JOINT INSTITUTE FOR NUCLEAR RESEARCH
DZHELEPOV LABORATORY OF NUCLEAR PROBLEMS

FINAL REPORT ON THE

SUMMER STUDENT PROGRAM

GNA framework: Implementation of GPU-based transformations

Supervisor:
Anna Fatkina

Student:
Maksim Abramovich

Participation period:
August 5 — September 29

Dubna, 2018

Contents

1 Introduction

Scientific and physical experiments are often based on acquisition of huge amount of data. Data processing and analysis performing in computers of high computational power. However, processing of such amount of data and mathematical modeling based on this data is still time and memory consuming. For example, Monte Carlo based methods typically take weeks or months on modern CPUs. There are special tools that optimize computations of such tasks. One of such tools is CUDA. It allows to use the power of GPUs on general-purpose tasks.

During SSP I joined to the development team of GNA framework. I was working on porting some CPU-oriented transformations to GPU.

1.1 GNA

GNA (Glabal Neutrino Analysis) – flexible, extensible framework for the data analysis of neutrino experiments. There are the following goals of GNA [1]:

- Developing a framework that enables a user to build comprehensive physical models and perform statistical analysis of this models with large number of parameters. Efficient, but flexible.
- Implementing the analyses for the Daya Bay [2] and JUNO [3] experiments.
- Implementing the global analysis of the neutrino data (experiments: Daya Bay, JUNO, NOvA, T2K, etc.).

The framework is an attempt to implement the following general principles [4]:

- The whole structure should be flexible enough to uniformly integrate arbitrary number of any kind of experiments into the one common flow;
- There should be granularity between analysis configuration step which is done once, and computations repeated multiple times during fits after the configuration;
- It should be possible to modify an existing computation chain to transform or completely replace any of its parts (formulas, tables, etc) in one place without changes over the whole code base.

The way to bring these principles into action is to introduce a number of simple independent computational blocks representing all the inputs or mathematical operations required to build a theoretical model of any experiment. The task of the user (analyzer) is to use those blocks as ingredients to construct a computational graph (fig. 1) producing the theoretical predictions and finally the desirable statistic.

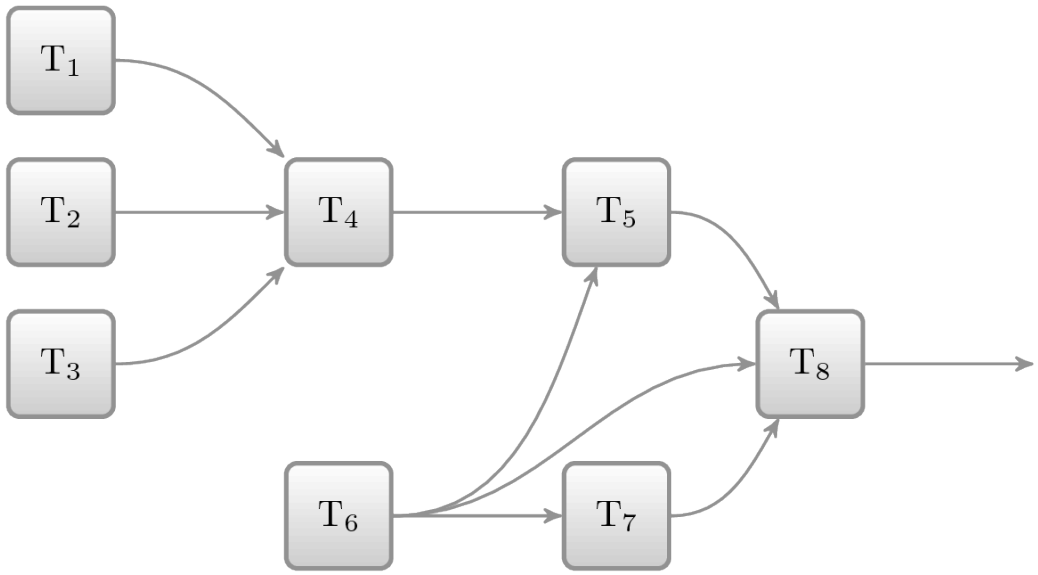


Figure 1: Schematic example of GNA graph

Since the blocks are small, simple and independent, they may be easily implemented in a relatively low-level language (namely C++) making all the repeating computations fast, while all the relations between them may be expressed by means of a slower but dynamic language (namely Python), leading to great flexibility. Since the whole computation flow may be traced before the computations start, it is possible to group the same computations with different inputs into one vectorized procedure. The block structure also makes possible to track with high granularity the changes of computations depending on variable inputs, potentially avoiding useless recomputations during the fit.

The computational core of GNA is transformation. Transformation is an encapsulated function, basic component of computations. It has the following specification (fig. 2):

- May have zero or more inputs and has at least one output (arrays).

- May depend on parameters (variables).
- Data container is associated with each transformation output.
- Transformation is computed once and the result may be reused.
- Transformation has taint flag. It is recomputed in case of it was tainted only. Taint flag is true when data is not up-to-date.
- Computations are executed only in case the output value of corresponding transformation is used.

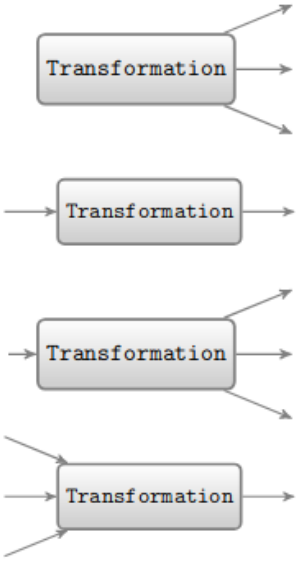


Figure 2: Example of transformation kinds

My work was directly related to transformations implementation. The main task during JINR SSP was implementation of GPU-based transformations. Porting computation from CPU to GPU should significantly accelerate framework performance. GPU support in GNA was announced this year [5].

