This is a post-peer-review, pre-copyedit version of an article published in **2019 International Conference on High Performance Computing & Simulation (HPCS)**. The final authenticated version is available online at: https://doi.org/10.1109/HPCS48598.2019.9188060

Auto-tuning methodology for configuration and application parameters of hybrid CPU+GPU parallel systems based on expert knowledge

Paweł Czarnul and Paweł Rościszewski
Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology
Gdańsk, Poland
pczarnul@eti.pg.edu.pl, pawel.rosciszewski@pg.edu.pl

***Abstract*—Auto-tuning of configuration and application parameters allows to achieve significant performance gains in many contemporary compute-intensive applications. Feasible search spaces of parameters tend to become too big to allow for exhaustive search in the auto-tuning process. Expert knowledge about the utilized computing systems becomes useful to prune the search space and new methodologies are needed in the face of emerging heterogeneous computing architectures. In this paper we propose an auto-tuning methodology for hybrid CPU/GPU applications that takes into account previous execution experiences, along with an automated tool for iterative testing of chosen combinations of configuration, as well as application-related parameters. Experimental results, based on a parallel similarity search application executed on three different CPU+GPU parallel systems, show that the proposed methodology allows to achieve execution times worse by only up to 8% compared to a search algorithm that performs a full search over combinations of application parameters, while taking only up to 26% time of the latter.**

***Index Terms*—performance optimization, hybrid high performance computing, auto-tuning, multi-core CPU, GPU**

I. Introduction

Solutions to many practical computational problems can be implemented as hybrid parallel applications [1], which allows for increased performance due to utilizing heterogeneous computing resources and combining multiple levels of parallelization. Unfortunately, such approaches lead also to multiplicity of parameters for which values have to be selected for a certain application execution. These parameters can be divided into two groups: *configuration parameters* dependent on the utilized hardware and *application parameters* related to the specific parallel implementation.

Since *configuration parameters* are related to the utilized hardware, the specific parameters and their feasible values depend on the architecture and components of the computing system. For example, one of the *configuration parameters* in a multi-core CPU is the number of used threads and the possible values are integer numbers ranging from 1 to the maximum number of threads allowed by the system. Similarly, an example of a *configuration parameter* in a multi-node cluster is the number of used nodes [2], in a multi-GPU system - the number of used GPUs [3]. In a system that utilizes GPUs, grid configuration has to be configured [4]. If Intel Xeon Phi accelerators are used, setting the proper page size for memory allocation can be beneficial [24]. *Configuration parameters* can also be related to the software solutions used in the computing system. For example, when using Hadoop, one might configure the maximum number of used machines or memory limit for each map and reduce tasks [5].

On the other hand, the feasible space of *application parameters* may depend on the used parallelization paradigm. For example the number of workers used in each iteration, task allocation and granularity of problem decomposition in the *master/worker (master/slave)* paradigm, number of tasks per each core in the *SPMD* paradigm [6], incurred latency in the *pipelining* paradigm [7] or execution tree depth and thread limit for the *divide and conquer* paradigm [8]. Paper [9] proposes tuning complex parallel applications using a combination of different parallel programming paradigms such as master-worker and pipelining. Resources are assigned properly to application components so as to achieve a performance increase.

Auto-tuning of these parameters through executing a small, yet representative probe of the computational problem with multiple parameter configurations can lead to significant performance improvements. Since interdependences between the multiple parameters can be non-trivial, searching through the whole parameter space is often required, instead of determining optimal values of the parameters independently [1]. However, the size of the search space, along with significant execution times of the probing executions, often make exhaustive search infeasible.

In this paper, we focus on the *configuration and application* parameters of a parallel similarity measure computation application executed in a hybrid CPU/GPU system. We propose an auto-tuning methodology that reduces the parameter search space based on expert knowledge about the components of the considered computing system. This allows to significantly reduce the auto-tuning execution time, while still finding sensible sub-optimal parameter configurations. The outline of the paper is as follows. We discuss related work in Section II, provide problem statement in III and describe the proposed

methodology in Section IV. Conducted experiments and their results are presented in Section V, while the paper summary and future work directions are given in Section VI.

II. Related Work and Motivations

In recent works, several approaches to parameter auto-tuning in high performance computing have been adopted. The following can be distinguished:

1) full space enumeration (exhaustive search) – very time consuming, unrealistic for many practical problems;
2) combinatorial search - testing only a subset of combinations chosen by a combinatorial search algorithm (random, hierarchical [10], direct search [11], using random forest feature importance [12], etc.);
3) modeling and simulation for fast execution time evaluation (for example modeling and simulation for exploring power/time trade-off of parallel deep neural network training [3]);
4) search space pruning using expert knowledge or domain-specific constraints (for example reducing the search space from above 524 million to over 379 thousand combinations in [13]).

It should be noted, that the aforementioned approaches can be mixed. For example, expert knowledge can be used to prune the search space, combinatorial optimization performed on selected variables and full space enumeration for the rest. Combinatorial search can be combined with machine learning such as proposed in [14] with Simulated Annealing for system configuration search and a machine learning trained prediction model for assessment of performance of a given configuration, prediction of performance was obtained using Boosted Decision Tree Regression [15].

