SLAC NATIONAL ACCELERATOR LABORATORY

LCLS2 Data Formatting

Hayden Blair[1], Clemens Weninger[2], Paul Christopher O'Grady[2,+]

[1]LCLS Intern

[2]Linac Coherent Light Source, SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA 94025, USA.

[+]Contact: cpo@slac.stanford.edu

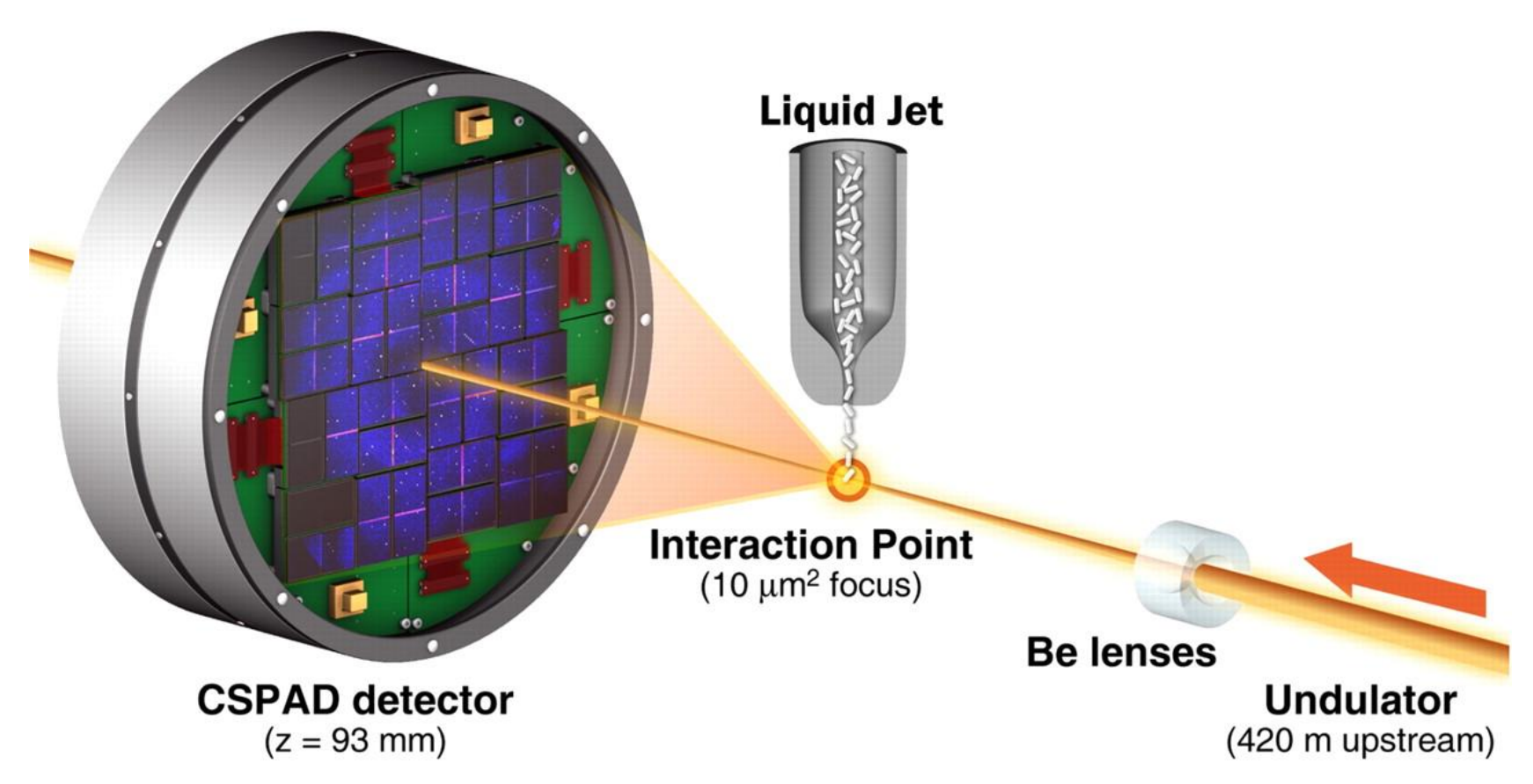
Introduction

The LCLS currently shoots 120 photon packets every second. The LCLS-II will be 8,000 times faster, shooting up to a million pulses per second. Each pulse produces a large memory block filled with data, called a Datagram. Each Datagram must be parsed, analyzed, and then stored. All of this must happen in under one millionth of a second. I have written optimized C code that will accept the Datagram, build a unique Python type object for it, and then distribute its components into that object. This object will then be used in further analytical processes, which will be carried out in Python.

