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:
Ilya Lebedev

Participation period:
July 1 — August 25

Dubna, 2018

Contents

1 Introduction

Currently, there is a number of experiments aimed to studying neutrinos, such as JUNO, Daya Bay, NOvA, T2K, etc. These experiments require specific computational tools to analyze produced data. Analysis methods used in such experiments introduce huge computational costs. For example, Monte Carlo based methods typically take weeks or months on modern CPUs.

GNA (Global Neutrino Analysis) data analysis framework is being developed by JINR group [1]. It is originally created for Daya Bay [2] and JUNO [3] experiments, but it is designed in a such way, that it also can be used for any kind of such experiments. Implementation details are discussed in the next paragraph.

GPU support in GNA was announced this year [4]. Extension of GPU codebase of GNA was my main task during JINR SSP.

1.1 GNA

Global neutrino analysis software was created specially for analyzing data of neutrino experiments. The whole product is an attempt to implement the following general principles [5]:

- the whole structure should flexible enough to uniformly integrate arbitrary number of any kind of experiments into the one common flow;
- there should clean separation between analysis configuration step which is done once, and computations repeated a lot of 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).

The concept of GNA framework is to represent inputs and mathematical operations, required to build a model for experiment, as simple independent blocks, called transformations. These blocks are then connected into computational graph as it is shown in fig. 1, which produces required predictions and statistic. Transformations are implemented in C++ and ensure fast computation. The relations between them are expressed by Python, leading to great flexibility.

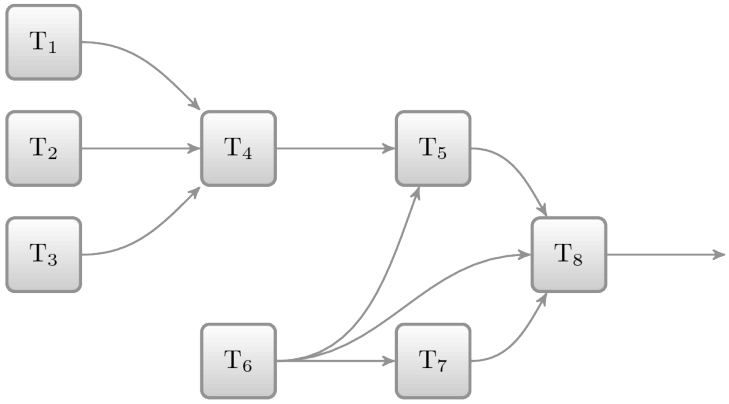

Figure 1: Schematic example of GNA graph

Transformations are computational core of GNA. Each transformation has one or more outputs and zero or more inputs. Inputs and outputs are usually arrays or array-based structures. Each transformation refers to data container that contains its outputs. To avoid extra recomputation it has special flag, which shows if data is relevant. The transformation evaluates if it tainted only. The block structure makes possible to track with high granularity the changes of computations depending on variable inputs, potentially avoiding useless recomputations during the fits. The drawback of the approach is more overhead due to dynamic nature of computations structure and keeping the parts independent, which can potentially overwhelm the possible advantages, depending on the implementation.

Currently, JINR developers are working on GPU support for GNA software. The idea is to use parallel computing and overcome high computational costs by harnessing the power of GPUs. But since computational task itself consist of a number of small simple blocks, they all should be ported on GPU in order to make the whole system work. The main tool to achieve this goal is CUDA — C/C++ extension for GPU.

1.2 CUDA

CUDA [6] is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). In GPU-accelerated applications, the sequential part of the workload runs on the CPU which is optimized for single-threaded performance while the compute intensive portion of the application runs on thousands of GPU

cores in parallel. 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.

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit (GPU) has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational power and very high memory bandwidth. The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation — exactly what graphics rendering is about — and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by the fig. 2.

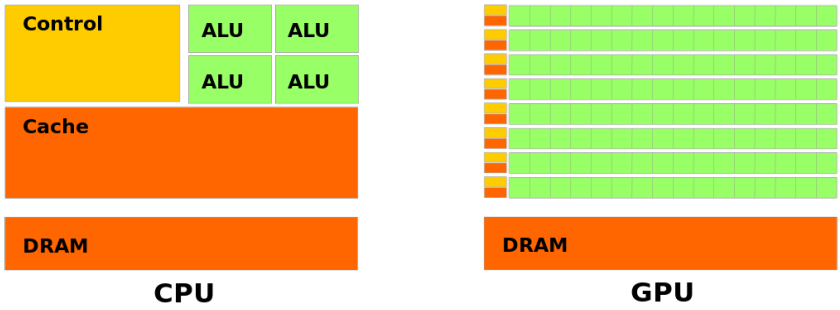

Figure 2: The GPU Devotes More Transistors to Data Processing

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations — the same program is executed on many data elements in parallel — with high arithmetic intensity — the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches [7].

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational

finance or computational biology.

GNA is suitable for GPU processing as it usually operate with matrices and arrays. Nevertheless, it can be improved and speeded up with data-parallel processing, only if the whole chain of transformations is ported to GPU, since data allocation and transfers from CPU to GPU and vice versa are quite slow and expensive part of some operations. Data transfer from CPU to GPU on each transformation call is not an option since it could lead to huge overall performance drop.

The idea is to transfer initial data from Host (CPU + RAM) to Device (GPU) once, complete whole computational graph on Device without Host-Device transfers and then transfer output data back to Host. Assuming the graph includes big amount of transformations, this will lead to high performance improvement. The example of data transfer is shown in fig. 3.

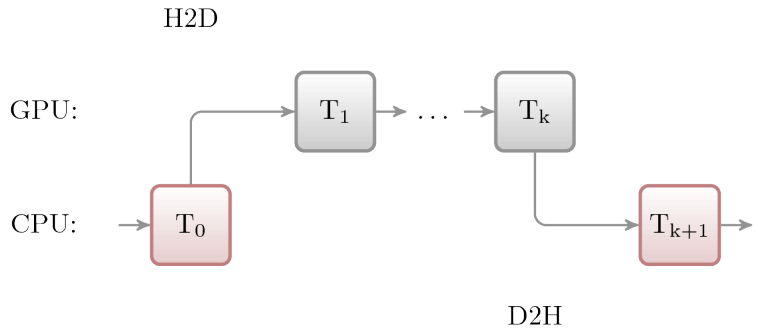

Figure 3: Schematic example of data transfer in GNA graph. H2D is data transfer from Host to Device and D2H is data transfer from Device back to Host.

