Fast JSON parser using metaprogramming on GPU

Krzysztof Kaczmarski*, Jakub Narębski†, Stanisław Piotrowski* and Piotr Przymus†

*Warsaw University of Technology, Poland
Email: krzysztof.kaczmarski@pw.edu.pl, stanislaw.piotrowski717@gmail.com

†Nicolaus Copernicus University in Toruń, Poland
Email: jakub.narebski@mat.umk.pl, piotr.przymus@mat.umk.pl

***Abstract*—We demonstrate a new idea of a parallel GPU JSON parser, which is able to optimize the parsing and initial transformation process through metaprogramming. It outperforms other well-known solutions like *simdjson*, *Pandas*, as well as *cuDF*– which also works on GPU. The resulting data is ready to be further processed in common data frame formats and may be incorporated by *RAPIDS*, *Apache Arrow* or *Pandas*. Our parser can therefore be a part of an industrial Extract-Transform-Load workflow.**

***Index Terms*—parallel parser, metaprogramming, ETL, JSON, GPU.**

I. Introduction

Data loading is no longer a one-time operation, as modern data analysis often requires permanent stream processing. As part of a typical workflow, various sources are used for Extraction, Transformation and Loading (ETL) into machines specialized for processing data and producing meaningful information for an online database, an AI system or a business intelligence warehouse. Most databases and machine learning applications use data organized in rows or columns as inputs. While binary data exchange formats are becoming increasingly popular, pure text formats still prevail. CSV data representation for ETL is attractive since it can be almost immediately loaded. Object structures like XML [1] or JSON [2], [3] are more expressive than CSV by preserving structure of multi-dimensional information, which is often essential in recent heterogeneous systems.

JSON format due to its object nature, human readable format, and simplicity is often used by various REST [4] web applications for data serialization. Especially JSON lines (*jsonl* [5], *ndjson* [6]) is useful as it consists of a collection of objects at root level which are separated by end-of-line characters and can be incrementally loaded as needed. Variety of tools were designed around this concept, including Logstash, plot.ly, Scrapy, Apache Spark, ArangoDB, Rumble, Neo4j, BigQuery, Airbyte and other.[1]

A. Motivation

1) Utilization of domain data knowledge: JSON appears to be often used in combination with languages like C#, JavaScript, PHP, or Python, which may alter objects structure at the runtime. In such cases, the structure of the JSON objects being accessed may remain unknown. Applications often do not make any a priori assumption on the structure of the underlying data and a general parsing technique is used. However, data manipulation could benefit from some speculative runtime assumptions or domain knowledge [7]:

- dictionary elements can have constant names, types, and order,
- data are read only partially and its usage presents a stable access pattern; some data are constantly omitted,
- even if there is no strict JSON schema, there are typically only a limited number of different structural variants of a JSON file,
- there is possibility of determining summarized structural information quickly by some indicating special case characters.

JSON simple types are limited to *None*, *Number*, *Bool* and *String*. Therefore, there is a need of automatic transformation of the encoded data to domain specific types like: date, time, color, geographical coordinates, etc. Transformations at runtime can result in an overhead. On the other hand, a specialized parser/transformer constructed and compiled for given JSON schema could significantly reduce the overall parsing and processing time.

2) Parsing process optimization: Extraction of text data consists of at least two steps. The first step involves single thread parsing and tokenizing of the input, while many consumer threads wait to push data to the next stage. The CPU thread is the bottleneck during this step [8]. The problem of a single CPU thread blocking the database input can be solved with parallel threads, if data stream can be buffered and read independently. This is possible only when a communication between processes is not necessary. However, using a parallel database requires specialized locks in order to preserve integrity and speed. For these reasons, data loading still remains a bottleneck for the fastest data sources.

In many cases complex data processing is now forwarded to GPU devices which have become standard accelerators of Machine Learning since their introduction in 2007[2]. GPU

Research was funded by POB Cybersecurity and data analysis of Warsaw University of Technology within the Excellence Initiative: Research University (IDUB) programme.

[1]A complete list available at https://jsonlines.org/on_the_web

[2]We consider the first CUDA release to be the beginning of industrial general purpose GPU programming.

devices excel at parallel data reading as they have a massive number of threads. We believe that utilization of a massive number of GPU threads may improve parsing bottleneck.

3) Nested objects data parsing: Data are subject to exchange in various forms and formats. Binary form is usually faster but not as transferable and portable as textual form. In heterogeneous systems, text data is usually the only choice. Efficient multi core parallel parsing of nested textual objects is a challenge. Currently, only *simdjson* [9] and *RAPIDjson* library can effectively utilize CPU SIMD commands for arbitrary nested JSON objects parsing. A GPU oriented *cuDF*[3] library from [10] can use multiple GPU threads but is currently limited in objects nesting – it can only parse flat JSON objects.

4) Data preparation for database offloading: In real life systems, the simplicity of creating JSON data on the fly by serializing runtime objects leads to heterogeneity of produced datasets. These datasets must be transformed [11] before database loading can accept them. This leads to unwanted effort in database processes or requires another stage of processing and resources allocation. We perceive parallel initial data transformation being done within parsing process as a way of saving resources and database time.

5) Sample usage scenarios that can benefit from the proposed solution: The proposed solution is aimed at applications in data processing pipelines using GPU calculations, where significant amounts of data coming in batches are processed and the JSON scheme is known. This allows one to easily optimize existing data processing pipelines by compiling a specialized JSON parser. Our parser integrates with *cuDF* which in turn allows for integration with the *Dask* framework, making it possible to use multiple GPU cards in a cluster or in a single machine. The use of multiple GPU cards helps to overcome the memory limitation of a single device.

B. Contribution

In this paper, we present a parallel parser which is able to outperform current JSON parsers, including these working on GPU, while supporting transformations which can be dynamically encoded. This task is achieved by metaprogramming techniques and GPU computing. Our tool specializes code for a given JSON schema and transformations to generate optimal parser and transformer code. The JSON schema can be given directly, or can be inferred from the sample data.

In many industrial applications JSON data have got strict repeatable object structure with well defined properties having names and types. For example, consider a well defined REST API, changes of serialized objects and consequently JSON schema will be rare and well documented, thus it is reasonable to construct a specialized parser.

1) Improve parsing JSON data to columnar format by utilizing metaprogramming: To our knowledge, there is no other tool which uses metaprogramming to generate a parser specialized for converting JSON lines to columnar format that takes advantage of the many opportunities to repeat the same operations while reading the input when performing this task. Instead of building a general parser, we opt for preparing a parser's internal behavior to the expected and repeatable input.