Keywords: Pulse, Datagram, Analysis

Research

Fig 1: Beam Visual



This is a depiction of how the LCLS procures data. The beam will hit its target at the Interaction Point, and then the result will be captured by the sensor array placed behind it. The depicted CSPAD is one of many detectors whose outputs must be accounted for during data parsing.
*Image courtesy of LCLS-II homepage webmaster

The goal of this project was to properly deliver the beam information to the data analysis routines. These routines will be executed in Python, because their implementation is much easier than it would be in C. However, Python is inefficient with memory management and cannot parse a million datagrams per second. This part of the process needs to be handled by C.

The Datagram:

Every datagram is segmented, with each segment representing the output of one sensor in the array. The leading bytes of each segment describe the names, data types, and positions in memory of all the output data from that sensor. **Figure 2** is a visualization of this structure.

Fig 2: Datagram

```
{(descriptor: "sensor temp", float, offset=0;   "position",
int[2], offset=4) [data: 53.9514;[2,5]]},
{(descriptor: "incident angle", float, offset=12) [data:
33.26]}, {(descriptor: …) [data: …]},
{(descriptor: …) [data: …]},
…
```

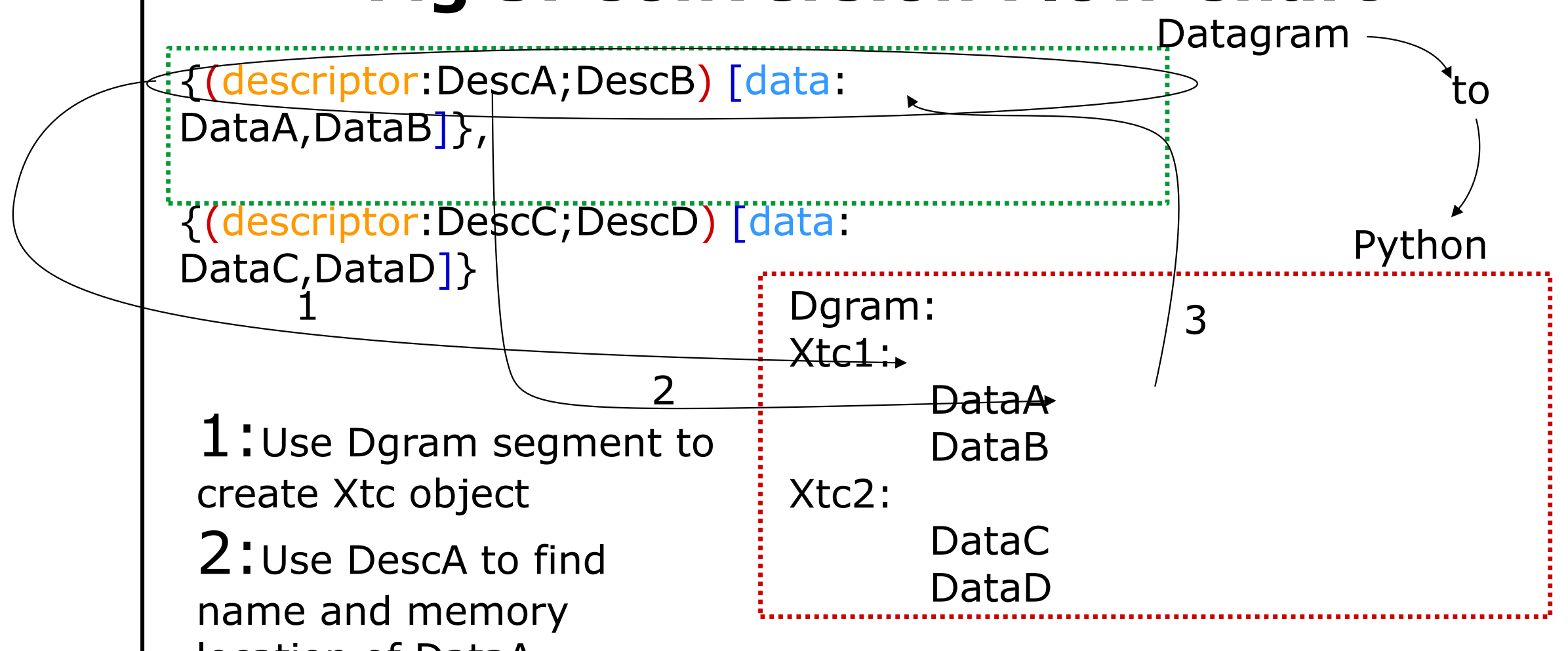
Each entry in "data" can be either a single value, or an array, of any fundamental data type (integer, float, double). The "offset" indicates the size (in bytes) of all the "data" entries before the one currently being described. No matter where in memory the Datagram is saved, we can find the location of any "data" member by adding its offset to the Datagram's starting memory address.