2 Transformations

Currently there are about 50 different transformations implemented in GNA. Within the JINR Summer Student Program I was able to port to GPU 10 of them, which include:

1. Sum
2. WeightedSum
3. WeightedSumFill
4. Product
5. Exp
6. SelfPower
7. Transpose
8. Poisson
9. Normalize
10. Renormalize

In this chapter the explanation and details of GPU implementation for each transformation is given.

1. Sum is the transformation, intended for sum of arrays or matrices. Input is N arrays or matrices of same dimension. Output is a single array or matrix. CPU version of transformation requested $N \times M$ iterations (where M is array length). In GPU version it only needs N iterations, since there are M parallel threads, which compute sum for each column.

   Calculation formula is

$$S = \sum_{k=1}^{N} a_k.$$

2. WeightedSum calculates weighted sum of arrays. Input is a vector of arrays' names of size N and a vector of weights' names, empty or of size N. Output is single array or matrix. CPU version of transformation. GPU implementation is same as Sum, except it needs one more iteration to product vectors or matrices to corresponding weights.

   Calculation formula is
   $$S = \sum_{i=1}^{N} \omega_i A_i.$$

3. WeightedSumFill is same as WeightedSum, additionally takes constant and adds vector filled by this constant to the sum. GPU implementation is same as WeightedSum.

   Calculation formula is
   $$S = \sum_{i=1}^{N} \omega_i A_i + C.$$

4. Product calculates an element-wise product of several arrays. Input is N arrays or matrices of same size and N weights. Output is a single array or matrix. GPU version, same as Sum, uses M threads and requires N iterations instead of $N \times M$ in CPU.

   Calculation formula is
   $$P_{ij} = \prod_{k=1}^{N} (a_k)_{ij}.$$

5. Exp returns exponential function applied to each element of input array. GPU version uses M threads (M is array length) and requires only one iterations versus M iterations in CPU. This leads to huge performance improvement in case of big sizes of arrays.

   Calculation formula is
   $$f_i = \exp x_i.$$

6. Self Power computes the result of a coefficient-wise $(x/a)^{\pm x/a}$ function. GPU implementation is similar to Exp transformation. Input and output is single array size of M.

   Calculation formula is
   $$f_i = \left(\frac{x_i}{a}\right)^{\frac{x_i}{a}}.$$

7. Transpose returns transposed input matrix. GPU issues a thread per each element of matrix. This operation takes only one operation to accomplish, which leads to very high performance on big matrices.

8. Poisson calculates the Poisson loglikelihood function value. Inputs are K pairs of theory μ of size N and data x size of N. Output is -2 Poisson loglikelihood function value for each pair of arrays.

   Calculation formula is

$$L(x|\mu) = \prod_{i=1}^{N} \frac{(\mu_i^{x_i} e^{-\mu_i})}{x_i!}.$$

   In source code there is used the natural logarithm of $L(x|\mu)$, so formula is

$$\log L(x|\mu) = \sum_{i=1}^{N} (x_i \log(\mu_i) - \mu_i - log(x_i!)),$$

   where N is a size of vectors μ and x.

9. Normalize for a given histogram or array divides each bin/element by a sum of all the elements. Input is array or histogram A. Output is array B of the same shape as A. Transformation uases K threads, where K is amount of bins/elements in A.

   Calculation formula is

$$B_{j,\ldots} = \frac{A_{j,\ldots}}{\sum\limits_{i,\ldots} A_{i,\ldots}}.$$

10. Renormalize for a given square matrix this transformation either:

    (a) Scales n diagonals by a given number.

    (b) Scales all the elements except n diagonals by a given number.

    After scale is applied, each column is normalized to 1.

    Renormalize is used to implement IAV correction uncertainty (Daya Bay) and is supposed to be used with EnergySmear transformation.

    Calculation formula is

$$E_{ij} = \frac{D_{ij}}{\sum\limits_{k} D_{kj}},$$

    where D in Diagonal case is

$$D_{ij} = \begin{cases} sC_{ij} & \text{if } |i-j| < n \\ C_{ij} & \text{otherwise} \end{cases}, \tag{1}$$

and in Offdiagonal case it is

$$D_{ij} = \begin{cases} C_{ij} & \text{if } |i-j| < n \\ sC_{ij} & \text{otherwise} \end{cases}. \tag{2}$$

Besides that, during SSP I was working on the following transformations:

- Cholesky. This transformation computes the Cholesky decomposition of the symmetric positive definite matrix. Input is matrix V. Output is lower triangular matrix L, such that $V = LL^T$. This is the only transformation, which required usage of cuBLAS library. The NVIDIA cuBLAS library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS). cuBLAS function `cusolverDnDpotrf` was used in order to compute the Cholesky decomposition. Currently, GNA doesn't fully support cuBLAS library, so this transformation wasn't integrated and tested.

- Matrix Product computes product of K matrices. Assuming each matrix is $N \times M$, first step is to find maxN and maxM. Then GPU issues a thread per each element of output matrix, which has maxN$\times$maxM elements. GPU function copies the first input matrix into output matrix, inserting 0 instead of excess elements. Then it performs $K - 1$ iterations of computing product. Due to architecture of this transformation, it can process any matrix, which amount of elements is less than CUDA block size, which is limited. If it requires to use more than 1 block, it leads to unstable thread behavior and computational errors. Currently, there exists a way to handle this behavior for 2 matrices [8], but product of K matrices is a topic to be studied. Due to unstable behavior with $N \times M$ matrices, this transformation wasn't integrated and tested.

3 Testing

In this chapter transformation test results and analysis is provided. All tests were performed with following hardware:

- Processor: Intel Core i7-3630QM CPU @ 2.40GHz 4 cores
- Graphics card: GeForce GTX 670MX @ 960 CUDA cores

Sum transformations.

Due to the similar architecture, Sum, WeightedSum and WeightedSumFill transformations were tasted on the same data. It consisted of N arrays size of M, where N is $[2..50]$ and $M = 10^i$ for $i = [1..5]$. In order to avoid deviations, each transformation was tested 50 times and median value was taken as result.