2) Utilize metaprogramming techniques to improve parallel operations and memory management: We propose a novel mechanism of managing the parallel parser operations and memory management through metaprogramming and precompilation configurations. Our parallel GPU workers are optimally generated to read the expected input data.

3) Incorporate ETL transformations into parser via metaprogramming transformation description: Utilization of high bandwidth GPU memory and GPU instruction throughput leaves enough power to perform additional data transformations while still being faster than the data ingestion. Data values selection, validation, transformation, types conversion, and simple arithmetic operations can be done on the fly without losing high throughput capabilities.

4) Integration of the solution with an industry standard libcudf and other similar libraries: Our tool is fully integrated with *libcudf* data structures and can be used in any other solutions which use *cuDF* data frames on GPU side in parallel computation. This shortens the processing steps necessary to achieve computational goals. It is possible to load JSON data using *Meta Parser* and then analyze it using *cuDF*.

II. Related work

A. JSON Parsers

JSON (Java Script Object Notation) has a few data types in two categories: *simple* (*String*, *Number*, *Boolean*), and *aggregated* (*Array* and *Object*), which can contain named or unnamed values of different types. Nesting causes JSON objects to have a tree-like structure. The standard does not guarantee fixed order or even presence of elements in objects. Parsing nested objects with lack of schema enforcing is much more difficult than parsing plain columnar formats like CSV, since there is no easy way to distinguish and predict the next expected values. For example in Fig. 1, the same path `obj["a"][1]["b"]` exists in three completely different structures which are valid JSON objects.

```
{
  "xxx": 1,
  "a": [
    true,
    {
      "b": "text1"
    }
  ],
  "yyy": null
}
```

```
{
  "a": [
    "some text",
    {
      "b": "text2"
    },
    "more text"
  ],
  "c": [
    3.14,
    3.1415
  ]
}
```

```
{
  "zzz": {
    "xyz": 42
  },
  "a": [
    1234,
    {
      "c": "not",
      "a": "no",
      "b": "text3"
    },
    5678
  ]
}
```

Figure 1: Three different JSON objects with a common path to a value. Variability of types in general format complicates parsing.

A parser needs to keep track of the path that was already traversed through JSON. Conversion to a tree-like structure

[3]We use *cuDF* to distinguish python RAPIDS interface and *libcudf* to name its C++ library.

of an object, may still be the most time consuming part of a Big-Data application [12]. Xie et al. checked that a general JSON naive parser can achieve only around 100MB/s per CPU core [13]. Optimization of JSON parsing is a challenge for the database and data analysis community.

One of the most obvious strategies is not to parse all the input data, but to only parse the required parts. Those parts can be defined by queries, and its destination can be localized by searching for special markers and by counting objects in sequences without converting all of them to data structures. It can be very efficient for CSV data [14]. *Mison* system [15] performed the same for JSON files by jumping directly into queried objects. This approach differs from the approach used by typical finite state machines based parsers. *Mison* finds marking characters like [',', ';', '"', '{', '}'] and builds bit indexes in order to identify parts of data for further investigation. A similar approach is utilized by *Sparser* [12] which filters input data to find the relevant information. This technique may be used to localize JSON lines and the content of objects in order to read only the fields which are to be parsed. In a similar way, one may divide content of an array or a big object into pieces and send these pieces to parallel processes independently. Our parser uses end-of-line character location analysis to separate subsequent JSON lines objects in the input.

FAD.js uses JIT parser compilation to achieve higher performance [7]. It is specialized to certain object structures which are identified at the runtime. It also generates machine code optimized for parsing only the expected properties and values. We use a similar technique by defining a relevant object's fields in a meta-description, which is later translated to a parser behavior. Thanks to this technique, GPU workers are optimized to look for the expected keys and values.

It should be noted that selective parsers can be used only in a limited number of applications. In data analysis, instead of loading a single object, we mostly need an entire column composed of selected properties from all objects of a given type. Therefore, scanning all the input data cannot be avoided. We only use parser specialization inside objects to build desired columnar information from all objects.

Pandas library [16], a standard Python tool for scientists, creates columnar data representation in so called *Data Frames*. It loads entire JSON file using standard Python JSON parser and then tries to infer which fields can be converted to columns.In contrast, our solution uses programmer's domain knowledge before parsing and therefore temporary JSON tree-like representation is not created.

SIMD CPU instructions open another direction by introducing data parallelism utilizing 128 to 512 bit long words. Langdale and Lemire [9] showed how these instructions can be used to increase the bandwidth of JSON file parsing. Their *simdjson* is currently perhaps the fastest CPU based JSON parser. *Mison*, *Sparser* and *FAD.js* offer parsing speed of around 2GB/s per CPU core [13]. *simdjson* can achieve 3GB/s on certain input files [9]. However, it stores the result in an own tree-like object structure.

A JSON parser delivering data to a GPU device is included in RAPIDS system [10], a collection of open source tools for data science on GPU. It creates a columnar data representation. In case of JSON lines it utilizes its own parser (from *libcudf*), while it falls back to Pandas parser in case of a classic JSON file and then converts the output to its format.

Python is used as a front-end language while C++ and GPU kernels work in the back end. A programmer may use their favorite data science tools which work on columns in Python or may directly refer via *libcudf* to vectors in memory and run custom GPU kernels. RAPIDS already proved to be an efficient solution for machine learning [17], [18].

Since *libcudf* is, to our knowledge, the only tool which is able to load JSON into GPU memory, it will be in center of our comparative analysis. Our solution returns GPU memory buffers in a format totally compatible with RAPIDS. Thus, any Python program may reuse them and benefit from RAPIDS and GPU acceleration.

B. Metaprogramming for GPU

Metaprogramming is a popular technique for generating programs from other programs in order to achieve optimization, program specialization, code simplification, and other benefits. Although metaprogramming is widely used in many programming languages, there are only two major approaches for GPU programming. The first one is based on a C++ compiler and its *template programming*. This is not a very programmer friendly interface but allows for typical notions like meta-functions which are evaluated during the compilation. This allows for powerful code generation, though it is difficult for a human to understand. Template programming was for example used to create *Expression Template* linear algebra libraries for GPU [19]–[24] which generated optimized GPU kernels, saving unnecessary memory accesses and allocations, when performing vector and matrix expressions evaluation.

