// unions for loading as...
union inputbytes_t
{
	uint32_t inputuint;
	unsigned char inputbytes[4];
};
// similar to above but uses signed chars
union infou32_i8_t
{
	uint32_t infouint;
	char infobytes[4];
};
union infou32_u8_t
{
	uint32_t infouint;
	uint8_t infoubytes[4];
};
union infou64_u16_t
{
	uint64_t infouint64;
	uint16_t infouint16s[4];
};
```

The kernel does a number initialization steps. The thread index and Lane ID are calculated, and the shared memory is declared to read the next full chunk. The current warp number is calculated, which will correlate to a record number. All threads in this warp are responsible for one record, unless we are past all records. Adjustments are made for erroneous records, which are skipped. Based on whether or not we are the first warp, looking at the first record, we determine the record start byte in the CSV, the length of the record, and the index into the Columns Table at the start of the record (using the Records-To-Columns Table if not at the beginning of the CSV). We also calculate how many 128-byte chunks we will need to process the entire record.

Note that except for the thread index and the Lane ID, the calculations in the code below are based on the Warp Number. Even though all threads in the warp will do the computations and set local variables, the values will be same for *all* threads in the warp, since they all share the same Warp Number.

```
int ix = blockIdx.x * blockDim.x + threadIdx.x;
unsigned char laneid = (unsigned char)ix & 0x1f;  // lane 0-31 w/in this warp.

extern __shared__ uint32_t nextchunkvals[];  // available to read 1 chunk ahead.

// this gets the warp number in the device (not the block).
uint32_t warpnumindevice = ((blockIdx.x * blockDim.x) + threadIdx.x) / warpSize;
int warpnuminblock = threadIdx.x >> 5;  // int divide by 32, throw away remainder

// Since we are making each warp take care of its own record,
// if we have more warps than records, simply have the whole
// warp end early - no work to do.
if (warpnumindevice >= recordscount) return;  // ignore anything in last block beyond
                                                 // source arrays length.

// however, we only do records without errors, so exit if this record has a
// col count error.
if (d_ColumnCountErrors[warpnumindevice] != 0) return;

// NOW have a non-error record to do.
// now must adjust destination record index.
// subtract off the errors to this point from exclusive scan.
// DO NOT write out data for a error record.  Skip it.
uint32_t destination_recindex = warpnumindevice -
d_ColumnCountErrorsSCAN[warpnumindevice];  // skip up to adjust for dropped col count
                                                 // error records.

uint32_t recstartbyte;
int32_t ColsTableIdxAtRecStart;

// get rec length.  NOTE: later use records table if have a cap at end.
uint32_t reclen;

if (warpnumindevice == 0)
{
       recstartbyte = 0;
       ColsTableIdxAtRecStart = -1;  // flags on first record.

       reclen = d_RecsTabl[0] - 2;
}
else
{
       recstartbyte = d_RecsTabl[warpnumindevice - 1] + 1;  // get the record for this
                                                 // warp.  add 1 to bump past header (LF)
       ColsTableIdxAtRecStart = d_RecsToColsTabl[warpnumindevice - 1];  // get index in
                                    // cols table for the rec delim for start of this record.

       reclen = (d_RecsTabl[warpnumindevice] - (recstartbyte - 1)) - 3;
}

// this calcs how many chunks reading.
```

```
uint16_t chunk128count = (uint16_t)((reclen + recstartbyte) >> 7) -
(uint16_t)(recstartbyte >> 7) + (uint16_t)1;
```

Before the outer loop, we also set up some local variables that will be used throughout the loops. Note that many of these are *lane-specific*. This means there values will depend on the lane, namely the relative thread within the warp (lanes 0 – 31).

```
// tracking for each lane, by each byte in lane.
infou32_u8_t RelColNum;  // This is the 1-based relative column num within the warp for
the current chunk.
infou32_i8_t ShufBack;  // This is the number of byte positions to shuffle back for each
byte:  0 - 3
infou32_i8_t Bytes2Write;  // This is the number of bytes to write from the current
position when it is shuffled back to byte 0 positon: 0-4
infou64_u16_t DestByteOffset;  // This tracks the offset in the destination buffer for
the current set of bytes, once aligned.

uint16_t ChunkColumnStart;  // this is the 0-based column # for the start of each chunk.
uint16_t temp16;  // short term 16 bit num.

