Andrii Antoniuk, Volodymyr Rusinov.

ACCELERATION OF NEURAL NETWORK TASKS ON HETEROGENEOUS CPU-GPU SYSTEMS

The article analyzes the issue of heterogeneous CPU-GPU system use in accelerating applied tasks, related to the neural network learning process.

Keywords: heterogeneous systems, neural networks, machine learning, CPU, GPU.
Fig.: 4. Tab.: 3. Bibl.: 11.

Relevance of the research topic. Area of research that deals with heterogeneous computing is currently underdeveloped. In machine learning, for example, huge effort goes into creating algorithms that are more efficient and getting more performance from GPUs, which leads to restricting CPU usage only to OS maintenance tasks such as scheduling for example. Despite that, most supercomputers and large data centers use systems that have both CPUs and GPUs [1]. General-purpose computing on GPU (GPGPU) has paved the road for PC users to experiment and work on projects that involve GPUs to outsource laborious tasks. CUDA, OpenCL are two examples of libraries that allow that [2, 11].

Another point to consider is recent developments in the CPU market, where 4 core CPUs have become standard, some 15 years from dual-core CPU introduction to the market, with enterprise or server level CPUs sporting up to 64 cores, which is reaching the recommended theoretical limit of SMP architecture. This in turn creates additional cost-efficient ways of getting CPU power.

There is also an increasing demand for neural network use in many practical spheres, as many large companies are developing intelligent systems that work directly with Big Data solutions. This is due to meeting two of the most important demands for effective neural network use: quality of the data and an increasing amount of compute power. Success in the last decade in neural network use has been dubbed Renaissance. This success is further fueled by the wide availability of cheap and performant CPUs, GPUs, and open source solutions to directly interact with them.

Target setting. Neural networks, as a way to find solutions to practical problems, are becoming more widespread each day. With this growth comes the demand for high-performance computer systems.

Actual scientific researches and issues analysis. Many studies explore how to work with GPUs, what are the best practices, how to create revise CPU algorithms to work on GPU [5]. CPUs are also very much explored. However, as stated earlier, heterogeneous systems are not as thoroughly examined and there are some articles on how to work with them, but the area is still not well understood.

Articles that deal with heterogeneous systems show promising results [1, 2, 4]. Heterogeneous CPU-GPU systems, although, underexplored, have shown to accelerate tasks related to machine learning, image processing, and Big Data tasks [1, 2].

Uninvestigated parts of general matters defining. Python is a popular programming language and part of its fame comes from machine learning libraries, such as Tensorflow, Scikit, etc. Most of such libraries put effort into the optimization of their workload around the GPU and Tensorflow goes a step further and finds ways to run better on TPUs. This approach is more reasonable in PaaS cloud solutions that usually provide end-users with high-performance GPU and modestly performing, low latency CPUs, or virtualize a powerful CPU into several virtual CPUs. Most other cases, including PaaS solutions that provide access to high-performance CPU and GPU, usually do not benefit from this union [3].

The research objective. The goal of this article is to determine if it is feasible to use heterogeneous systems for neural network tasks. To achieve this goal, a set of experiments will be conducted that involves timing each subtask and based on that, determine the speedup of each system.

The statement of basic materials. Before looking too deep into the problem, first, we need to understand what hardware we are dealing with. All of the experiments are run on NVidia GPUs. Architecturally, GPUs are quite different from CPUs. While 'casual' customers focus on simple metrics – memory, memory frequency, processor frequency, professional customers are more interested in what goes into such devices. There is much more to it. For example, let us take TU102, used in RTX 2080Ti mainstream GPU and Quadro RTX 6000 professional counterpart. This model is used as an example because each contemporary model of GPU derives for this design. TU102 subdivides into 6 graphics processing clusters, 36 texture processing clusters, and 72 streaming multiprocessors (SM). Each SM consists of 64 CUDA cores, 8 Tensor cores, 256 KB register file, 4 Texture units, and 96 KB of shared memory. On top of that, each core has access to 6144 KB of L2 cache. To add to the complexity of this design, SM's cores are divided depending on their use, so instead of saying CUDA cores, a more appropriate way to address them is by what type of data they compute. There are 64 FP32 Cores and 64 INT32 cores. Therefore, GPU is a very specialized device, which makes use of hardware acceleration [6].