1.2 CUDA

CUDA [6] is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

What is the difference between a CPU and a GPU? A modern CPU has less than one hundred processing cores clocked from 1 to 4 GHz. A CPU is powerful because it can do everything. If a computer is capable of accomplishing a task, that is because the CPU can do it. Programmers achieve this through broad instruction sets and long feature lists shared by all CPUs.

A GPU (graphics processing unit) is a specialized type of microprocessor. Its optimized to display graphics and do very specific computational tasks. It runs at a lower clock speed than a CPU but has many times the number of processing cores. Video rendering is all about doing simple mathematical operations over and over again, and thats what a GPU is best at. GPUs have thousands of processing cores running simultaneously. Each core though slower than a CPU core, is tuned to be especially efficient at the basic mathematical operations. This massive parallelism is what makes GPUs so powerful (fig. 3).

If a CPU is a Leatherman, a GPU is a very sharp knife. You cant tighten a hex bolt with a knife, but you can definitely cut some stuff. A GPU can only do a fraction of the many operations a CPU does, but it does so with incredible speed. A GPU uses hundreds of cores to make time-sensitive calculations for huge amount of data. However, as fast as a GPU can go, it can only really perform dumb operations. For example, a modern GPU like the Nvidia GTX 1080 has 2560 shader cores. Thanks to those cores, it can execute 2560 instructions, or operations, during one clock cycle. By comparison, a four-core Intel i5 CPU can only execute four simultaneous instructions per clock cycle. However, CPUs are more flexible than GPUs. CPUs have a larger instruction set, so they can perform a wider range of tasks. CPUs also run at higher maximum clock speeds and are capable of managing the input and output of all of a computers components. For example, CPUs can organize and integrate with virtual memory, which is essential for running a modern operating system. Thats not something a GPU can accomplish.

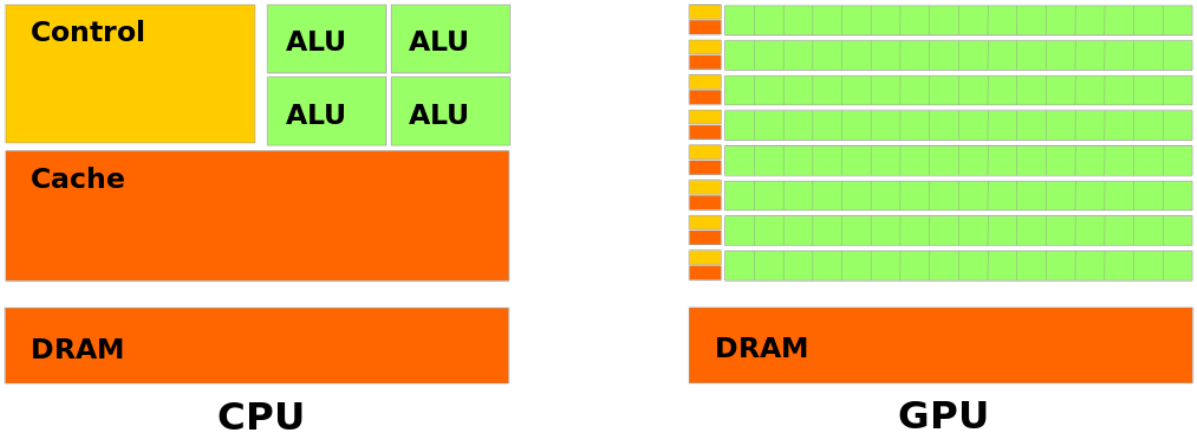


Figure 3: The difference between a CPU and a GPU architecture

CPUs and GPUs have similar purposes but are optimized for different computing tasks. Even though GPUs are best at video rendering, they are technically capable of doing more. Graphics processing is only one kind of repetitive and highly-parallel computing task. Tasks like experiments results processing and statistical analysis rely on the same kinds of massive data sets and simple mathematical operations. Operating with matrices and vectors and a lot of linear algebra operations that's why GNA is suitable to GPU porting.

But there are additional requirements to transformation GPU porting. Not all transformations are suitable because of their complexity and the presence of dependencies between data parts that's calls for time-consuming data synchronization. The other problem is data transfer. Initially allocated on a Host (CPU and RAM) data must be transferred to Device (GPU) for processing on it. After computations data must be transferred to Host for their usage. Data transfers on each transformation are very expensive and this will not lead to performance enhancement. Therefore it is necessary to minimize data transfers in computation chain and build as long GPU-based chains as it possible (fig. 4). Final task is building full GPU-based computation graphs with input data transfer to Device at the start and output data transfer to Host at the end of computation. This will lead to better performance enhancement.

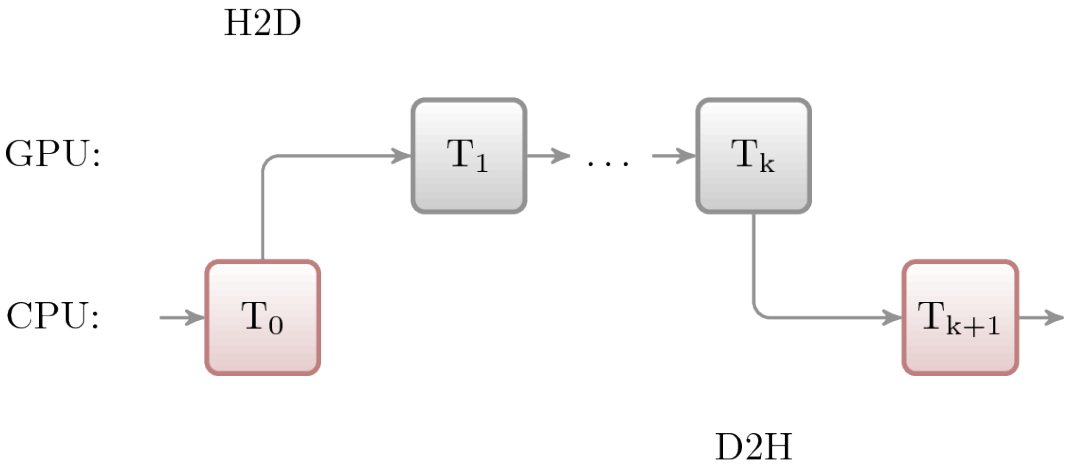


Figure 4: Example of data transfer from Host to Device (H2D) and from Device to Host (D2H) in GNA computation graph