// housekeeping that spans the chunks.
uint32_t CurColStartingByte, CurColEndingByte;
uint32_t curcolheader, nextcolheader;
uint16_t N;  // 1-based col number for entire record.
char ChunkDone = 0;  // flag used when breaking out of columns loop to go to next chunk.
char haveColStartEnd = 0;  // flag set between chunks, when already have col start and
end in local vars, so don't need to access memory again.
```

The outer loop is fairly straightforward and has already been explained. We will examine the inner loop in detail, but also refer to the logic that takes place when the chunks progress. Since all threads in the warp are active at once, each cycle of the outer loop completes when all threads in the warp (all lanes) “agree” that they are done.

There is a tracking variable N which starts at 1 and progresses through all the columns. This is carried over between chunks. At the beginning of the inner loop we set N, based on which chunk we are in:

```
// set chunk col start here.  if chunk 0, set to 0.
if (curchunk == 0) ChunkColumnStart = 0;
// otherwise build with carryover N.
else ChunkColumnStart = N - 1;
```

We also initialize some variables we use within the inner loop. These are based on unions defined above.

```
RelColNum.infouint = 0;  // 0 means not evaluated yet.
ShufBack.infouint = 0;  // 0 default no shuffles back.
Bytes2Write.infouint = 0;  // default no bytes to write.
DestByteOffset.infouint64 = 0;  // default no offsets.
```

Now we start the inner loop. We set N to 1 or, for chunks after the first, a number based on where we left off. We check if we are done.

```
// RelCol is column relative to current chunk,
// will start at 1 for first chunk in a record.
// loop is to max # rel cols, but normally break earlier.
// therefore its max is 128 which would be for a 128 byte chunk
// of only column headers.
for (uint8_t RelCol = 1; RelCol <= 128; RelCol++)
{
       // now get N by adding RelCol to ChunkColumnStart
       N = ChunkColumnStart + RelCol;

       // if now have an N > field count (even though relcol is not),
       // done with this chunk and also this record for byte logic.
       if (N > fieldcount)
       {
              ChunkDone = 1;
              break;  // break out of col loop.  below will proceed to
                      // next chunk, but this will have been the last one.
       }
```

The purpose of the inner loop and the N variable is essentially to partially serialize what otherwise would be strictly parallel with all threads acting at once. As N progresses through columns, it essentially "highlights" one column at a time going from left to right, and in so doing, one or more threads containing the given column will process while others will remain quiet. Because N is a loop variable, it will have the same value in all threads at the same time, whereas other variables will thread-specific.

The inner loop continues by checking if we are in the first chunk. If so, are we at the very beginning of the CSV? We set up housekeeping accordingly, and use the Tables if needed to figure out the start and end of the current column, based on whatever N we are on

```
// if coming in on a new chunk will already have calc'ed col params
// for first col in chunk
if (haveColStartEnd == 0)
{
       // at the very beginning, curcol hdr idx = 0, next is first entry.
       if ((ColsTableIdxAtRecStart == -1) && (N == 1))
       {
              nextcolheader = d_ColsTabl[0];
              CurColStartingByte = 0;  // start byte is very first byte
       }
       // not at the very beginning.
       else
       {
              // reuse nextcolheader if have one.
              if (nextcolheader != 0)  curcolheader = nextcolheader;
              else curcolheader = d_ColsTabl[ColsTableIdxAtRecStart + N - 1];

              // read nextcolheader from cols table.
```

```
            nextcolheader = d_ColsTabl[ColsTableIdxAtRecStart + N];

            // get start byte index for column
            CurColStartingByte = curcolheader + 1;  // byte after | or LF
        }


        // have specs on Column N.  now get last byte index for col.
        if (N == fieldcount) CurColEndingByte = nextcolheader - 2;  // byte before CR LF
        else CurColEndingByte = nextcolheader - 1;  // byte before |


        // check if at a col that starts after this chunk.
        // NOTE: even though don't need CurColEndingByte (calc'ed above) to do
        // this comparison, set it anyway,
        // so we will be ready for reuse in the next chunk
        if (CurColStartingByte >= ((recstartbyte & 0xffffff80) + ((curchunk + 1) * 128)))
        {
            ChunkDone = 1;
            haveColStartEnd = 1;  // don't need to re-read entering next chunk.
            break;  // out of cols loop.
        }
}
else haveColStartEnd = 0;  // ensure this is reset since this is only carried over
                            // between chunks.
```

Now we know which column we are processing in the loop, based on N. N and the column information are the same across all threads. However, now each thread must look into its own lane and the bytes in it (4, unless at the very end of the CSV), to check which bytes participate in the column.

For each byte in the lane we calculate how far it must be shuffled back so that the start of the column data will start on an even 4-byte boundary – zero through three bytes to shuffle. This of course is based on the position of the first byte of column data.

Note that we have already cleaned up non-printing characters based on our comma-delimited pre-processing, so we assume every byte between a column start (a delimiter character or a linefeed) and a column end (a delimiter character or a carriage return) are included in the column content to write to an Output Array.

The section of code below shows this processing. A lot of housekeeping is done so that the each byte has information on its destination. Each byte will track how many bytes it writes forward from its position for the current column it is in, if it will be the *first* byte to write in an aligned shuffle. (Keep in mind that each lane may contain bytes for multiple columns. Also some columns may be empty. They still are tracked, but no bytes are to be written.)