Test results clearly show that GPU transformation requires only about 15 microseconds to compute sum of N arrays size of M with any N and M in listed boundaries. On the other side, CPU transformation computational time varies from 1 microsecond up to 4500 microseconds and depends on input N and M. Computational time plots show that CPU function is more effective for small data processing (arrays with < 1000 elements), while GPU processes bigger input arrays much faster, up to 300 times faster on $N = 50, M = 100000$. Fig. 4 shows that cross point for these transformations is $N = 1000$ and $M = 25$, where both CPU and GPU transformations require about 15 microseconds to compute. We can affirm that if amount of data is higher than that, it is more suitable to use GPU, otherwise it is cheaper to use CPU transformation. Test results including plots for each N can be found in Appendix.

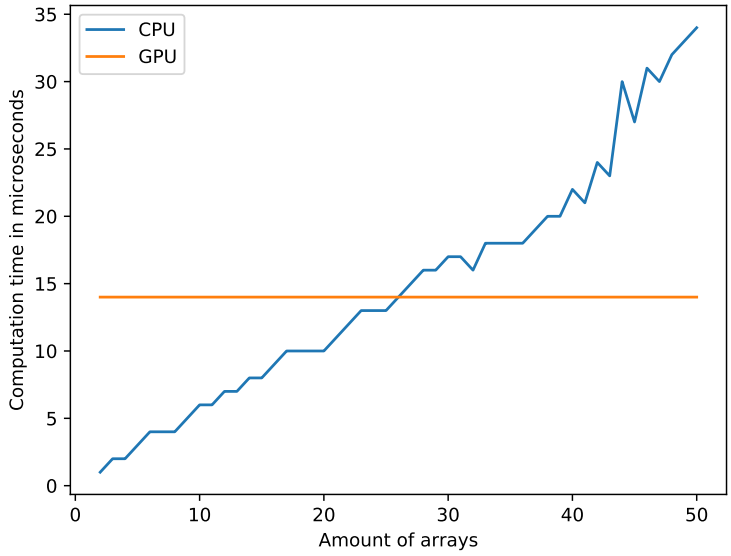

Figure 4: Sum transformation test results with N = 1000

Approximately the same result shown in fig. 5 can be obtained from WeightedSum transformation tests:

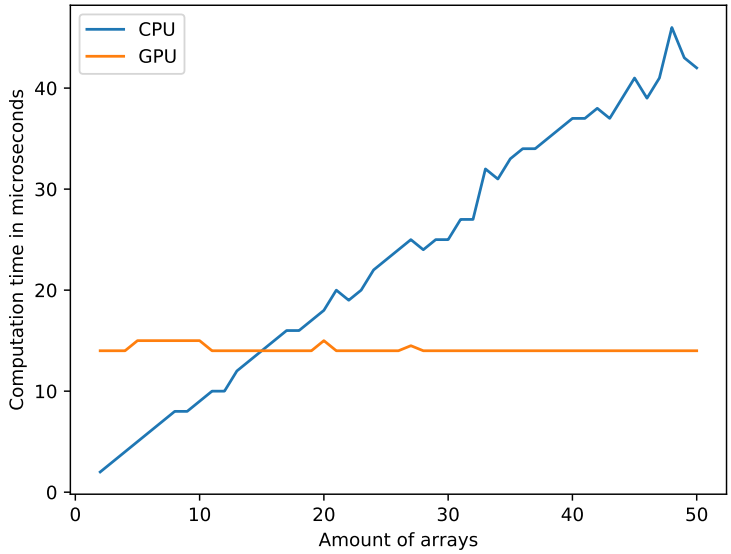

Figure 5: WeightedSum transformation test results with N = 1000

WeightedSumFill transformation tests show almost the same results and WeightedSum due to similar architecture.

Product transformation.

Product transformation was tested on the same data as Sum transformations (N arrays size of M). Results are almost the same as for Sum transformations tests. It is shown in fig. 6.

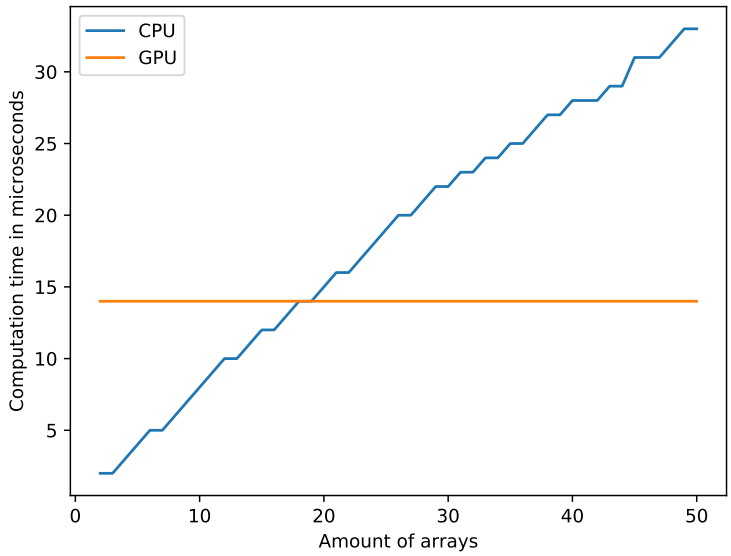

Figure 6: Product transformation test results for N = 1000

Exp transformation.

This transformation takes one vector as input, it testing data was different from Sum and Product transformations. It was tested on a single vector with N amount of elements, where $N = [1..100000]$. Test results show the same CPU and GPU behavior in computational time: CPU is better on small amount of elements and GPU. On the other side, is much faster on big amount of elements ($N > 1000$). Amount of elements, corresponding to intersection point in fig. 7 is about 2300.