We use a similar C++ technique based on the MP11 MPL.Boost library. Our GPU workers and memory allocations are configured and generated during compilation from C++ templates. *Meta Parser* behavior is composed of actions which are defined by nested template structures and are a source for a compile-time C++ code generation.

The second approach uses a separate language and its compiler to embed or generate C++ code which is then compiled for GPU processor. This strategy is used by *pyCuda* and *pyOpenCL* [25], *jCuda* [26], and *Numba* [27].

Using a general purpose programming language to generate GPU source code has got several advantages: friendlier syntax and semantics, better error reporting, more libraries which can be incorporated in the process. On the other hand it also needs an additional compiler with all its supporting environment.

Due to the complicated syntax of C++ templates, we decided to prepare a Python interface for our solution. The parser definition can be setup in a very similar way to *cuDF* or *Pandas*, which simplifies the overall process.

We noticed that the C++ compiler runs slowly when it encounters templates in a source code. This influences the

parser creation process so much, that we considered moving all of the code generation to a tool like *pyCuda* to emit C++ templates. Nevertheless, compile time can be amortized when the same parser/transformer is used multiple times or on sufficiently big input data.

III. Proposed approach

In every ETL process there must be domain knowledge on data involved in order to prepare meaningful input for the database loading stage. A typical textual data extraction parser requires the schema to be defined. Usually JSON parsers try to guess it. A user's role is to convert the data being parsed to a format acceptable by the transformation stage (data filtering, cleaning, normalization, etc.). At this stage domain knowledge is necessary even if it was not used before.

In an ETL process extraction is followed by transformations T_1 to T_n which prepare data to be loaded to the next analysis stage (Fig. 2, top). A general parser usually reads all of the input data in memory, including fields which are not required. This step cannot be avoided for JSON files because the parser is not able to predict which fields may appear. Memory and processing time is wasted as later transformations may only use a subset of these fields.

The *Meta Parser* approach tries to overcome this problem by using or guessing a schema to specialize the parser to read certain data fields and combine it with transformations (Fig. 2, bottom) in order to save memory and computation time. This approach is different from other solutions like Pandas, because the schema is needed in a preparation stage and leads to parser specialization.

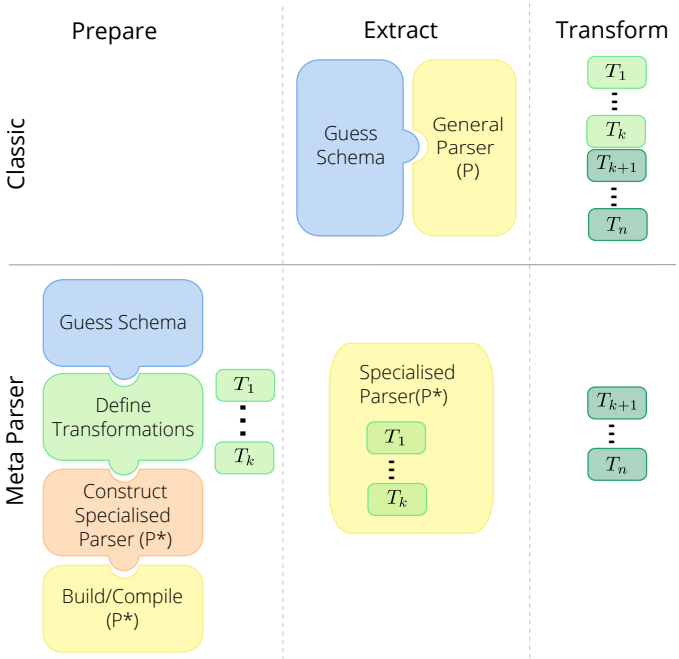


Figure 2: Comparison of a classical parser and transformation process vs. *Meta Parser* approach.

During the preparation stage, a programmer uses domain knowledge to define a specialized JSON parser/transformer, either by specifying the expected JSON schema and transformations $T_1 \ldots T_k$, or allowing the JSON schema to be guessed and specifying transformations afterwards (Fig. 2, bottom). The point when the domain knowledge is used is changed which leads to formalization of the data transformation before the process begins and in fact may improve the overall quality of the system design. It must be noted that not all the transformations can be done while scanning the file. Some of them ($T_{k+1} \ldots T_n$) must be postponed until all the data resides in memory.

Once this process is finished, a specialized parser is ready to be run continuously for a stream of data performing selected data transformations. The overall number of transformations which need to be performed in order to prepare data for loading decreases. If the data structure and transformations change, *Meta Parser* building needs to be done again. While this may seam limiting, mind that in many cases the JSON lines data will be generated by the same library or REST API, thus its structure will be constant. Even in case of structural changes, there may be a cache of compiled parsers allowing for automatic parser selection based on the data.

IV. Description of the solution

A. Software libraries design

Meta Parser has two layers: an inner core C++ library, and the outer Python wrapper with interface similar to *cuDF* or *Pandas*. The Python interface may be used to guess the JSON schema of a data sample, enrich it with transformations, and generate C++ template description. A wrapping tool may automatically call the C++ compiler, load compiled GPU code and control all the data loading. In special circumstances, meta definition of the parser may be also prepared manually. An example of code involved in the process starting from a JSON object to generated C++ templates is shown in Fig. 3.

The C++ library is in fact an independent part and may be used in three different scenarios (Fig. 4). Headers only library (the green path) allows making additional adjustments before the final compilation, but significantly extends the compilation time. Static library (the red path) can be easily linked in a program compilation, and it does not extend the compilation time. The blue path presents a compilation path of a dynamic library, which allows swapping parsers from a cache without stopping the program.