```
// now does this column apply to the bytes in our lane?

// loop through all four bytes and check individually.
for (char ByteIdx = 0; ByteIdx < 4; ByteIdx++)
{
        // if this is the first col, then flag out any bytes that are less.
       if ((N == 1) && (CurColStartingByte >(LaneByte0IndexIntoBuf + (uint32_t)ByteIdx)))
       {
              RelColNum.infoubytes[ByteIdx] = (char)0xff;  // flag byte prior to start.
       }
       // if this is last col, then flag out any bytes after the content.
       if ((N == fieldcount) && ((LaneByte0IndexIntoBuf + (uint32_t)ByteIdx) >
CurColEndingByte))
       {
              RelColNum.infoubytes[ByteIdx] = (char)0xff;  // flag byte past end.
       }

       // otherwise is the cur col relevant to the current byte?
       // start of col has to be <= curbyte, and end has to be >= curbyte – 1
       // (the - 1 allows for 0-length cols).
       if ((CurColStartingByte <= (LaneByte0IndexIntoBuf + (uint32_t)ByteIdx)) &&
(CurColEndingByte >= (LaneByte0IndexIntoBuf + (uint32_t)ByteIdx)))
       {
              RelColNum.infoubytes[ByteIdx] = (char)RelCol; // flag rel col for cur byte.

              // now how far will cur byte have to be shuffled back for an aligned write.
              // check the position of the start of the column.  we put this on all bytes
              // in the col, since applies to all.
              ShufBack.infobytes[ByteIdx] = (char)(CurColStartingByte & 3);

              // the destination offset for the current byte is its position less
              // the position of the starting byte.
              DestByteOffset.infouint16s[ByteIdx] = (uint16_t)((LaneByte0IndexIntoBuf +
(uint32_t)ByteIdx) - CurColStartingByte);

              // finally set bytes to write for cur byte.  this reflects how many
              // bytes to write forward, once this byte is aligned.
              // so should only be set for a byte which will be the first in the
              // aligned shuffle.
              // ex. if the align/shuffle back for this col is 3, the byte in question
              // would be 3.
              if (ShufBack.infobytes[ByteIdx] == ByteIdx)
              {
                      temp16 = (uint16_t)((((int32_t)CurColEndingByte + (int32_t)1)
((int32_t)LaneByte0IndexIntoBuf + (int32_t)ByteIdx)));
                     if (temp16 > 4) Bytes2Write.infobytes[ByteIdx] = 4;  // max 4 bytes.
                     else Bytes2Write.infobytes[ByteIdx] = (char)temp16;

                     // the destination offset for the current byte is its position
                     // less the position of the starting byte.
                     DestByteOffset.infouint16s[ByteIdx] =
(uint16_t)((LaneByte0IndexIntoBuf + (uint32_t)ByteIdx) - CurColStartingByte);
              }
              else Bytes2Write.infobytes[ByteIdx] = 0;  // if 0-length col or not
                                                        // an aligned start, 0 to write.
       }
}
```

At this point, we determine if the current column, based on N, continues into the next chunk, which is in shared memory. If so, flags are set so processing of this column can continue into the next chunk, and we break out of the inner columns or N loop.

```
// now at end of byte processing, check if our ending byte is in the next chunk.
//// if (CurColEndingByte >= (recstartread + ((curchunk + 1) * 128)))
if (CurColEndingByte >= ((recstartbyte & 0xffffff80) + ((curchunk + 1) * 128)))
{
       // ChunkColumnStart = N - 1;
       ChunkDone = 1;
       haveColStartEnd = 1;  // don't need to re-read entering next chunk.
       break;  // out of cols loop.
}
```

At this point, we do the shuffle down from one lane to the earlier one the 4 bytes of the lane (in unsigned 32-bit Int), one value per thread/lane. If we are in the last lane (31), but not at the last chunk, we pull the shuffled value in from the first 4 bytes of shared memory.