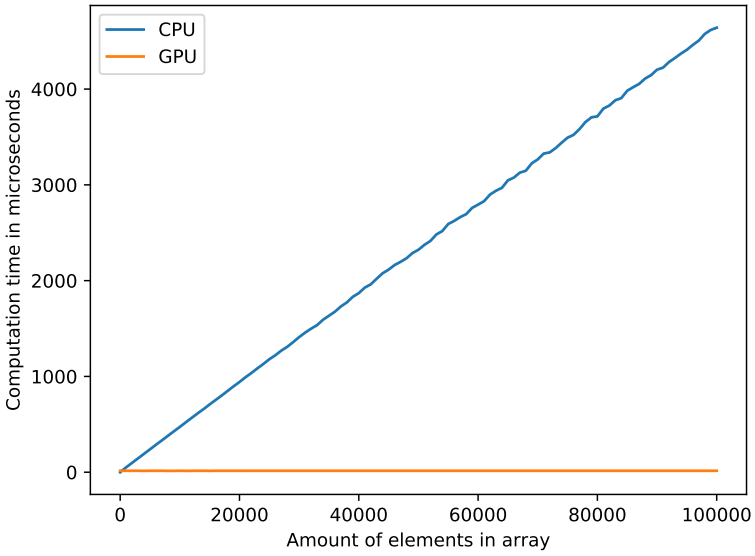

Figure 8: SelfPower transformation test results

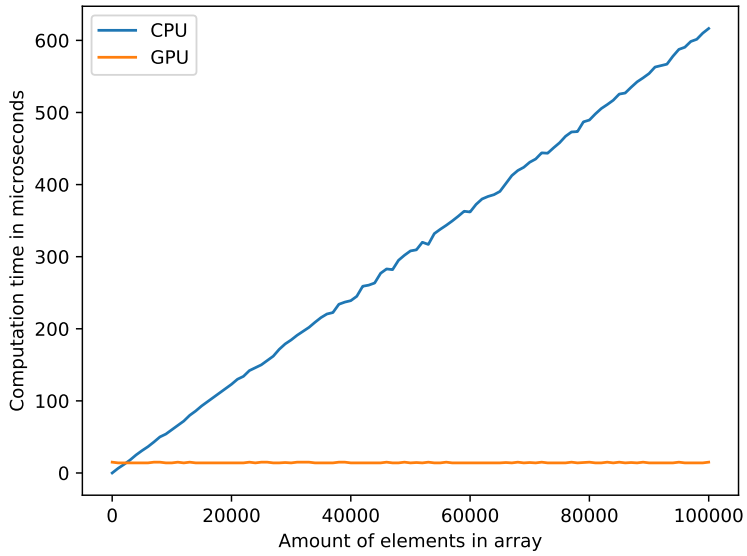

Figure 7: Exp transformation test results

SelfPower transformation.

SelfPower takes a single array as input, same as Exp, so the testing data is same. Since SelfPower requires more computations, GPU surpasses CPU even faster than in Exp case. Amount of elements, corresponding to intersection point in fig. 8 is about 300.

Transpose transformation.

Transpose transformation takes a single matrix as input. In order to test this transformation, matrix of $N \times N$ elements was taken, $N = [1..100]$. Intersection point for CPU and GPU time shown in fig. 9 is about 280. This means that GPU is more effective for square matrices if $N > 17$.

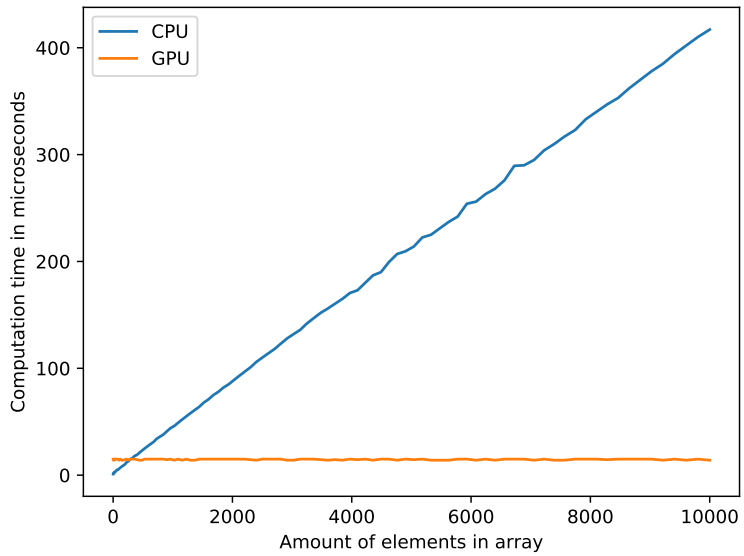

Figure 9: Transpose transformation test results

Poisson transformation.

Poisson transformation was tested on 2 input arrays, with N elements each, $N = [1..100000]$. Fig. 10 that intersection point for CPU and GPU time at N approximately equal to 200.

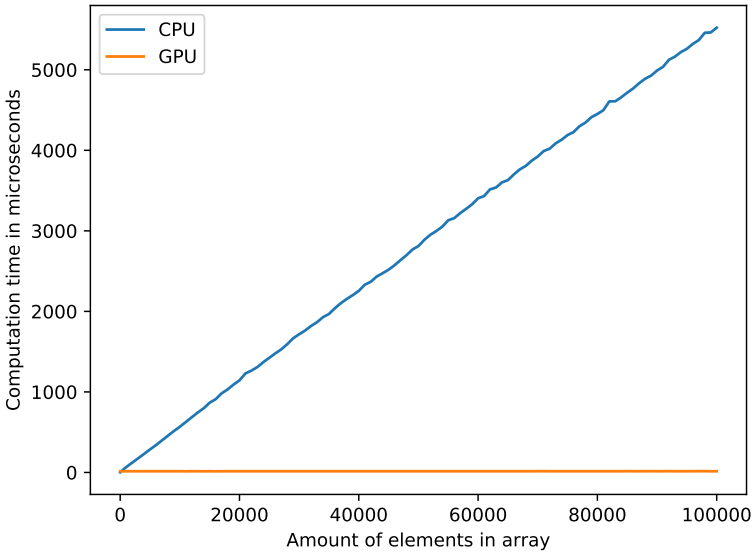

Figure 10: Poisson transformation test results

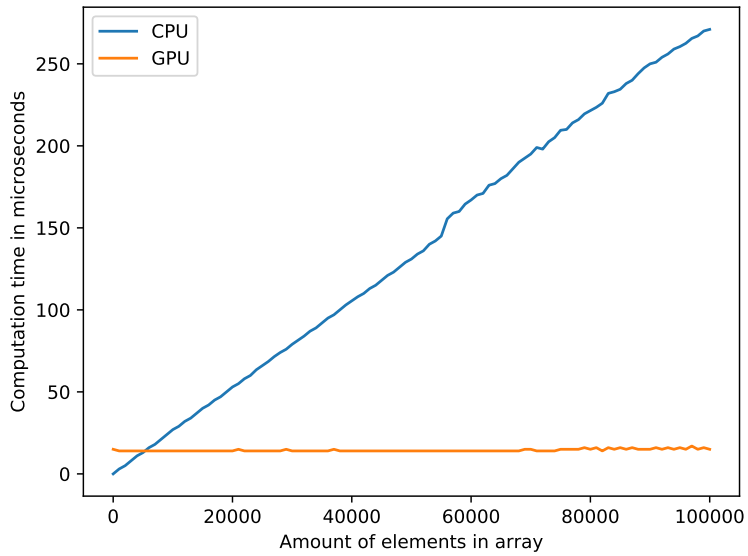

Figure 11: Normalize transformation test results

Normalize transformation.