With architecture in mind, NVidia has created a specialized software solution to access GPU potential called CUDA (Compute Unified Device Architecture). CUDA lets developers use NVidia GPUs for GPGPU tasks. CUDA platform gives access to a set of instructions, commands, and parallel programming tools to successfully develop and run compute kernels. Compute kernel is a routine, compiled for hardware accelerators, in our case, GPUs. CUDA is widely used across different libraries, though, officially, its only supported languages are C and C++ [10, 11].

As stated before, this article's goal is to find out whether it is feasible and quantify how well it runs for neural network tasks, and the most popular language for data science, data engineering, and machine learning engineering is Python. Numba

for Python is a library that gives access to JIT-compiler that can compile Python code into compute kernels [11].

Testing is performed on the following computer systems:

1. AMD Ryzen 9 3900X 12 cores, 24 threads, 3.8 GHz base clock, 4.1 GHz boost clock, RTX 2060, 16GB of RAM, X570 chipset.
2. AMD Ryzen 5 2400G 4 cores, 8 threads, 3.6 GHz base clock, 3.8 GHz boost clock, GTX 1060-3 GB, 16 GB of RAM, A320 chipset.
3. AMD Athlon 760K 4 cores, 4 threads, 3.8 GHz base clock, 4.1 GHz boost clock, GTS 450, 4 GB of RAM, X89 chipset.
4. Intel Core i5-7200U 2 cores, 4 threads, 2.5 GHz base clock, 3.1 GHz boost clock, Geforce 940MX, 8 GB of RAM, KBU chipset.
5. Intel Xeon E5-2630 2 cores, 2 threads, 2.3 GHz base clock, 2.8 GHz boost clock, Tesla K80, 8 GB of RAM, C604 chipset.

To do experiments with said systems, a simple planning solution is required. To put it simply, we need to split the job between CPU and GPU. When dealing with the CPU-GPU system, you have to understand that execution time on GPU is not something that can be predicted. Before we do experiments, we need to measure how much time it takes for CPU and GPU to finish the task and to see how much time it takes to send the data back from GPU.

The task itself involves some of the steps that the neural network model takes while learning. These steps are normalization, weighing, activation.

Normalization is a function that fits the initial values to a common scale, in our case, ranging from 0 to 1. This is not unlike database normalization procedure and different neural networks developed for other scenarios will use different normalization techniques, depending on the input data.

Weighing is a process of applying a weight parameter to input data by multiplication. Weight is a coefficient that is constantly changing in neural networks during learning. It defines how the feature of the node this weight represents influences the desired outcome.

Activation uses a special activation function that defines output. There are many different activation functions, for this case, we will be using the hyperbolic tangent function, defined by the following formula:

$$f(x) = \tanh(\mathrm{x}) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \quad (1)$$

The following graphs (Fig.1-3) show execution times on GPU and CPU as well as GPU to CPU send times.

Using the time measurements, we create a simple regression model using least squares method. However, before that, step one is to make an initial split using the

following formula (2):