2 GPU-based transformations

2.1 FillLike transformation

That was the first ported transformation. FillLike fills all the input array elements with given value. It has single output — array, filled with value. This value transformation get as an argument. Transformation realization is not so complicated. Porting was implemented using wrapper for the GPU array where defined several frequently used mathematical and basic operations, including fill array with value. Main task there was to embed GpuArray function in existing GPU integration scheme in this way to get acquainted with framework and general transformation realization principle. This transformation is perfect match to be calculated in parallel because it doesn't need any data synchronization.

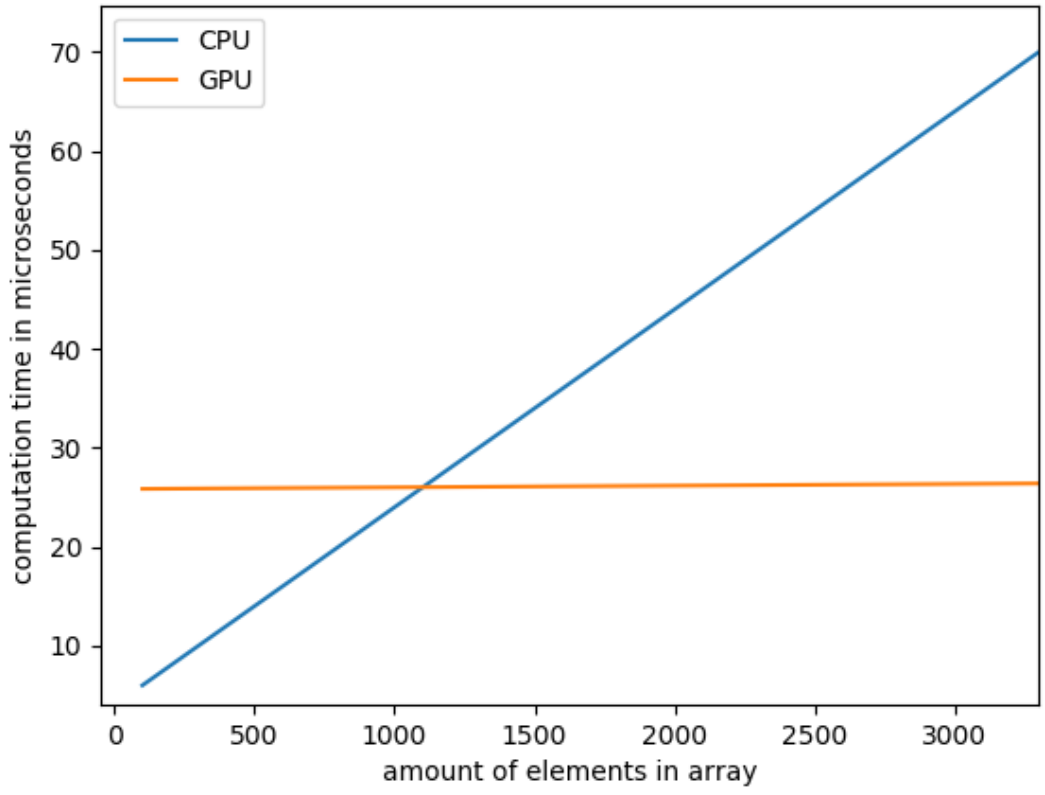


Figure 5: FillLike transformation performance comparison

As we can see in fig. 5 GPU-based transformation gives us speed up with about 1000 element array size. It can be seen that CPU computation time is growing with data size while GPU

computation time is remains constant which is typical for a GPU computations because of distribution of data between processes. We can see the same picture in fig. 6 with larger volume of data.