Normalize transformation was tested on the same data as Exp transformation due to the same input structure. Intersection point at N approximately equal to 5300 in fig. 11. Therefore, if input arrays has more than 5300 elements it is reasonable to use GPU.

Renormalize transformation.

Renormalize transformation takes a single matrix as input, so it was tested on the same data as Transpose transformation: matrix size of $N \times N, N = [1..100]$. Amount of diagonals = 1. Test result is presented in fig. 12.

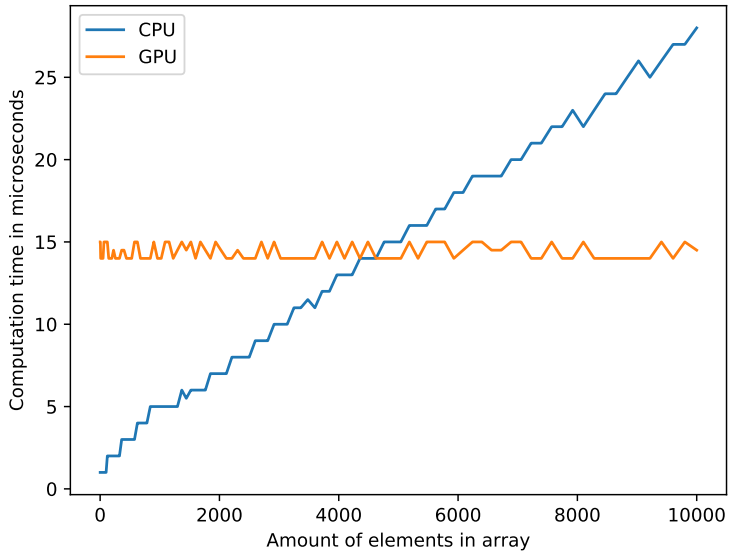

Figure 12: ReNormalize transformation test results

Resume

We can conclude some statements based on the received results. Firstly, CPU transformations are more effective on small data and CPU computational time shows linear growth with increasing of input data. Secondary, GPU transformations are much more effective on big amount of data, starting from few hundred of elements. It is much more suitable for real computations because on practice GNA works with arrays, which contain about 10000 elements.

It should be noted, though, that data transfer time is not included in tests and it can affect final results of graph computation time. But the idea of GPU integration is to make only two main data transfers in graph total - one for input and one for output. Assuming that we have only two data transfers total, GPU transformations show really big potential and promising results.

Conclusion

JINR group has created GNA software to analyze results of neutrino experiments and they are actively developing it. I contributed to the development of GNA during JINR Summer Student Program. I was working on GPU implementation of GNA transformations. I have studied about GPU programming and applied this knowledge in practice. 10 transformations in total were successfully ported to GPU.

We tested every transformation and results were very promising. Although it requires more work to be done in the future, the GPU extension for GNA shows high potential.

Acknowledgements

First of all, I would like to thank Dmitry Naumov for the opportunity to participate in JINR SSP. I also would like to thank my supervisor, Anna Fatkina, for interesting tasks and support during Summer Student Program. I am very grateful to JINR Summer Student Program organizers for providing such educational opportunities. It was very interesting and valuable experience for me to become a part of JINR community.

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] 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.