$$N_{gpu} = W_{gpu} * N \quad (2)$$

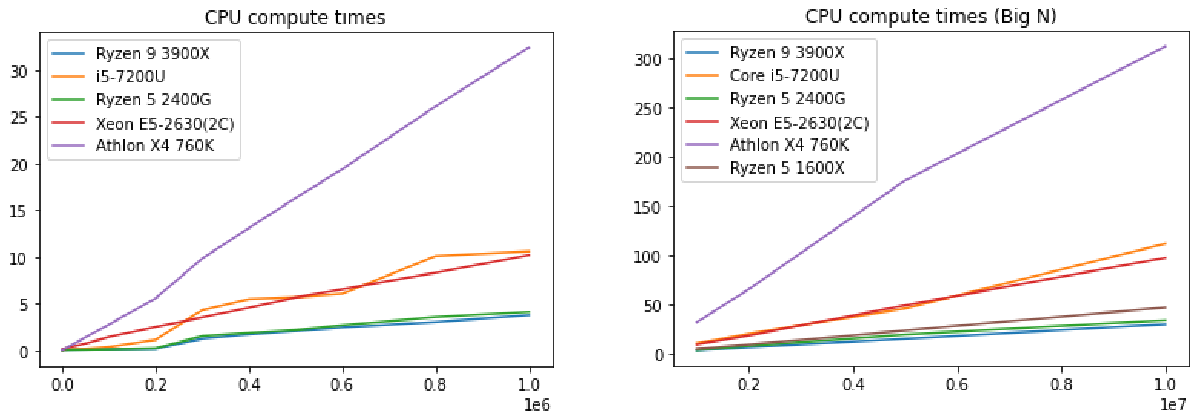

Fig. 1. Compute times on CPU

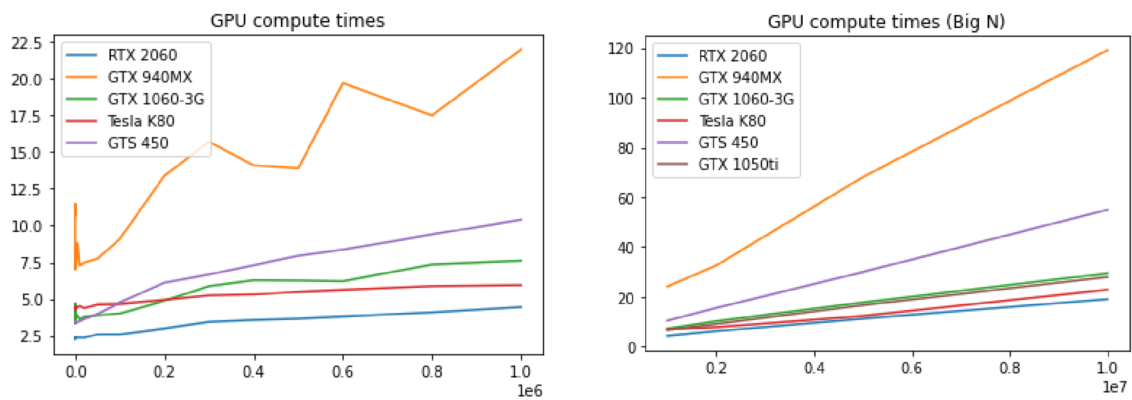

Fig. 2. Compute times on GPU

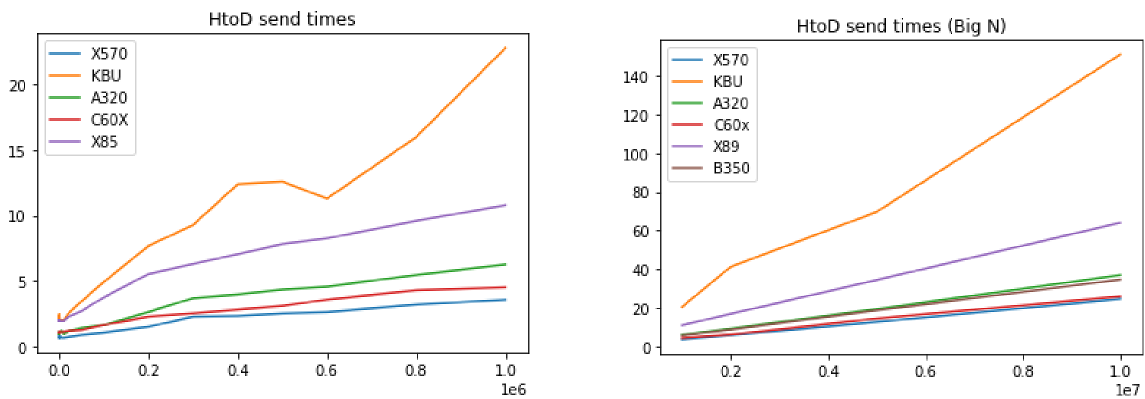

Fig. 3. Send times from GPU to CPU

Where N is task dimension and W_{gpu} stands for computation weight. To compute computation weight, the following formula is used:

$$W_{gpu} = \frac{t_{gpu}}{t_{gpu} + t_{cpu}} \quad (3)$$

Where t_{gpu} is execution time on GPU, t_{cpu} is execution time on CPU and W_{gpu} is weight of GPU computations.

Step two is to take the regression model and find the value for Ngpu and Ncpu computed in step one. This way we get a new pair of time predictions, with which we make a new split. This step is repeated 10 times to get a more 'fair' split.