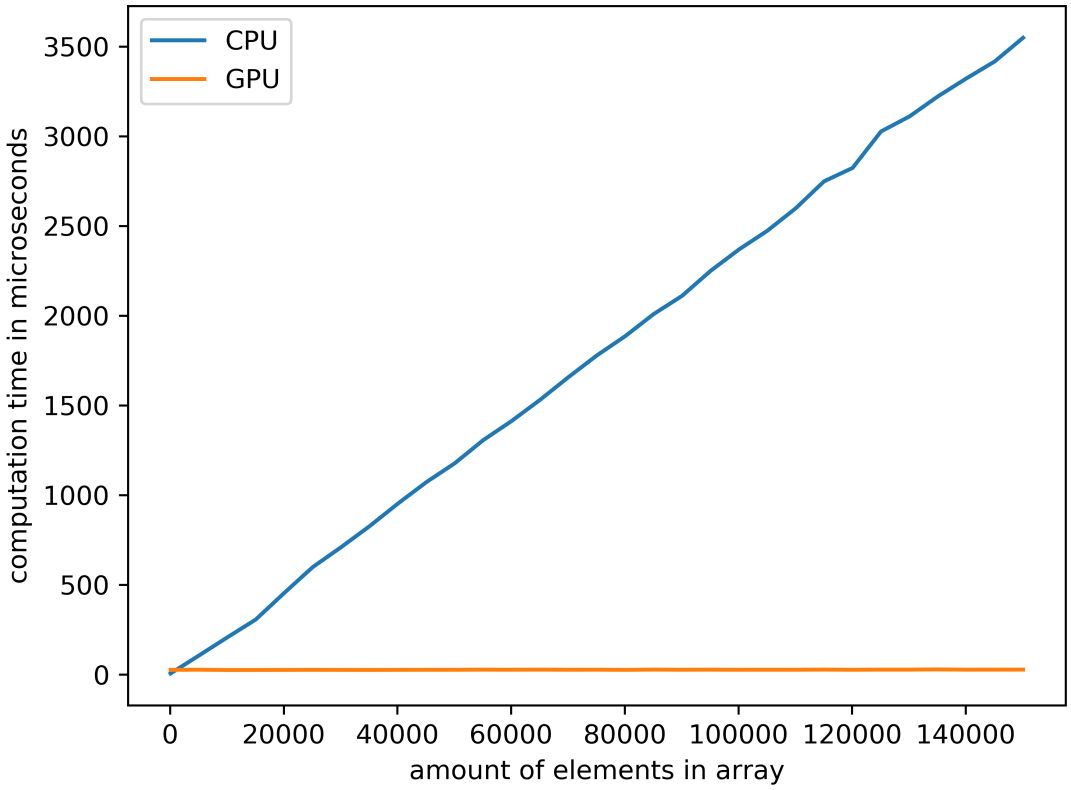


Figure 6: FillLike transformation performance comparison with larger volume of data

2.2 Derivative transformation

This transformation calculates the derivative of the multidimensional function versus parameter. Uses finite differences method of fourth order. Transformation takes two arguments: parameter instance and finite difference step h. Input and output have the same representation: array of size N.

Finite difference method approximate a derivative to an arbitrary order of accuracy with finite difference. The second order finite difference reads as follows:

$$D_2(h) = \frac{f(x+h) \quad f(x \quad h)}{2h}:$$

Due to the need of high accuracy in calculations there are used finite differences of fourth order that is reads as follows:

$$\frac{dy}{dx} = D_4(h) = \frac{1}{3}\ 4D_2\ \frac{h}{2}\qquad D_2(h) =$$
$$= \frac{4}{3h}\ f\ x + \frac{h}{2}\qquad f\ x\ \frac{h}{2}\qquad \frac{1}{6h}(f(x+h)\quad f(x\quad h)):$$

Initially this transformation was ported using implemented GpuArray functionality. The following operations were used: multiplication by a constant, addition and subtraction of vectors. But this approach needed additional buffer GpuArray. Memory allocation on GPU during computation slow down it significantly. Current memory management system can't eliminate this drawback. Therefore expected acceleration wasn't achieved (fig. 7). Additional buffer memory allocation will be moved to the stage of computation graph construction in the future that is will give us needed acceleration.

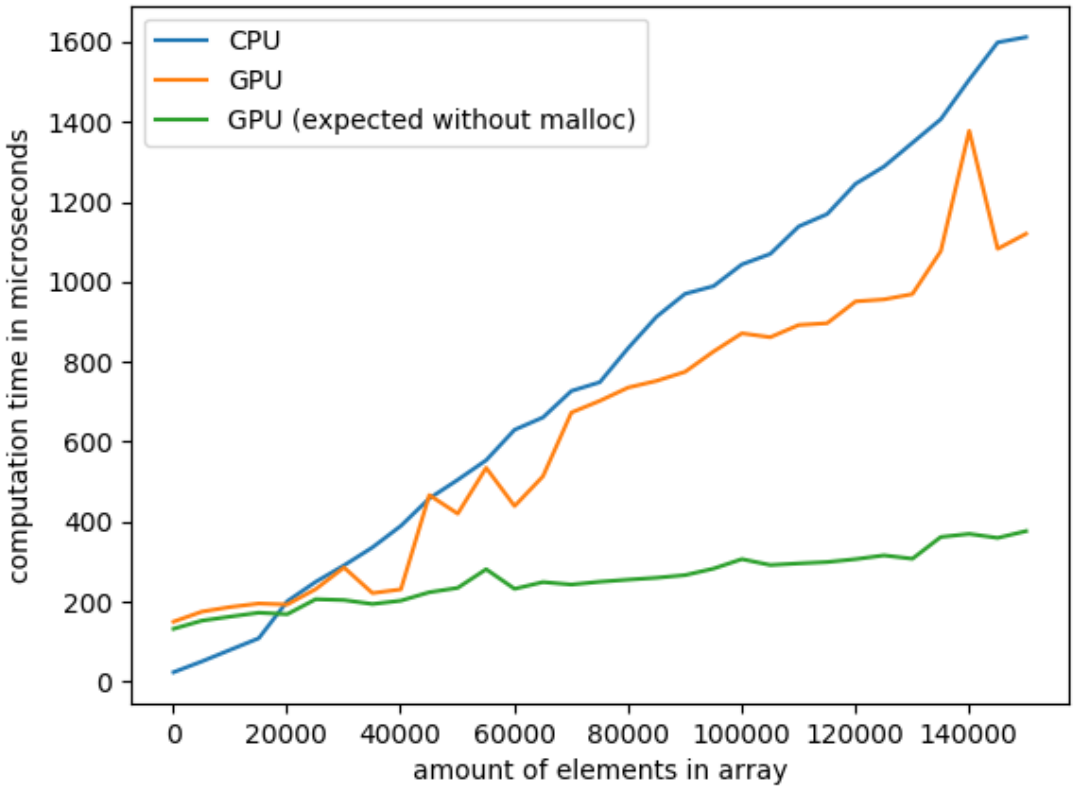


Figure 7: Derivative transformation achieved and needed performance

Second realization of GPU part of transformation was implemented on a single direct kernel call that is performs calculations according to the formula. This method need additional buffer memory allocation as well as the previous one. The difference is the size of allocated memory: this method require four times more memory because it is perform all calculation at once whereas first method perform calculation iteratively with preprocessing of temporary data on each step. This drawback can be resolved by upgrading memory management system in the future. However, this realization did not bring the expected results.