B. Meta Parser definition language

The core part of our solutions is built in C++ templates. Although their interface is not easy to use, they constitute a powerful meta-programming language. C++ templates are also supported directly by NVIDIA compilers, which makes combining our metaprograms with GPU relatively straightforward. Additionally, writing complex metaprograms may be easier thanks to several metaprogramming libraries for C++, like MPL, MP11, Metal, Kvasir or Hana [28].

```
{ "id": 1, "year": 1963, "salary": 5000, "name":
"John Doe", "position": "worker"}
```

```
{ "$schema": "http://json-schema.org/draft-04/[...]",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer"
    },
    "year": {
      "type": "integer",
      "transformation": "Cut([0,1968,1970])"
    },
    "salary": {
      "type": "integer",
      "transformation": "MinMax(1000,8000)"
    },
    "name": {
      "type": "string",
      "transformation": "Hash()"
    },
    "position": {
      "type": "string",
      "transformation": "Map({\"boss\":10,
                             \"head\":20,
                             \"worker\":30})"
} } }
```

```
// KEYS
using K_L1_id = metastring("id");
using K_L1_year = metastring("year");
using K_L1_salary = metastring("salary");
using K_L1_name = metastring("name");
using K_L1_position = metastring("position");
// TRANSFORMATIONS
using SalaryMinMaxFunctor =
    MinMaxNumberFunctor_c<1000, 1, 8000, 1, float>;
using SalaryOptions = mp_list<mp_list<
    JNumberOptions::JNumberTransformer,
    SalaryMinMaxFunctor>>;
// DICT
using BaseAction = JDict<mp_list<
    mp_list<K_L1_id,
        JNumber<uint32_t, K_L1_id>>,
    mp_list<K_L1_year,
        JNumber<uint32_t, K_L1_year>>,
    mp_list<K_L1_salary,
        JNumber<uint32_t, K_L1_salary,
        SalaryOptions>>,
    mp_list<K_L1_name,
        JStringStaticCopy<mp_int<32>, K_L1_name>>,
    mp_list<K_L1_position,
        JStringStaticCopy<mp_int<32>, K_L1_position>>
>>;
```

Figure 3: A sample JSON (top), corresponding JSON-schema with fields transformations (middle) and a fragment of a generated meta-description in C++ templates (bottom). For the simplicity only one transformation meta definition for *salary* field is shown.

The JSON structure is reflected in C++ template classes. Each major component of a JSON object (like string, number, bool, array or dictionary) is encoded as a separate template class. Template parameters provide details on the resulting value's type, properties or transformation. All currently implemented transformations are gathered in Table I. A transformation may, for example, state that given JSON value is a String containing a date and must be converted to a DateTime value. Array and dictionary description may be composed of nested classes of contained type, and thus form a tree representing the initial JSON structure (a sample is given in the bottom of Fig. 3). *JDict* object contains *JNumber* and *JString* elements. The templates are converted during the compilation stage into parser code composed of *Actions*. Each action is designed to parse a single input element in a way that makes it unaware and independent of the whole structure of a JSON object. Actions are similar to shaders in computer graphics which often execute a small program and the logic of their code does not include pipeline specific features. They define their way of communication with an external system by declaring input and outputs. This makes them maintainable and allows creating complex pipelines with small and simple pieces. The same goes with Actions, with the difference that they are not dynamically loaded when needed. Instead, they are directly compiled into a parser of a given configuration. A sequence of actions execution may vary depending on the order of keys in JSON dictionary and parent action properties.

Table I: Available read and transformation actions between JSON simple types and *Meta Parser* types.

JSON type ⇒	**⇒ Action ⇒**	**⇒ Resulting type**
Bool	ReadBool	Bool
Number	ReadInteger	Integer
	ReadRealNumber	Float
	Cut	Integer
	MinMax	Float
String	ReadString	String
	ConvertDateTime	DateTime
	HashMap / Hash	Integer
	ToLower / ToUpper	String
Number or NULL	FillNA(0)	Integer
String or NULL	FillNA("")	String

Additionally, an action instance may contain a hint to the parser, which may simplify the parsing process by specifying if the order of keys in a dictionary is fixed or random. Some transformations are combined with parsing action in order to speed up the overall process. An example of that is *Hash* which is computed for characters of a string being read (see Figure. 5). In general, transformations may be combined with any input read action if only the types conform.

C. Threads workgroup

A workgroup is a group of threads that work tightly together and may benefit from shared memory, coalesced memory accesses, prefetching, and a functionality of interchanging information within a warp to speed-up computations. We distinguish a workgroup from a warp because the number of cooperating threads may be smaller then 32 based on the needs. The size of a workgroup can be optimized to specific

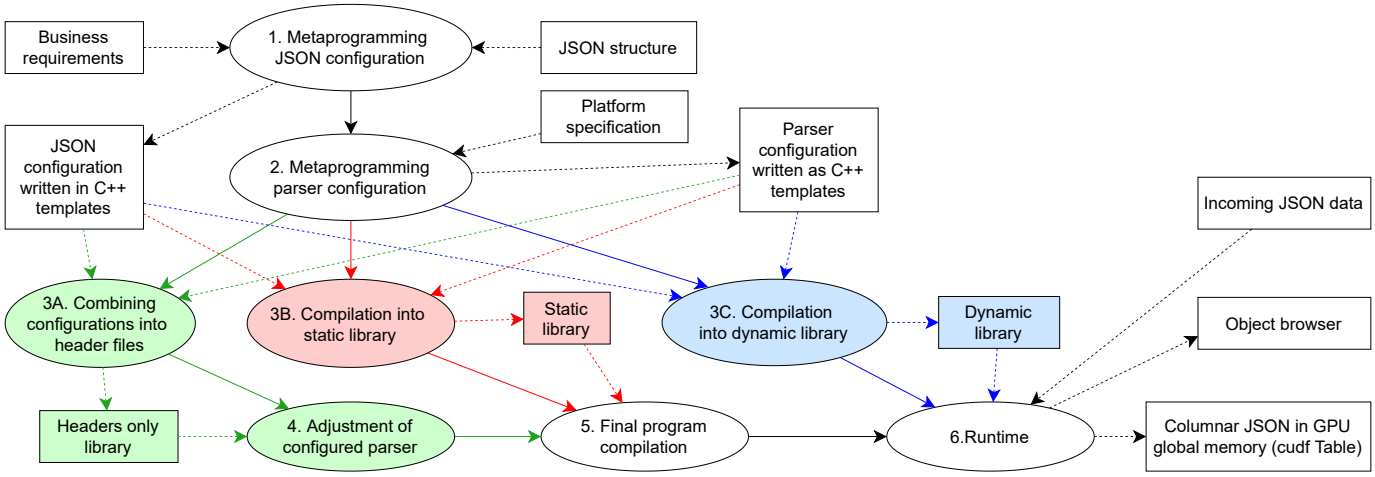


Figure 4: Activity diagram of the C++ library solution usage with three distinct paths of the *Meta Parser* utilization: A (green) – header only, B (red) – static library, C (blue) – dynamic library. Solid lines indicate transitions between subsequent steps, dotted lines show consuming or producing artifacts in the actions. Current Python helper tool covers steps from 1 through 3C to 6.

JSON characteristics and should be the subject of application profiling.