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

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

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

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

Appendix 1. Sum transformation testing results

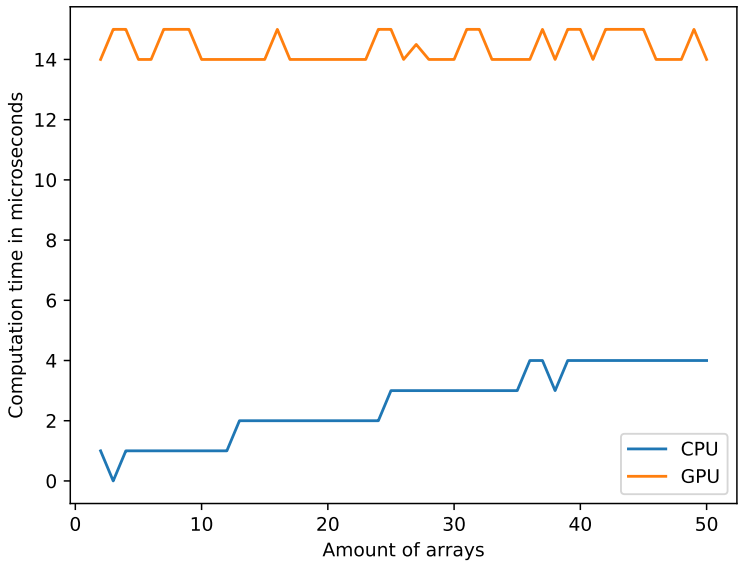

Figure 13: Sum transformation test results with N = 10

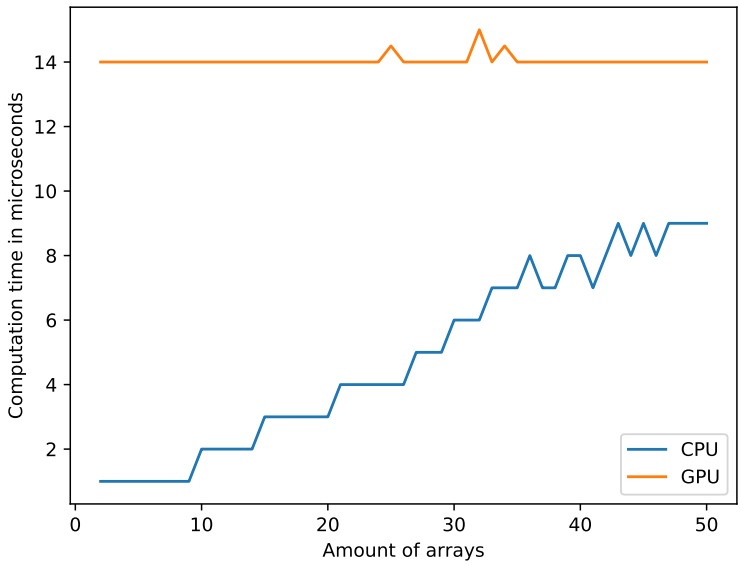

Figure 14: Sum transformation test results with N = 100

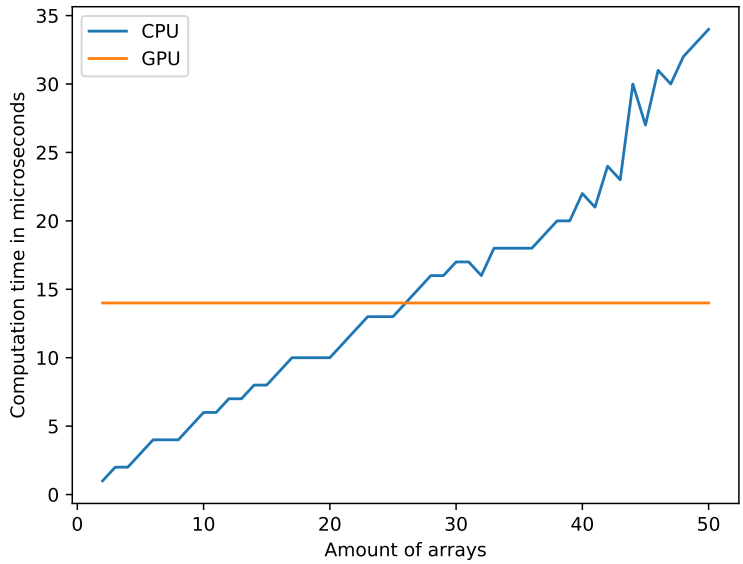

Figure 15: Sum transformation test results with N = 1000

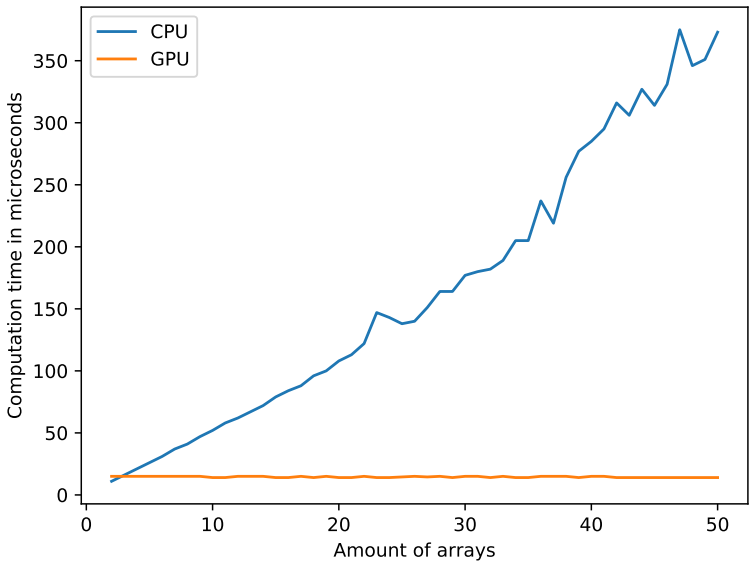

Figure 16: Sum transformation test results with N = 10000

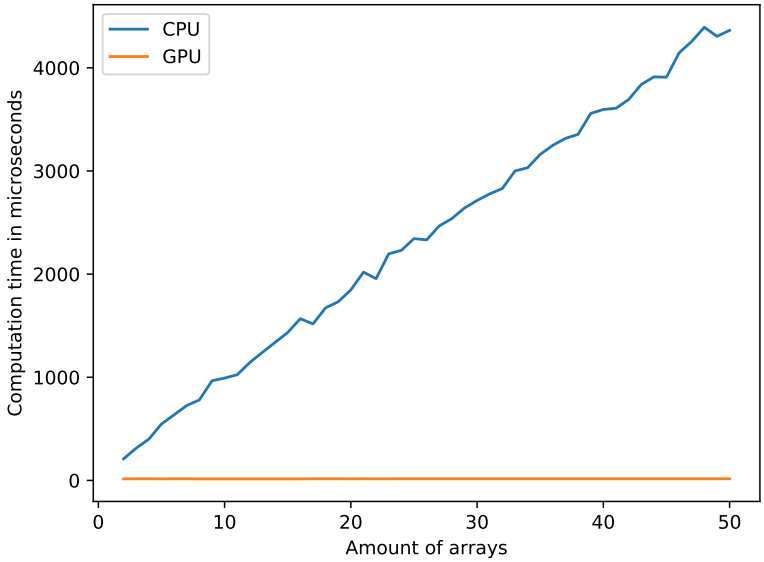

Figure 17: Sum transformation test results with N = 100000

Appendix 2. WeightedSum transformation testing results.