Both realizations show almost the same efficiency in terms of net computation that can be seen in fig. 8. Plot become linear with small coefficient that is caused by preprocessing.

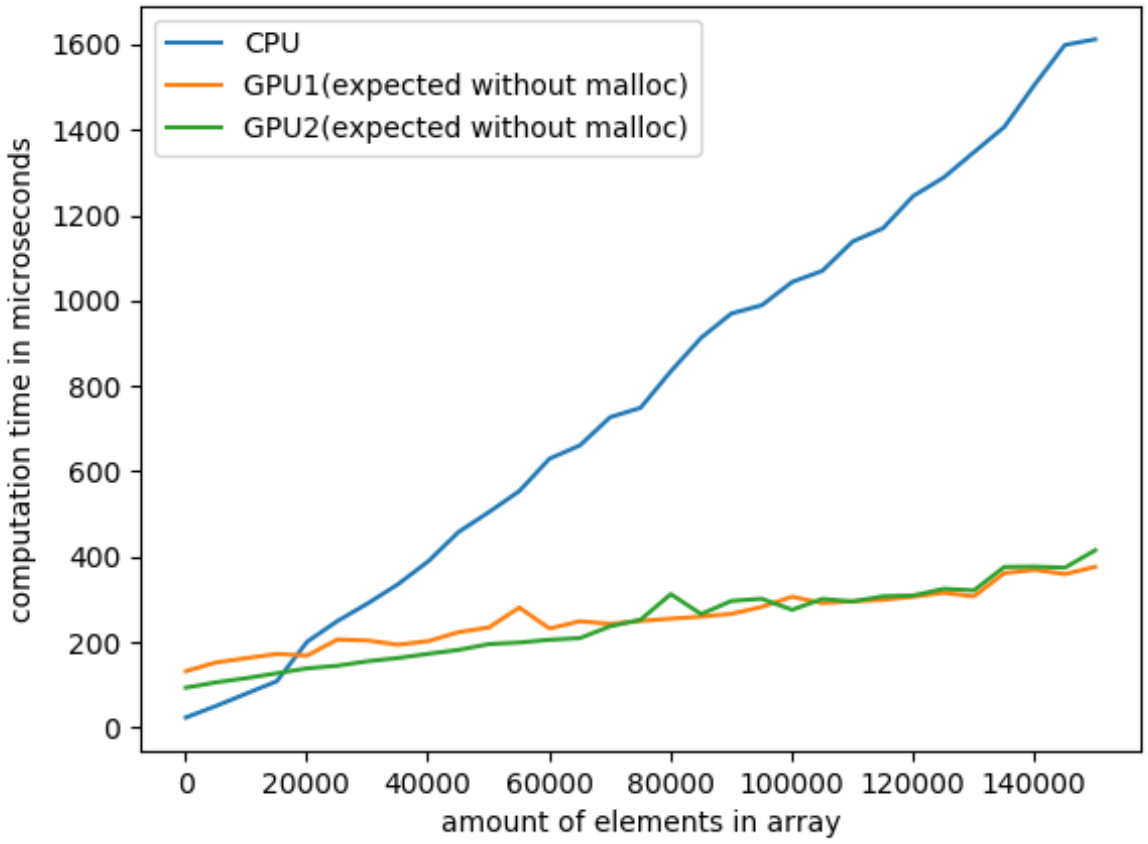


Figure 8: Expected performance of different Derivative transformation realizations

Usage of second realization become wholly unwarranted. Even though in smallest amount of data it seems to be a little more efficient, they are the same in general. In terms of current

memory management system actual difference in computation time is significant (fig. 9). As was indicated earlier more memory is required which is essential.