```
// do the shuffles here because shouldn't do inside conditional.
// will manipulate below using these.
uint32_t shuffledown = __shfl_down(inval.inputuint, 1);
// can't shuffle into lane 31. so get from shared or set to 0.
if (laneid == 31)
{
       if (curchunk < (chunk128count - 1))
       {
              // if not the last chunk read the shuf value from FIRST byte in shared
              // for this warp.
              shuffledown = nextchunkvals[(warpnuminblock * warpSize)];
       }
       else
       {
              // if there is no next chunk, make the shuf value 0.
              shuffledown = 0;
       }
}
```

The next part of the code loops through the 4 bytes in the lane, and effects an actual shuffle, using shifts, to achieve alignment. An unsigned 32-bit integer is built up for a potential write from this byte position. Depending on how many bytes are to be written from the current byte position in the lane, masking is used to write only the correct number of bytes (zeros are written beyond that number). If there are bytes to write, these are written to the appropriate column, in the appropriate position, always on a 4-byte boundary within the column.

```
// now loop through the "shuffles"
for (char shufflecount = 0; shufflecount < 4; shufflecount++)
{
       // NOTE: the vars below may cause undue register pressure.
       // we can swap out for the values they get set to here in code below.
       // get the bytes to write for the byte in the position.
       char curbytestowrite = Bytes2Write.infobytes[shufflecount];
```

```
        uint8_t currelcol = RelColNum.infoubytes[shufflecount];
        uint16_t curdestoffset = DestByteOffset.infouint16s[shufflecount];

        // ALSO: could even get rid of writeout by putting one big expression
        // in the write.
        uint32_t writeout;


        // if (curbytestowrite > 0)
        if (Bytes2Write.infobytes[shufflecount] > 0)
        {

                // First build the raw 32 bits properly shuffled.
                //Shuffle to shift everything "down" 8 bits, trying to achieve alignment.
                if (shufflecount == 0)
                {
                        writeout = inval.inputuint;
                }
                else if (shufflecount == 1)
                {
                        writeout = inval.inputuint >> 8;  // this will reduce the value
                                                           // by 256
                        writeout |= ((shuffledown & 0xff) << 24);  // this puts the low byte
                                            // of the next into the high byte of the current.
                }
                else if (shufflecount == 2)
                {
                        writeout = inval.inputuint >> 16;  // this will reduce the value
                                                            // by 256 * 256
                        writeout |= ((shuffledown & 0xffff) << 16);  // this puts the 2 low
                                    // bytes of the next into the high 2 bytes of the current.
                }
                else  // shufflecount == 3
                {
                        writeout = inval.inputuint >> 24;  // this will reduce the value by
                                                            // 256 * 256 * 256
                        writeout |= ((shuffledown & 0xffffff) << 8);  // this puts the 3 low
                                    // bytes of the next into the high 3 bytes of the current.
                }

                // now mask off if there are less than 4 bytes to write.
                if (curbytestowrite == 3) writeout &= 0x00ffffff;
                else if (curbytestowrite == 2) writeout &= 0x0000ffff;
                else if (curbytestowrite == 1) writeout &= 0x000000ff;

                // now write the 32 bits out.
                int16_t curcolbytewidth = d_fieldbytewidths_C[(currelcol +
ChunkColumnStart) - 1];
                if ( curcolbytewidth > 0 )
                {
                        *(uint32_t*)(d_fieldptrs_C[(currelcol + ChunkColumnStart) - 1] +
(destination_recindex * curcolbytewidth) + curdestoffset) = writeout;
                }

        }

}
```

At the end of the 4-byte shuffles loop, we are at the end of the chunk loop. After the chunk loop, the kernel exits.

```
            // if chunk done flag set
            if (ChunkDone == 1)
            {
                  ChunkDone = 0;  // reset it.
                  continue;  // continue to next chunk.
            }
      }  // end of chunk loop.

      return;
}  // end of WriteCSVRecord2 kernel.
```

Completion of Core Processing

Once the Core Kernel has completed, all the Output Arrays are fully populated with CSV data. At this point the Output Arrays can be copied back to the CPU for consumption. However, typically they will remain in the GPU for further processing.

Meanwhile, all the GPU memory, except for the Output Arrays, can be freed at this point, if it has not been freed already. (Normally, any GPU memory should be freed as soon as it is no longer needed – especially considering the high demands on GPU memory.)