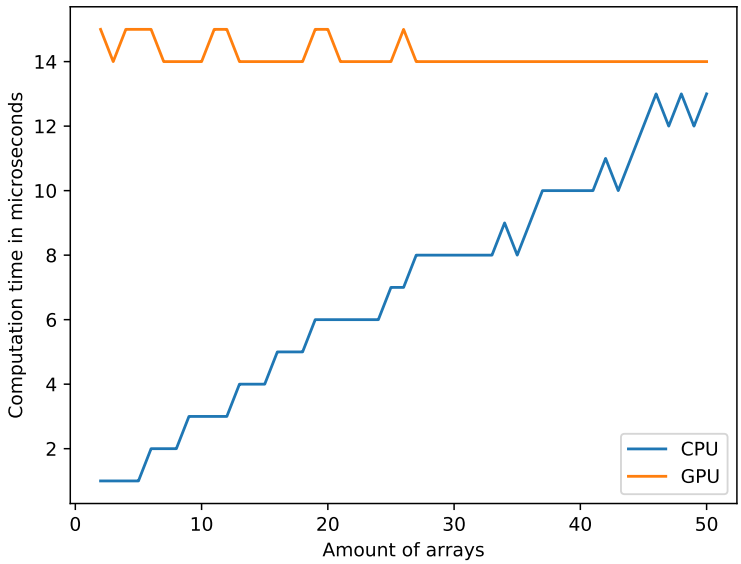

Figure 18: WeightedSum transformation test results with N = 10

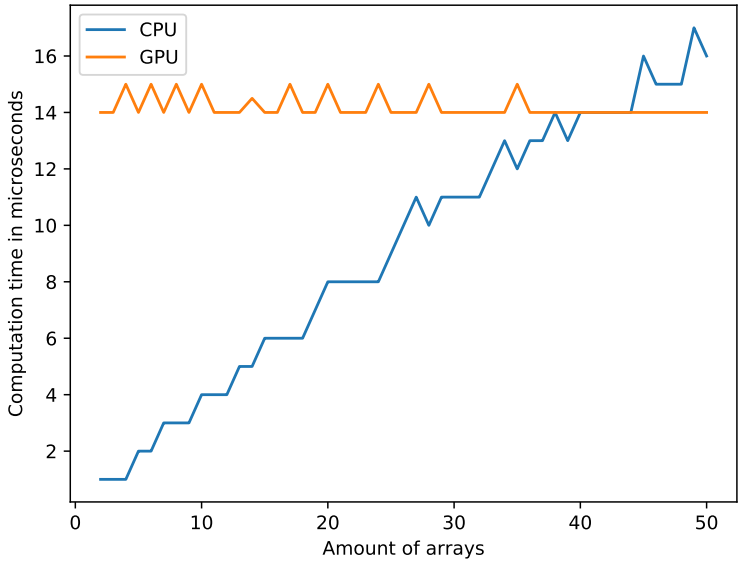

Figure 19: WeightedSum transformation test results with N = 100

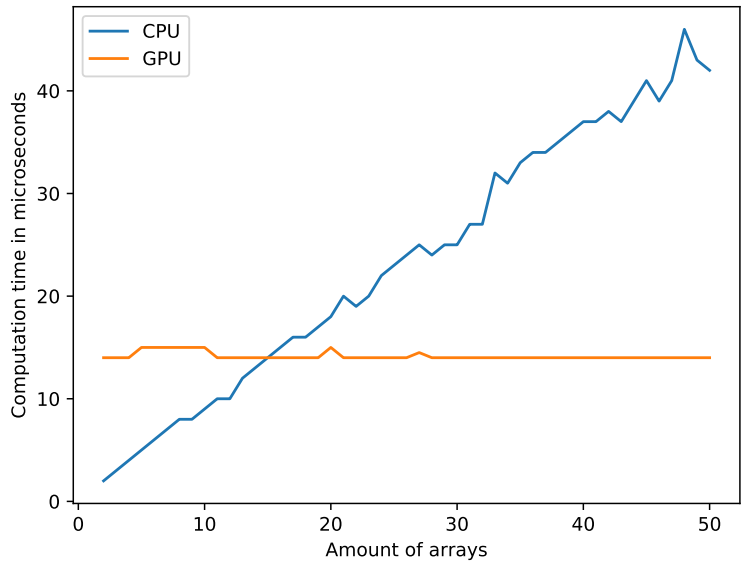

Figure 20: WeightedSum transformation test results with N = 1000

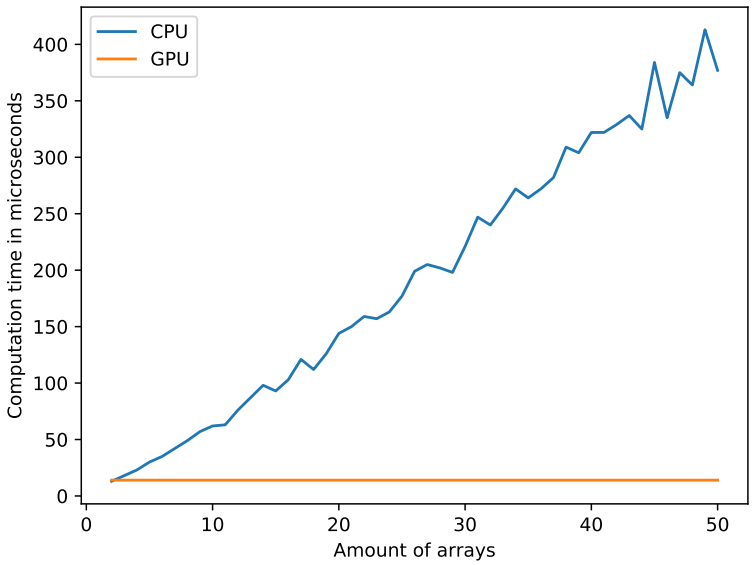

Figure 21: WeightedSum transformation test results with N = 10000

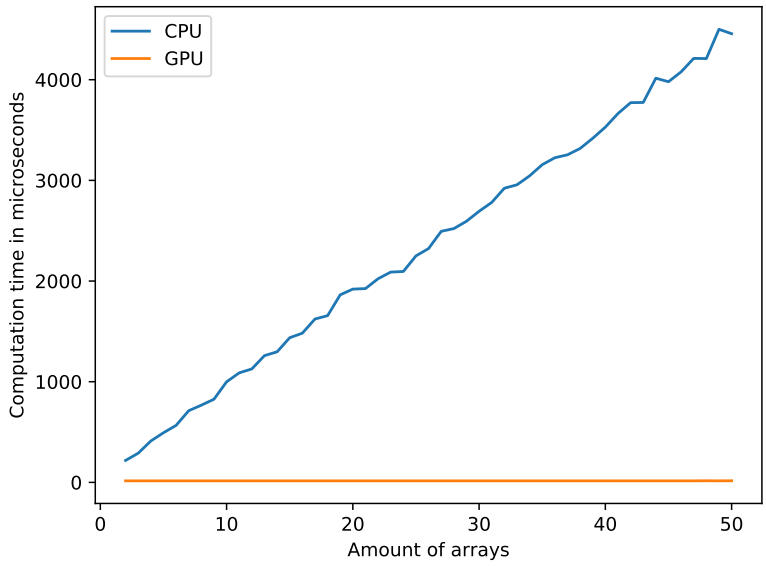

Figure 22: WeightedSum transformation test results with N = 100000

Appendix 3. Product transformation testing results.