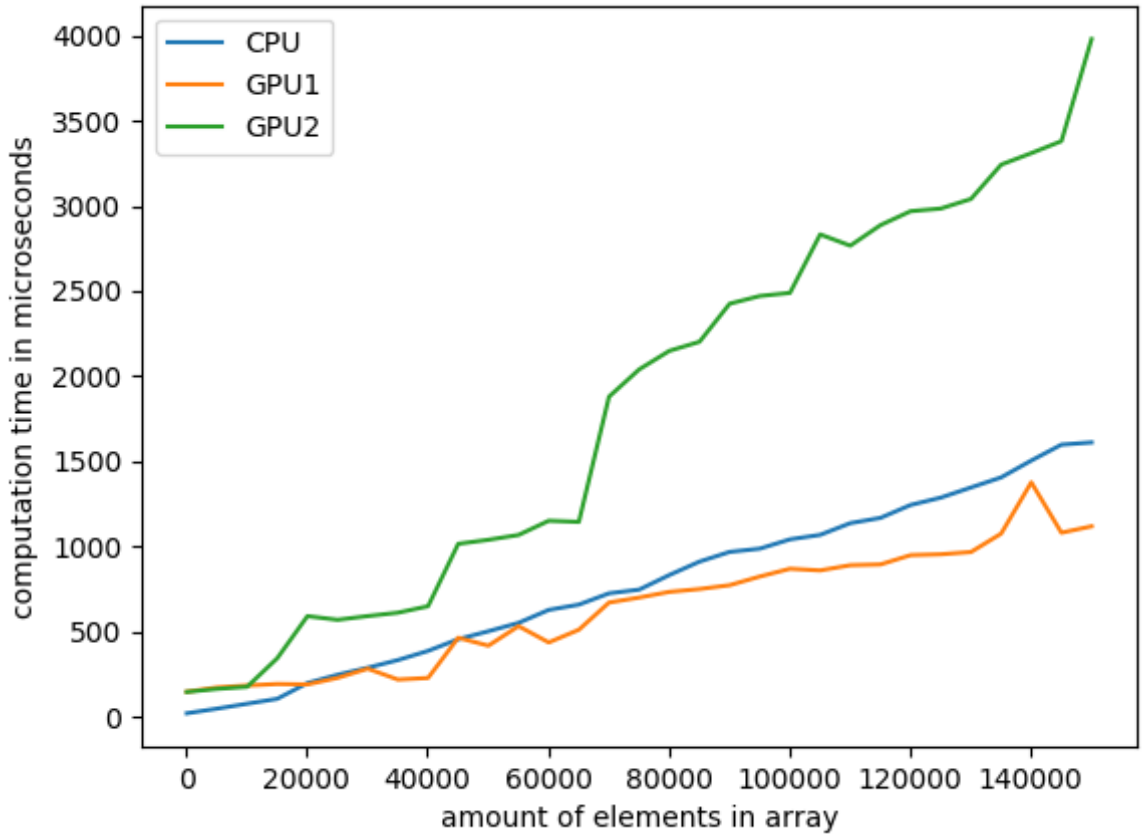


Figure 9: Actual performance of different Derivative transformation realizations

2.3 Rebin transformation

This transformation performs the histogram rebinning. The rebinning is implemented via multiplication by a sparse matrix. The transformation takes a histogram as an input, defines input edges, and produces output as histogram with new binning. Thansformation takes new bin edges as arguments. That is an original implementation leading to significant flexibility loss. So bin edges cannot be preprocessed and transferred to Device at the stage of computation graph construction for this reason. Data transfer at the computation stage is costly and it cannot be bypassed in current transformation realization. Also matrix fill algorithm is running step by step

using in current step data from previous so it cannot be calculated in parallel efficiently because it's need costly synchronization in each iteration or overlapping of computation. That's why this transformation can't be ported in full. But these computations are not primary. They will be run only once for the model evaluation. Multiplication by a sparse matrix is more important as it occurs at each evaluation.

Matrix multiplication is trivial task for parallel computation but not the easiest one [7]. In this particular case matrix multiplied by vector which is performing in this transformation. Native algorithm can be calculated in parallel with one thread per each vector element. Threads will perform multiplication matrix row by given vector operations. Results of such approach can be seen in fig. 10. So it gives performance efficiency with 2000 bin edges and more. But such amount of data don't correspond to actual tasks demand.

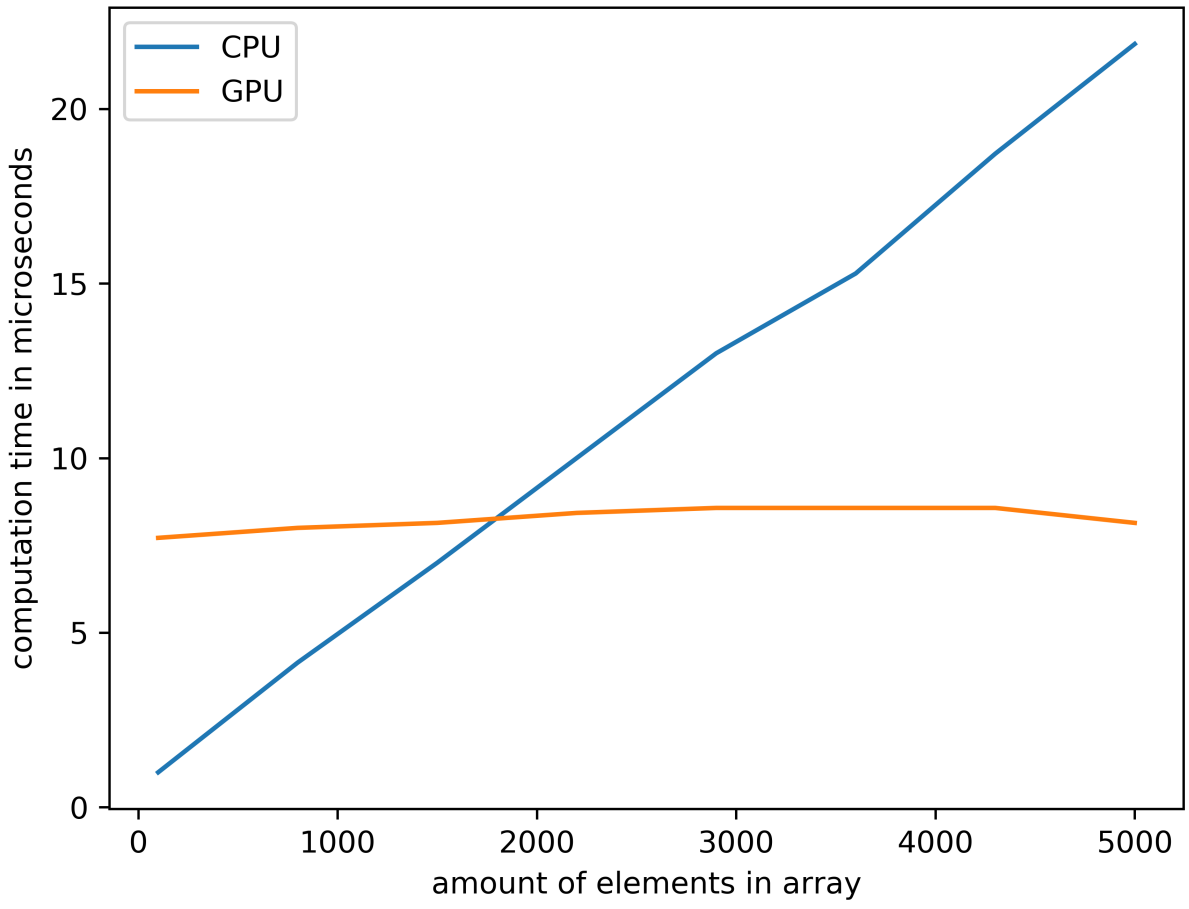


Figure 10: Performance of Smear calculation in Rebin transformation

If map threads for each matrix elements this will make it necessary to use synchronization in result recording. This approach is not efficient. The solution would be to use shared memory [8]. The main idea is to use shared memory block per block of threads. The result can be computed by parts and summarized at the end of computation. However, this not lead to performance enhance (fig. 11). Computation time became mostly the same. This is due to necessity of synchronization between threads that share memory. To summarize the result, shared memory usage accelerate computation but synchronization slowdown it more.