The Python Conversion:

In order to make the Datagram's structure intuitive and easy to navigate, I created two new python objects to store its information.

Each individual sensor output in the Datagram is known as an Xtc. The first python object I created acts as an information retriever for a *single* sensor segment, so I also called it "Xtc". This Xtc object stores the memory locations and names of the output data for its specified detector. These names and pointers are created using the information contained in the Datagram's Descriptor. This is depicted in **Figure 3**. The Xtc's data names can be accessed by implementing its "keys()" function. It also calls a unique constructor when the user tries to access an array from the Datagram; allocating large chunks of memory takes a non-negligible amount of time, so once a sizable array is stored at a certain location, for performance's sake, it should not move. When a user calls an array for retrieval, the Xtc python object identifies its memory positions, and returns a Numpy array whose entries point back to those positions. This way, the previously allocated data will not be duplicated.

Fig 3: Conversion Flow Chart


Datagram
to
Python
{(descriptor:DescA;DescB) [data: DataA,DataB]},
{(descriptor:DescC;DescD) [data: DataC,DataD]}
1
2
3
Dgram:
Xtc1:
DataA
DataB
Xtc2:
DataC
DataD

1: Use Dgram segment to create Xtc object

2: Use DescA to find name and memory location of DataA

3: DataA in Xtc points to original DataA memory address in Dgram

The second python object is meant as an ease of access tool and allows the user to quickly maneuver all of the information in the Datagram. Since it holds every piece of information we want to be able to use, I called the object "Dgram". It contains a dictionary that can be accessed using the tab complete routine in IPython.
Each value in the dictionary is an Xtc python object, with the corresponding key being the name of the sensor assigned to that Xtc. When a user presses "Tab" on a Dgram object instance in IPython, a list of every sensor contained in that Dgram is printed to the screen. The user can then cycle through all of sensors by repeatedly pressing "Tab". This utility combined with the "keys()" method contained in the Xtc python object creates an efficient data management routine. The user can create a Dgram instance, tab complete to the desired sensor, print a list of its components, and then use the Xtc object's predefined, custom retrieval functions to access their desired data member.

Complications and Troubleshooting

One of the first challenges we faced was figuring out how long to store the data from the laser in memory. Each new pulse produced a new Datagram from the sensors, but analysis requires comparing data across multiple shots. We couldn't simply store every single Datagram, because of memory access restrictions. Solving this issue required the use of a Python tool known as "reference counting". When a Datagram is first created, it has a reference count of 1. In order to access the correct information from an array in that Dgram, the memory that contains that array must not be altered. When a user accesses an array, the reference count of the Dgram is increased, and is subsequently decreased when that array is no longer in use. The Datagram's memory is then deallocated when its reference count reaches 0.
The biggest issue this program currently faces is data inflation. The output of a single pixel from the sensor array is 16 bits long. The first 2 bits describe the gain the pixel was set at during bombardment, and the last 14 hold its received information. In order to access the data as quickly as possible, the gain needs to be stored separately. However, the smallest, store-able unit is 8 bits, meaning the original 16 bits must become 24. This is a 50% data increase the LCLS-II cannot afford.

Conclusions

My work has helped lay foundations for future data handling at the LCLS-II. The Python extensions I have written will need to be updated and changed as the LCLS-II develops, but the fundamental data structure and reference counted memory management are solid. The data inflation issue is a good example of the unforeseen complications that will keep popping up as development continues. These complications can necessitate moderate or even dramatic change in the software, and future efforts in LCLS-II data formatting must be able to keep up with the changing needs of this project.

Acknowledgments

The successes of this project are due in large part to the efforts of the faculty that guided me. I am very grateful to Chris O'Grady and Clemens Weninger for all of the clear, deeply beneficial advice they provided, and thank them both for the opportunity to be a part of this effort.

Date: 08/07/2017