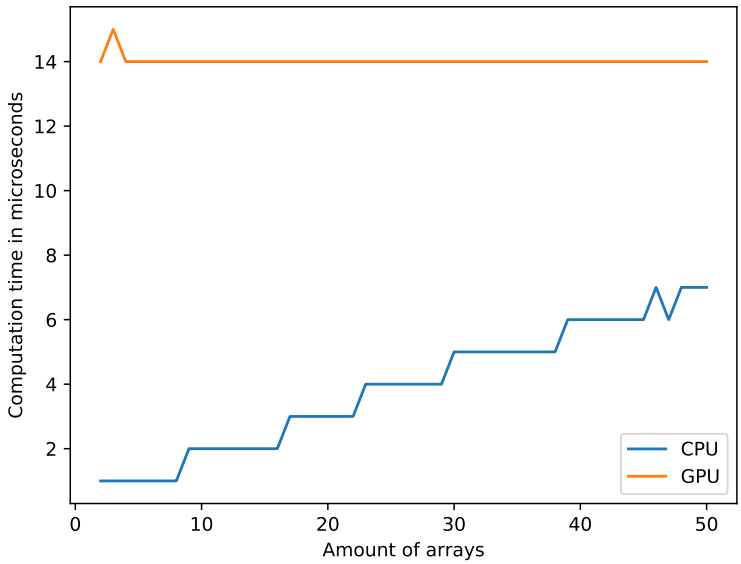

Figure 23: Product transformation test results with N = 10

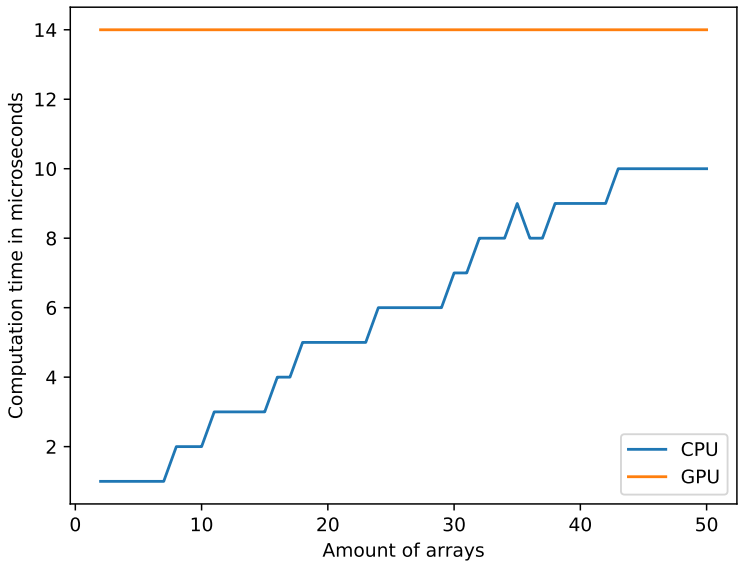

Figure 24: Product transformation test results with N = 100

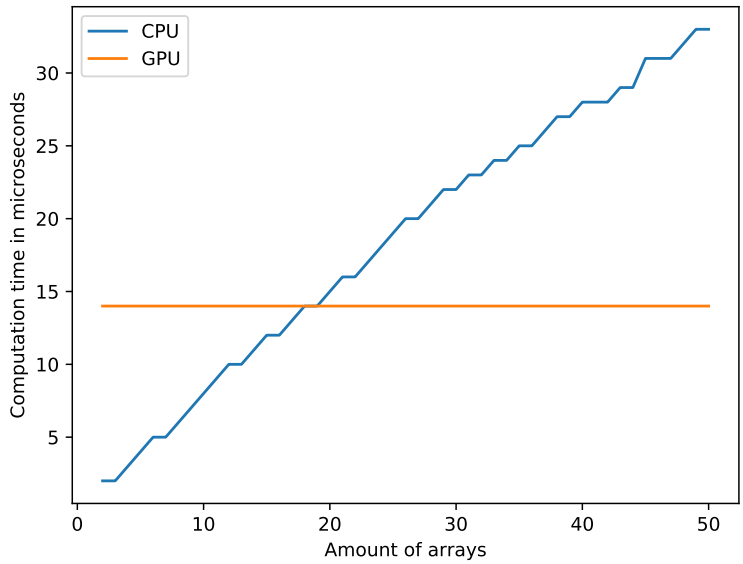

Figure 25: Product transformation test results with N = 1000

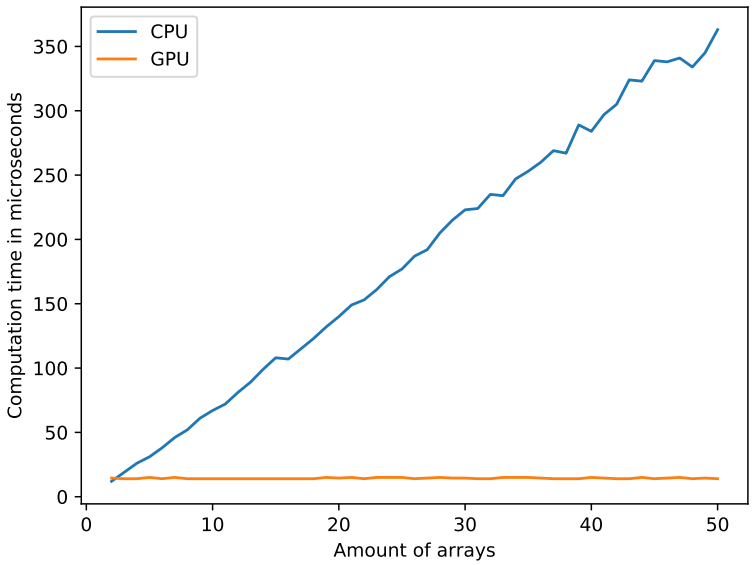

Figure 26: Product transformation test results with N = 10000

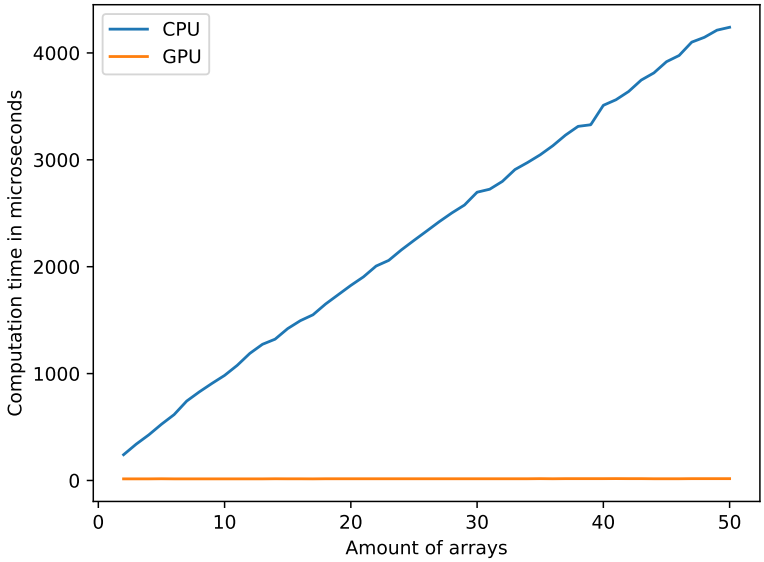

Figure 27: Product transformation test results with N = 100000