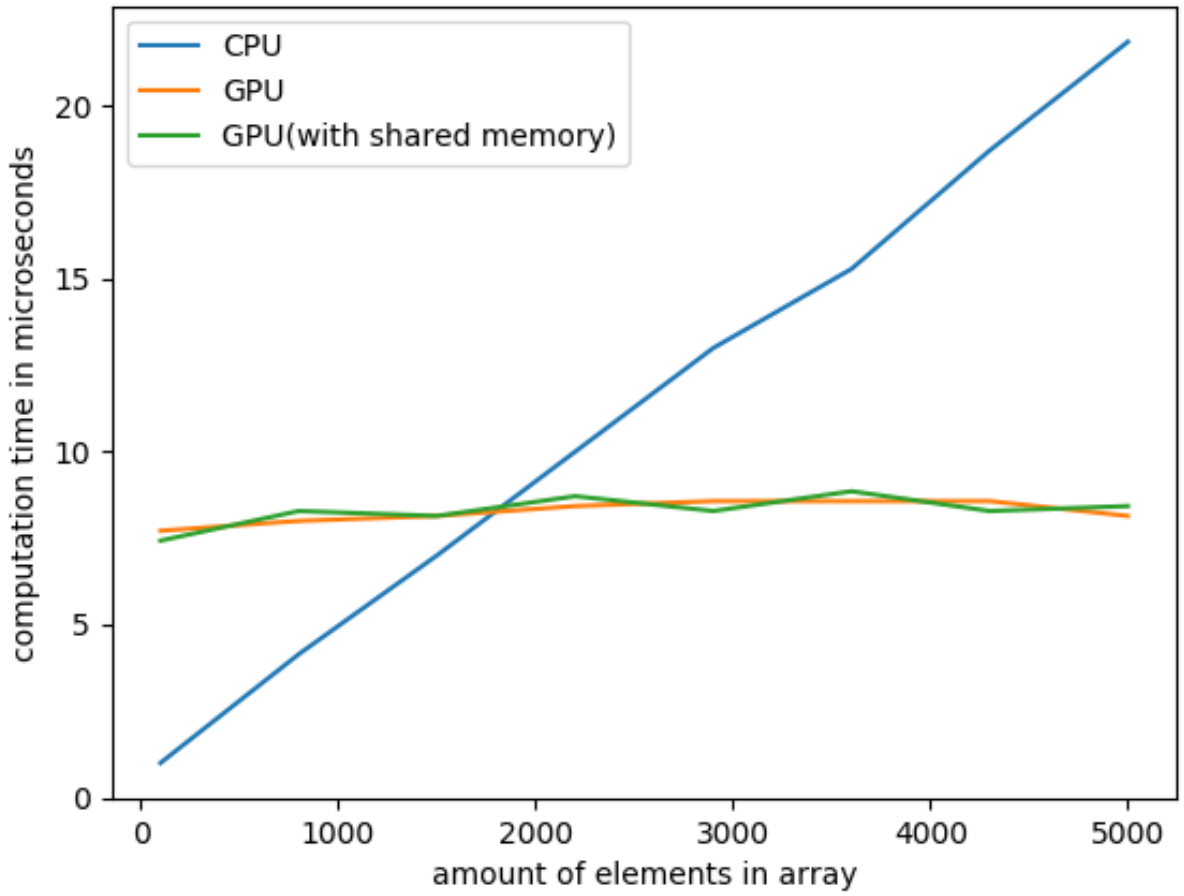


Figure 11: Performance of Smear calculation using shared memory

2.4 EnergyResolution transformation

It applies energy resolution to the histogram of events binned in E_{vis}. Transformation single input is one-dimensional histogram of number of events N_{vis} and single output is one-dimensional smeared histo of number of events N_{rec}.

The smeared histo N_{rec} and true N_{vis} are connected through a matrix transformation:

$$N_i^{\text{rec}} = \sum_j V_{ij}^{\text{res}} N_j^{\text{vis}};$$

where N_i^{rec} is a reconstructed number of events in a i-th bin, N_j^{vis} is a true number of events in a j-th bin and V_{ij}^{res} is a probability for events to flow from j-th to i bin.

That probability is given by:

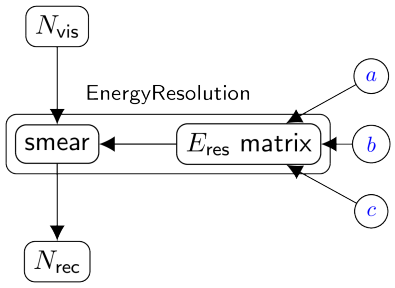
$$V_{ij}^{\text{res}} = \frac{1}{\sqrt{2}\ (E_j)} \exp\left[\frac{(E_j - E_i)^2}{2\ ^2(E_j)} \right];$$

where (E_j) is:

$$(E_j) = E_j \sqrt{a^2 + \frac{b^2}{E_j} + \left(\frac{c}{E_j}\right)^2}$$

where a, b, c are resolution parameters.

Energy resolution bundle scheme is the following:



Matrix fill might be perfect match for parallel computation because of effective exp function calculation on GPU and each probability V_{ij}^{res} is calculated singly that maximize the number of threads. But specific transformation implementation makes it impossible because computation is being performed in callback when GPU memory is not allocated yet. It also runs at the initial

stage of computation and recalculating with data addition. So it's not so impact on general performance.