Each workgroup of N threads parses a distinct JSON object and works independently of other workgroups. Threads are organized in a way that each one by default reads a single character from an input so a workgroup has access to N consecutive characters. In order to overcome the race condition between workgroups and maintain high performance, we designed *WorkGroupReader* which is created by each thread at the beginning of a kernel. It gets pointers to input data, shared and global memory buffers and provides methods for reading characters in *Actions*. It ensures coalesced memory reads, regardless of a workgroup's size.

Each action needs some number of shared memory buffers of different sizes and types: prefilled read-only, persistent through action execution, short lived for computation only. *MetaMemoryManager* created during compilation time manages those buffers for the workgroups which is more economical than allocating separate and exclusive memory. Size of the buffers is optimized during compilation time upon particular action meta definition.

V. Evaluation

Because combining proper *cuDF* library version, CUDA drivers and compilers is complicated and fragile, for simplicity of use we created a *docker* image containing all the libraries and tools necessary for building *Meta Parser* and running it together with *cuDF*. All the source code, configurations and sample datasets are available online as Open Source software as well as thorough how-to documentation[4].

In the experiments we used a machine with the following configuration: Intel(R) Core(TM) i9-7920X CPU @ 2.90GHz (24 cores), 62 GB RAM, NVIDIA GeForce RTX 2080 Ti 11GB GDDR6 (CC: 7.5, Turing Architecture), NVIDIA driver 470.94, CUDA 11.4.

There were two groups of tests performed: *synthetic tests* using artificial data crafted to measure particular properties of the system and *end-to-end tests* on data similar to a real life system measuring all the workflow from a Python library user's points of view.

A. Synthetic tests

Synthetic tests were designed to measure efficiency and scalability of *Meta Parser* engine in comparison to *libcudf* library. The input data contained from 10 to 10^6 objects with fields of particular tested type in various configurations, for instance:

```
{"date": "2021-03-18 00:16:48"}
{"a":true,"b":false,"c":false}
{"generic": 1, "scientific": 1e0, "fixed": 1.0}
{"uint": 1024}
```

Some of the tests included type conversions, for example from different number notations to a float or to an integer, or from a string to *DateTime*, and other. All runs were repeated 50 times and a median value was taken.

1) Process scalability: The JSON lines parsing pipeline contains the following stages performed on a host (H) and on a device (D):

a. Initialization – creating structures, loading drivers, etc. (H);
b. Memory allocation and copying data to the GPU – creating memory buffers on the device side and copying data from a file to those buffers (H&D);
c. Finding new line characters to locate objects – GPU kernel which runs through data marking positions of potential object (D);
d. JSON processing – *Meta Parser* core engine created via metaprogramming for particular JSON structure (D);

[4]Available at: https://github.com/mis-wut/meta-json-parser

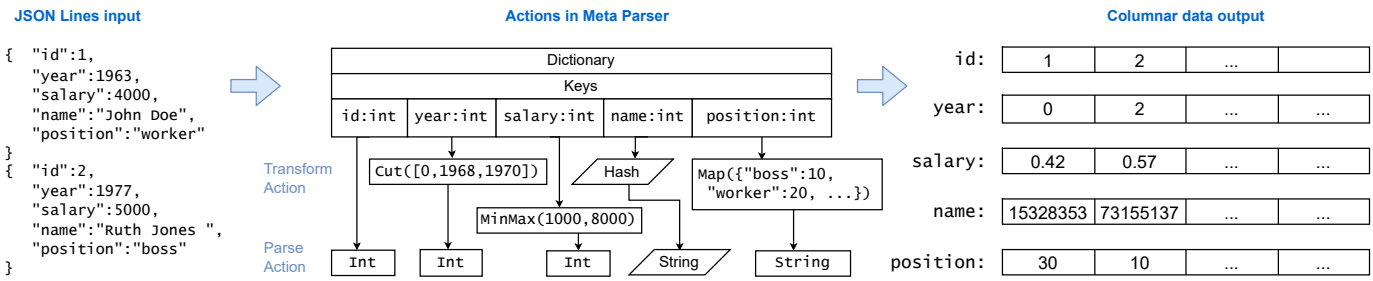


Figure 5: Conversion process from sample two JSON objects to columnar data. In the center, *Actions* nested in JDict metaobject of 5 keys in a form of a workflow diagram. On the right there is the result in columnar form which is converted to *libcudf*.

e. Converting results to *cuDF* data format – mostly performing strings compaction and several post processing pointers manipulation (D).

Fig. 6 shows breakdown of these processing stages run for different input sizes. We can easily observe that the initialization time is constant. Other processes tend to linear time for significant data input size. Typically to other GPU data intensive processing for small number of input objects we can observe constant penalty time for organization of the processing and memory data structures. For bigger inputs it disappears among longer processing time.

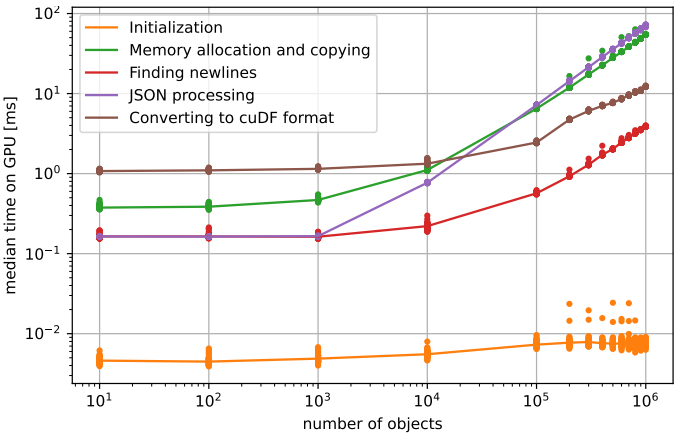


Figure 6: Scalability of the *Meta Parser* process stages. The plot presents data relative to median time of 100k objects processing. Measurements performed for 10 to 1M objects in a file.

2) Parsing elements efficiency: Single object parsing mainly depends on three factors: number of fields, length of a label string and length of a value. Fig. 7 presents time comparison of *Meta Parser* and *libcudf* parsing 10^6 objects containing from 1 to 3 bool fields. We can observe that number of fields influence the parsing time, however *Meta Parser* scales much better than *libcudf*. Parsing three fields in an object is about 2 times slower than one field while in case of *libcudf* it is about 3 times slower.