After the job has been split, we get the following results (Table 1):

Table 1

Job split between the GPU and CPU

Dimension	**GPU**	**CPU**
R9 3900X + RTX 2060		
5000000	2098724	2901276
10000000	4302225	5697775
25000000	10783998	14216002
50000000	21500424	28499576
R5 2400G + GTX 1060-3GB		
5000000	1897180	3102820
10000000	3319339	6680661
25000000	9503751	15496249
50000000	18664993	31335007
Dimension	GPU	CPU
I5 7200U + Geforce 940MX		
5000000	1708896	3291104
10000000	3308685	6691315

Ended Table 1

Dimension	**GPU**	**CPU**
25000000	7836838	17163162
50000000	15244193	34755807
Xeon E5-2630(2C) + Tesla K80		
5000000	2112889	2887111
10000000	6198501	3801499
25000000	16353979	8646021
50000000	33230526	16769474
Dimension	GPU	CPU
Athlon X4 760k + GTS 450		
5000000	3301774	1698226
10000000	6930421	3069579
25000000	17943324	7056676
50000000	36351454	13648546

To quantify the speedups, we introduce two coefficients: speedup coefficient relative to CPU and speedup coefficient relative to GPU, since we are exploring a heterogeneous system, CPU and GPU perform differently. To calculate the speedup coefficient relative to CPU, we use the following formula:

$$K_{pg} = \frac{T_g + T_s}{T_h}$$

Where Tg is time to finish the task on the GPU, Ts is time to send the data from the GPU to the CPU, and Th is time to finish the task on a heterogeneous system.

To calculate the speedup coefficient relative to GPU, we use the next formula:

$$K_{pg} = \frac{T_c}{T_h}$$

Where Tc is time to finish the task on the CPU and Th is time to finish the task on a heterogeneous system.

To get these results, the program has been launched 100-1000 times, depending on how deviant results are. Results are an average of 100-1000 time measurements made during execution.

Conclusion. This article studied opportunities to accelerate neural network subtasks like normalization, weighing, and activation using combined compute potential of CPU and GPU running in parallel. Results show that the heterogeneous approach is effective in neural network tasks. Applying this approach significantly reduced the overall execution time.

From the results, we can infer that the program is taking advantage of the new CPUs and performance speedups relative to GPU range from 1.11 to 4.39 (Table 2) and performance speedups relative to CPU range from 0.96 to 3.48 (Table 3) show that in most cases, the CPU performance can be enhanced if GPU is also utilized (Fig. 4).

In some runs, the CPU-only approach shows better results as the coefficient sometimes goes below 1 and reaches 0.96 (Fig. 4), however, for most experiments that is not the case. This may take place due to how much time it takes for the GPU to send back to the CPU all of the computed data. Data transfer alone takes 40-60% of the time. Referring back to the graphs shows that even a single floating-point number can take up to 6 milliseconds for GPU to handle, due to the synchronous nature of the GPU design.

Table 2

Speedup coefficient relative to the GPU

System	**5000000**	**1000000**	**25000000**	**5000000**
R9 3900X + RTX 2060	1.73	2.01	2.18	2.36
R5 2400G + GTX 1060-3GB	1.8	2.14	2.59	2.77
I5-7200U + Geforce 940MX	2.94	3.66	4.06	4.39
Athlon X4 760k + GTS 450	1.16	1.25	1.25	1.26
Xeon E5-2630 (2C) + Tesla K80	1.11	2.82	2.71	3.01

Table 3

Speedup coefficient relative to the CPU

System	5000000	1000000	25000000	5000000
R9 3900X + RTX 2060	1.12	1.39	1.58	1.73
R5 2400G + GTX 1060-3GB	0.96	1.1	1.43	1.55
I5-7200U + Geforce 940MX	0.99	1.51	1.66	1.81
Athlon X4 760k + GTS 450	3.17	3.28	3.43	3.48
Xeon E5-2630 (2C) + Tesla K80	1.59	1.72	2.61	2.81