Smear calculation in this transformation takes the same place as in the previous one. But there are bigger, wider and less sparse matrices that's why performance difference more significant (fig. 12). GPU accelerate computation almost with the smallest amount of data.

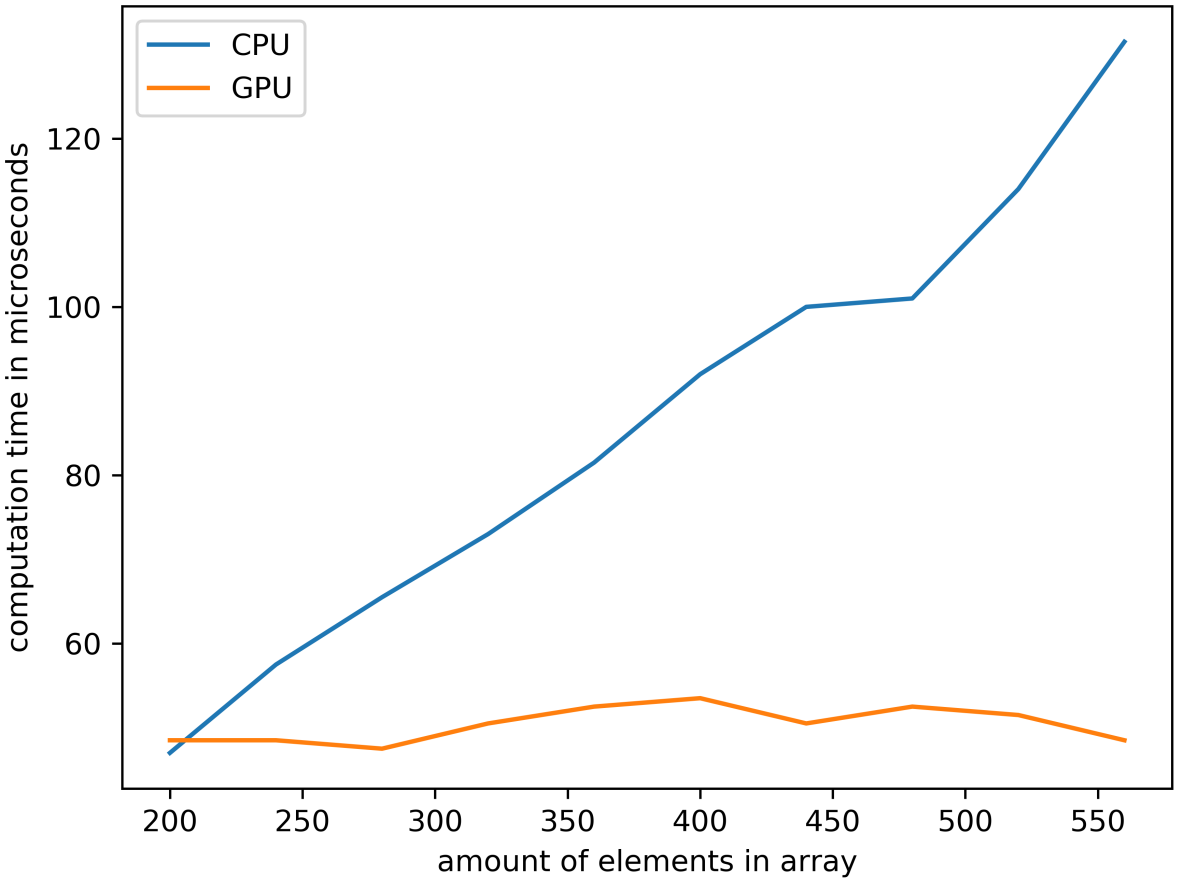


Figure 12: Smear calculation for EnergyResolution transformation

Conclusion

CUDA support was successfully integrated into GNA framework. GNA development group is working on its improvement. Even from the early phases it is gave promising results.

GPU codebase includes core library and transformation extensions. I was involved in implementation the second one. I was faced problems during porting of transformations. So addressing the identified problems in the future can significantly improve system generally. Ported transformation shows good results. However, there are still ways of further developing.

In this way CUDA support in GNA framework looks like especially perspective and efficient tool in data analysis and achievement of the assigned aims.

Acknowledgements

I am very grateful to my supervisor Anna Fatkina for invitation and given opportunity to take part in this especially interesting and valuable student program. Thanks for her support, patience, kindness, considerateness for me and professionalism in research field.

Special thanks to Summer Student Program organizers for providing the opportunity to being a part of such prestigious scientific institution as JINR. I am especially grateful to JINR and SSP organizers in particular for the financial support and for assistance provided in solving all the issues.

Also I would like to thank my university stuff. Thanks to academic adviser N. A. Volorova for given recommendation. Thanks to academic S. I. Sirotko for suggested idea to participate in SSP. Thanks to E. V. Kukar for assistance provided in some organization moments.

References

[1] *GNA repository*. https://git.jinr.ru/gna/gna.

[2] Daya Bay Collaboration et al. "A precision measurement of the neutrino mixing angle θ_{13} using reactor antineutrinos at Daya Bay". In: *arXiv preprint hep-ex/0701029* (2007).

[3] Fengpeng An et al. "Neutrino physics with JUNO". In: *Journal of Physics G: Nuclear and Particle Physics* 43.3 (2016), p. 030401. URL: http://stacks.iop.org/0954-3899/43/i=3/a=030401.

[4] *GNA documentation*. http://gna.pages.jinr.ru/gna/.

[5] A. Fatkina et al. *CUDA support in GNA data analysis framework*. English. Vol. 10963 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2018, pp. 12–24. URL: www.scopus.com.

[6] CUDA Toolkit Documentation. *Programming Guide*. 2016.

[7] Robert Hochberg. "Matrix Multiplication with CUDA - A basic introduction to the CUDA programming model". In: 44 (2012).

[8] Edward Kandrot Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. English. 2010.