Simple type fields parsing speed is presented in Fig. 8. In case of *libcudf* there is a possibility of defining type of a parsed field with *dtype* parameter. Its usage has got positive influence on working time. Results for types with *dtype* specification are indicated by uppercase labels. *Meta Parser* has got type information all the time since it is compiled into the GPU kernel.

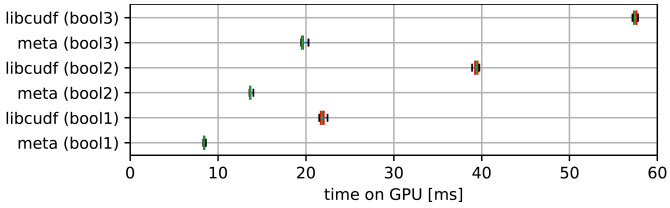


Figure 7: Comparison of Bool type objects parsing with variable number of fields (from 1 to 3). 1 million lines processed.

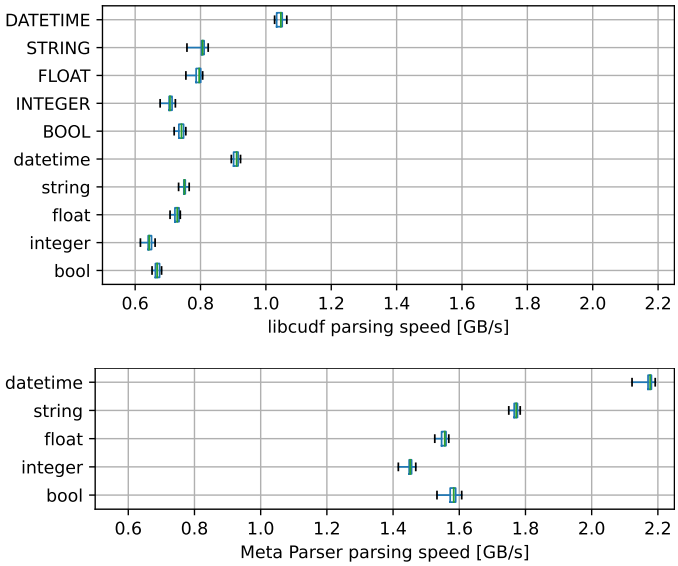


Figure 8: Comparison of parsing speed between *libcudf* and *Meta Parser*. 1M objects containing one field of given type. Uppercase labels indicate *libcudf* parser using type information.

B. End-to-end test on semi synthetic data

The purpose of the end-to-end test is to evaluate the performance of the complete solution on a real-life data.

Starting with reading the input, transferring it to the GPU, creating a columnar representation and performing initial data transformations.

We base the experiment on a real life dataset [29], exported Reddit posts in JSON lines format, which is big enough to simulate industrial requirements. The data is semi synthetic, as we cleaned it to be accepted by all parsers considered. The cleaning procedure involved removing nested objects and cleaning edge escape cases from the texts. Since *simdjson* has only experimental JSON lines support, for its tests we converted the data into classic JSON format.

We consider two processing scenarios which are common in case of ETL processes: 1) parse data and leave it in columnar format, 2) parse data and perform transformations on the values preparing the further processing.

Input data structure and transformations used are collected in Table II. There are several string, bool and integer fields totaling 20 elements, processed in each object. We also defined 8 transformations which seam reasonable in this scenario. *Meta Parser* applies the transformations during parsing, while Pandas and *cuDF* transform the data after constructing the *Data Frame*.

In this setup we know both the definition and initial transformation beforehand, thus this information can be passed to the given parser (if supported). Simdjson infers schema from JSON, Pandas and *cuDF* support schema inferring and providing the scheme directly, *Meta Parser* must be given a schema directly. For Pandas and *cuDF* we report the best performing variant, i.e. Pandas – infer schema, *cuDF*– direct schema. Only *Meta Parser* supports transformations during parsing, Pandas and *cuDF* perform transformations after parsing phase. In case of simdjson only parsing time is considered as we do not convert it into *Data Frame*-like object.[5]

Results presented in Table III and Figure 9 are medians of times of run repeated 10 times. In order to mitigate the disk storage and cache influence, we made our best to fill disk cache before the measurements (equivalent procedures for all parsers). In the table, *No Transformation* stands for scenario 1, while *Transformation* columns present results for scenario 2. Simdjson (simdj in the table) shows only pure parsing time without construction of a *Data Frame*. Consecutively, we do not perform any transformations on the data as it is left in *simdjson* internal representation.

Starting from 100k objects data sizes, *Meta Parser* dominates this benchmark independently of the transformations used. Bellow 100k objects *simdjson* performs better, but it does not construct *Data Frame* like object, and its memory representation will not allow low cost transformation to a *Data Frame* like object. It should be observed that transforming the data after parsing will add some extra time to the work flow in most of the cases. Surprisingly, if we apply transformations during the parsing it could improve the parsing time. This is due to the fact that part of the transformation reduces the final size of the stored data, which improves data transfer. This is the case of a hash function, which returns a number instead of a long string field.

If we compare the speedup between *Meta Parser* and second fastest *cuDF* 9b, we observe improvement between 3 to 1.3 times without the transformations and from 5.5 to 1.53 (note that values in table were rounded to fit into the column). The degradation of speedup with increasing input data size appears due to relatively high cost of initialization for small files in case of *cuDF*. With larger files the initialization cost does not have that much of an impact anymore.

Table II: Reddit augmented sample data structure and transformations used in the experiment: 20 fields, 8 transformations

# fields	type	transformation
2	String	Hash
2	String or NULL	FillNA and Hash
1	String	ToLower
5	String	-
3	Bool	-
1	Integer or NULL	FillNA
6	Integer	-

Table III: Time in seconds of end-to-end JSON lines parsing by various engines. (pand. – Pandas internal parser, meta – *Meta Parser*, cudf – *cuDF* parser, simdj. – *simdjson*. The table show median of time from 10 runs per each method and size of JSON file.