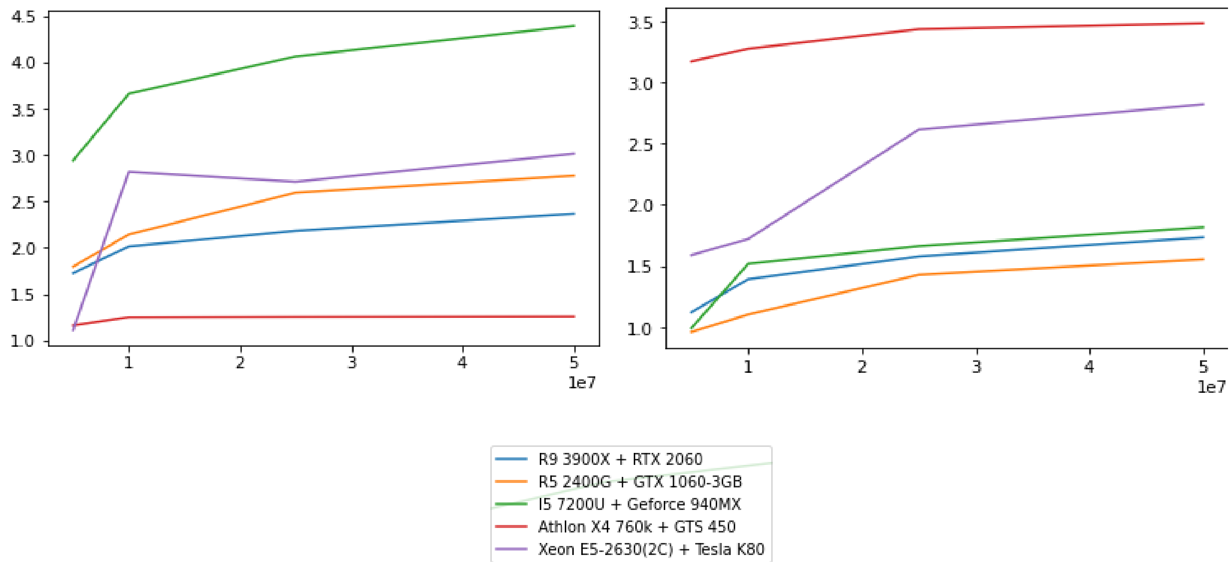

Fig. 4. Speedup coefficients on heterogeneous system

References

1. Chen, Tianqi & Li, Mu & Li, Yutian & Lin, Min & Wang, Naiyan & Wang, Minjie & Xiao, Tianjun & Xu, Bing & Zhang, Chiyuan & Zhang, Zheng. (2015). MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems.

2. J. E. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," in *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66-73, May-June 2010, doi: 10.1109/MCSE.2010.69.

3. E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, 2016, pp. 1-4, doi: 10.1109/FPL.2016.7577314.

4. Van Werkhoven, Ben & Maassen, Jason & Seinstra, Frank & Bal, Henri. (2014). Performance models for CPU-GPU data transfers. Proceedings - 14th

IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014. 10.1109/CCGrid.2014.16.

5. Y. Kim, P. Mercati, A. More, E. Shriver and T. Rosing, "P4: Phase-based power/performance prediction of heterogeneous systems via neural networks," 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, 2017, pp. 683-690, doi: 10.1109/ICCAD.2017.8203843.

6. GPU-Accelerated Applications [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf

7. NVidia Turing GPU Architecture [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

8. Yang, Canqun & Wang, Feng & Du, Yunfei & Chen, Juan & Liu, Jie & Yi, Huizhan & Lu, Kai. (2010). Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. Proceedings - IEEE International Conference on Cluster Computing, ICCC. 19-28. 10.1109/CLUSTER.2010.12.

9. J. Hestness, S. W. Keckler and D. A. Wood, "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors," 2015 IEEE International Symposium on Workload Characterization, Atlanta, GA, 2015, pp. 87-97.

10. Soyata, T., 2018. GPU Parallel Program Development Using CUDA. New York: CRC Press.

11. CUDA Refresher: Reviewing the Origins of GPU Computing [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

AUTHORS

Volodymyr Rusinov – student, Department of Computer Engineering, National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”.

E-mail: VRusinovIO51@office365.fiot.kpi.ua

Andrii Antoniuk (supervisor) – associate professor, PHd, Department of Computer Engineering, National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”.

E-mail: ant5298g@gmail.com