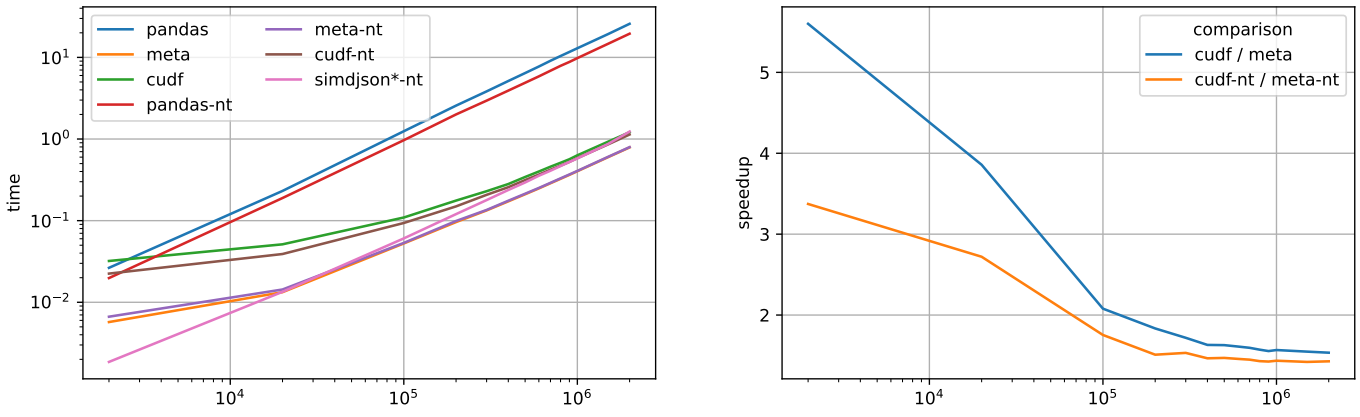
		Transformation			**No transformation**			
# obj.	**Bytes**	pand.	meta	cudf	pand.	meta	cudf	simdj.
2.0 k	1.2M	0,03	0,01	0,03	0,02	0,01	0,02	0,00
20.0 k	12M	0,23	0,01	0,05	0,19	0,01	0,04	0,01
100.0 k	58M	1,25	0,05	0,11	0,97	0,05	0,09	0,06
200.0 k	116M	2,57	0,10	0,18	2,00	0,10	0,15	0,12
300.0 k	173M	3,83	0,13	0,23	2,95	0,14	0,21	0,18
400.0 k	230M	5,14	0,17	0,28	3,92	0,18	0,26	0,24
500.0 k	287M	6,43	0,21	0,34	4,89	0,21	0,31	0,29
600.0 k	344M	7,73	0,25	0,40	5,85	0,25	0,37	0,35
700.0 k	401M	9,09	0,29	0,46	6,87	0,29	0,42	0,41
800.0 k	459M	10,39	0,33	0,51	7,88	0,33	0,47	0,47
900.0 k	516M	11,66	0,36	0,57	8,80	0,37	0,52	0,52
1.0 M	573M	12,94	0,40	0,63	9,79	0,41	0,58	0,58
1.5 M	860M	19,26	0,60	0,93	14,65	0,60	0,86	0,87
2.0 M	1.2G	25,80	0,79	1,21	19,51	0,80	1,14	1,23

VI. Discussion

A. Metaprogramming: new horizons in structural texts parsing

From the synthetic data types test we can observe that simple type values parsing is significantly faster using specialized *Meta Parser* than *libcudf*. Optimization generated by automatic code specialization saves 70% of computations. This leaves time for additional operation which may be done while parsing objects. During the end-to-end testing scenario it was proved that transformations included in the parser code does not slow down the overall process is an observable way. In the contrary, it may even speed up if transformation result consumes less space and memory bandwidth may be saved.

[5]Conversion could be done, but it was dominated by the parser time, thus we decided to report only the parsing time.



(a) Parsing time [s]

(b) Speed-up compared with fastest CuDF

Figure 9: Comparison of various JSON parsing engines. pandas – Pandas internal parser, meta – *Meta Parser*, cudf – *cuDF* parser, simdjson*-nt – *simdjson*. *nt* stands for *No Transformation* – only time needed to parse and construct a *Data Frame* is reported. Axis X: *size* represents number of JSON objects. Axis Y: time in seconds (left), speed-up ratio (right). The plots show median of 10 runs per each method and size of JSON file. There are very little differences between the measurements, so the confidence intervals are unobservable in the graph.

What we can also derive from our experiments is that *libcudf* (currently the only GPU-based JSON parser we know) parsing speed, is computation bound. Otherwise, it would not be possible to achieve faster parsing speed in *Meta Parser*, which also must have been memory bound. We can conclude that it is still possible to increase speed of structural data parsing using metaprogramming or other techniques. Metaprogramming is able to generate code better than a general solution. We can predict that together with development of *Meta Parser* capabilities and abilities to handle more and more complex data structures its domination over other solutions will increase.

B. Compilation time

Our process consists of code generation phase and code compilation phase. The C++ compiler consumes 99% of all the tool generation time. Unfortunately, due to heavy use of templates, it takes time that can be amortized if the parser is used multiple times. Thus, the scope of utilization of our solution is a specialized parser/transformer that is used in a heavy workload over extended period of time. In such scenarios gains can be significant.

Many metaprogramming-based solutions use dynamically loaded libraries cashing. If a structure of a source is already known, instead of a complete compilation, a library from a cache is reused [20]–[22]. The same solution could be used in *Meta Parser* end-to-end tool.

C. Schema guessing

During the end-to-end tests we experienced that the schema guessing is sensitive to random errors in the input stream. Even in data coming from a professional web service missing elements, wrong string encoding or field order change may happen. Therefore a specialized parser must be prepared for random changes in data, skipping wrong records, and falling back to a safe state as soon as possible. In such case *Meta Parser* may do this faster than a general parser, since an error may be recognized sooner.

Exceptions in the input data appeared to be a problem for systems which convert JSON to columnar format. Assumptions done earlier cannot hold in the later parts of the input buffer. Pandas was able to react quite well to this problem (although also made mistakes), while *cuDF* generally cannot handle sudden changes in the input stream. *Meta Parser* simply skips records which are not fulfilling the format definition.

What is important, inferring data schema from the stream may lead to degenerated *Data Frame*. Thus, forcing fixed schema and handling exceptions differently may improve the overall user experience. In the End-to-end experiment we observed anomalies in the data that lead to wrongly inferred *Data Frame* scheme in Pandas and *cuDF*, caused by few lines with unusual data structure. The possibility of treating these lines as exceptions would be the preferred route.

D. Speed-up limitations

As it was visible in the synthetic and end-to-end tests, copying the JSON data from disc/cache to RAM and then to a GPU device is a significant bottle neck. Even if the parallel parser engine itself breaks speed records, overall performance according to Amdahl's law is limited by single process-based dominant part namely memory copying. This problem touches all data intensive GPU applications and is the main driver for increasing data transfer bus bandwidth.

This can be partially overcome by concurrent data copying and computing at the same time, or by using multiple GPU cards (on single machine or cluster setup). Thanks to

Dask-cuDF integration one can work with multiple GPUs. Additionally, thanks to GPUDirect Storage [30] one can move data directly to the GPU skipping CPU, which can drastically increase throughput between storage and GPU.

VII. Conclusion and future work

The goal of the *Meta Parser* prototype development was to verify if metaprogramming can improve the parsing speed. We believe that indeed it proved its usefulness and promises successful application in many other structural text processing tasks. The next stage will be a full support of transformations nesting and developing a complete sub-language for transformations description. These transformations could also increase capability of reacting to unexpected situations in data by defining for example special cases handling methods.

The *Meta Parser* architecture is, in fact, data format agnostic, and could potentially be applied to define any textual data parser, for example CSV, some types of XML or other structural text formats used for data.

The Python wrapper tool could dynamically self-optimize and adapt its behavior depending on input data structure or properties. When processing streams of data, a background process may compile a new parser and use it when ready, test its behavior and then react.

The *Meta Parser* technique could be used to generate a query processor which will query parsed JSON data and evaluate a more complicated expression on the fly returning already processed results which are then send to a database. The capabilities of such meta query processor should be further investigated and could benefit from research presented by Müller et al. [11] in Rumble system.

References

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," Nov. 2008. [Online]. Available: https://www.w3.org/TR/xml/

[2] ECMA International, "ECMA-404. The JSON data interchange syntax," Dec. 2016. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[3] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Internet Engineering Task Force, Request for Comments RFC 8259, Dec. 2017, num Pages: 16.

[4] M. Scheidgen, S. Efftinge, and F. Marticke, "Metamodeling vs Metaprogramming: A Case Study on Developing Client Libraries for REST APIs," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Wąsowski and H. Lönn, Eds. Cham: Springer International Publishing, 2016, pp. 205–216.

[5] I. Ward, "JSON Lines. Documentation for the JSON Lines text file format," 2021. [Online]. Available: https://jsonlines.org/

[6] T. Hoeger, C. Dew, F. Pauls, and J. Wilson, "Newline Delimited JSON Specification." [Online]. Available: http://ndjson.org/

[7] D. Bonetta and M. Brantner, "FAD.js: fast JSON data access using JIT-based speculative optimizations," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1778–1789, Aug. 2017.

[8] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki, "DBMS Data Loading: An Analysis on Modern Hardware," in *Data Management on New Hardware*, ser. Lecture Notes in Computer Science, S. Blanas, R. Bordawekar, T. Lahiri, J. Levandoski, and A. Pavlo, Eds. Cham: Springer International Publishing, 2017, pp. 95–117.

[9] G. Langdale and D. Lemire, "Parsing Gigabytes of JSON per Second," *VLDB Journal*, vol. 28, no. 6, pp. 941–960, Feb. 2019.

[10] R. D. Team, "RAPIDS: Collection of Libraries for End to End GPU Data Science," 2018. [Online]. Available: https://rapids.ai

[11] I. Müller, G. Fourny, S. Irimescu, C. B. Cikis, and G. Alonso, "Rumble: data independence for large messy data sets," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 498–506, Dec. 2020.

[12] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, "Filter before you parse: faster analytics on raw data with sparser," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1576–1589, Jul. 2018.

[13] D. Xie, B. Chandramouli, Y. Li, and D. Kossmann, "FishStore: Faster Ingestion with Subset Hashing," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1711–1728.

[14] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "NoDB in action: adaptive query processing on raw data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1942–1945, Aug. 2012.

[15] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: a fast JSON parser for data analytics," *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1118–1129, Jun. 2017.

[16] W. McKinney, *Python for Data Analysis*. Sebastopol, California: O'Reilly, 2014.

[17] S. Rabhi, W. Sun, J. Perez, M. R. B. Kristensen, J. Liu, and E. Oldridge, "Accelerating recommender system training 15x with RAPIDS," in *Proceedings of the Workshop on ACM Recommender Systems Challenge*, ser. RecSys Challenge '19. New York, NY, USA: Association for Computing Machinery, Sep. 2019.

[18] S. Raschka, J. Patterson, and C. Nolet, "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence," *Information*, vol. 11, no. 4, p. 193, Apr. 2020, number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[19] D. D. Chandar, J. Sitaraman, and D. Mavriplis, "CU++ET: An Object Oriented Tool for Accelerating Computational Fluid Dynamics Codes using Graphical Processing Units," in *20th AIAA Computational Fluid Dynamics Conference 2011*. American Institute of Aeronautics and Astronautics Inc., 2011.

[20] F. Winter, "Accelerating QDP++/Chroma on GPUs," *PoS LATTICE2011*, 2011.

[21] J. Chen, B. Joo, W. Watson, and R. Edwards, "Automatic offloading C++ expression templates to CUDA enabled GPUs," *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pp. 2359–2368, 2012.

[22] D. Demidov, "VexCL: Vector expression template library for OpenCL - CodeProject," 2013. [Online]. Available: https://www.codeproject.com/Articles/415058/VexCL-Vector-expression-template-library-for-OpenC

[23] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling, "Programming CUDA and OpenCL: A case study using modern C++ libraries," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, sep 2013.

[24] K. Ahnert, D. Demidov, and M. Mulansky, "Solving ordinary differential equations on GPUs," in *Numerical Computations with GPUs*, V. Kindratenko, Ed. Springer International Publishing, jan 2014, pp. 125–157.

[25] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[26] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating java programs with CUDA," in *LNCS (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, H. Sips, D. Epema, and H.-X. Lin, Eds., vol. 5704 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 887–899, iSSN: 03029743.

[27] "Numba: A High Performance Python Compiler." [Online]. Available: https://numba.pydata.org/

[28] A. O'Dwyer, "Comparative TMP #1: MPL, Mp11, Kvasir, Hana, Metal." [Online]. Available: https://quuxplusone.github.io/blog/2019/12/28/metaprogramming-n-ways/

[29] J. Baumgartner, S. Zannettou, B. Keegan, M. Squire, and J. Blackburn, "The Pushshift Reddit Dataset," vol. 14, pp. 830–839, 2020. [Online]. Available: https://ojs.aaai.org/index.php/ICWSM/article/view/7347

[30] A. Thompson and C. J. Newburn, "Gpudirect storage: A direct path between storage and gpu memory," *NVIDIA Developer Whitepapers*, vol. 8, 2019.