The auto-tuning methodology proposed in this paper comes with a tool for iterative generation and testing of application parameter combinations. Similar tools are available for specific parallel programming platforms. For example, runtime parameters of OpenMPI programs can be optimized using the Open Tool for Parameter Optimization (OTPO) [16]. In the paper, the tool is used for optimizing InfiniBand communication parameters. Analogously to our solution, the optimization process in OTPO requires configuration of the parameters and their potential values to be tested. The parameter file includes default values, lists of possible values, start and end value and traversal method. The latter is similar to `<stepmode>` in the tool proposed in this paper and specifies the method to traverse the range of variables for the parameter. In the first version, OTPO includes one method: "increment". It is similar to the LINEAR change in the tool proposed in this paper, which supports also advanced traversal methods, such as BASICEXP and SEARCHEXP. More importantly, the proposed tool allows to auto-tune application-specific parameters, as well as configuration parameters related to GPU execution.

The latter have been subject to tuning in [17]. The authors used expert knowledge about the GPU architecture and parameter constraints to tune the grid size values for chosen linear algebra micro-benchmarks. This approach allowed to find configurations characterized by high GPU occupancy.

Auto-tuning of MPI *configuration parameters* has also been proposed in [18]. Specifically, the authors proposed an MPI parameters plugin allowing to find sets of parameters such as eager message size limit for using the eager or randezvous protocol, buffer size, bulk transfer parameters, affinity, polling or non-polling when waiting for a message, polling interval, faster communication vs larger memory trade-off related parameter. Search strategies for particular parameters can include exhaustive search, heuristic strategies such as GDE3 or random based. Expert knowledge is employed for tuning the eager limit based on sizes of exchanged messages. From the point of view of this work, that contribution is limited to *configuration parameters* only and does not include GPU-oriented parameters.

Auto-tuning of stencil computations has been considered in [19] for a number of computing architectures, including Intel Clovertown, AMD Barcelona, Sun Victoria Falls, IBM QS22 PowerCell 8i and NVIDIA GTX280. The tuned parameters included *configuration parameters* such as core block size, thread block size, register block size, DMA size and cache bypass, as well as an application parameter that determined the chunk size of problem decomposition. Heuristics were used for constraining search space in the auto-tuning process.

Expert knowledge is taken into account by a static code analyzer for CUDA programs proposed in [20], which allows for automatic identification of computational intensity. Based on kernel's GPU occupancy, the proposed approach allowed to reduce the configuration search space from 5,120 to 640 due to thread settings suggestions, without requiring any program runs.

Application parameter tuning was investigated in [21] for an application of GS2 for studying low-frequency turbulence in magnetized plasma. Parameters such as grid point density, energy grid and number of nodes were considered in analysis. Performance variablity was studied and a Parallel Rank Ordering algorithm was presented for tuning *application parameters*.

A study aiming specifically for similarity search algorithms in [22] distinguished the following performance-related *application parameters* that can be subject to tuning: shortcutting flag, number of vectors in block, number of query partitions, query batch size and data striping size in dimensions. A *configuration parameter* has also been considered, namely the number of threads per partition. An auto-tuner based on simulated annealing provided up to two orders of magnitude improvement over the worst settings. Hyperparameters of a similarity search algorithm have also been tuned for approximate nearest neighbor (ANN) search in [23], where indexing methods are tuned based on randomized space-partitioning trees. The approach is competitive with existing index building approaches in terms of achieved recall while being significantly faster.

Auto-tuning at a more global operating system level for a set of applications is presented in [24]. Tuning is transparent

to the user and is performed at the Linux kernel level for system wide performance at runtime. Examples such as a bzip2 compression and video processing applications were used as case studies for demonstration of approach benefits in an environment with an Intel Core 2 Quad CPU and the Linux operating system.

There is a need for methodologies and tools that will take into account both *application* and *configuration parameters* and expert knowledge for hybrid applications. This paper proposes an approach that allows auto-tuning by exploiting expert knowledge and experiences from optimization of *configuration parameters* from previous works and heuristic multidimensional search through combinations of *application parameters*. This approach allows to achieve very good results in terms of obtained application execution times much faster than enumeration of parameter combinations.

III. Problem statement

Firstly, we assume that there are several parameters that can be divided into the following two groups:

1) configuration parameters $CP = \{cp_1, \ldots, cp_{|CP|}\}$ – set before/at application runtime such as:
   - number of threads run on CPU(s),
   - number of threads run on accelerators,
   - thread affinity,
   - grid configuration for GPU(s),
2) application parameters $AP = \{ap_1, \ldots, ap_{|AP|}\}$, e.g. buffer sizes in the application.

Assuming parallel application PA, input data D, parallel system PS, the goal is to minimize the execution time et by finding proper combination of values of both *configuration* and *application parameters*:

$$\min_{AP,CP} et(PA, D, PS, AP, CP) \tag{1}$$

IV. Proposed Auto-tuning Methodology

A. Design and implementation of an auto-tuning tool

The proposed tool iteratively generates combinations of *configuration* and *application parameters* as environment variables and subsequently invokes a given application in an environment in which those variables are available. In the tests, values of such environment variables were passed to the application as command line arguments but could also be read directly from within an application. The tool reads a configuration file with the following syntax:

```
CPU_THREAD_COUNT <min> <max> <stepmode> \
<startval>
GPU_COUNT <min> <max> <stepmode> <startval>
GPU_THREADS_IN_BLOCK_COUNT <min> <max> \
<stepmode> <startval>
APP_PARAM_0 <min> <max> <stepmode> \
<startval>
APP_PARAM_1 <min> <max> <stepmode> \
<startval>
COMMAND <appcommand>
TIME_OUTPUT <timeparsecommand>
```

where values of parameters i.e. <min>, <max> and <default> are minimum, maximum and default i.e. starting values for a given parameter. <stepmode> denotes how the tool changes the value of parameter p and can be one of the following in the current version of the tool:

LINEAR – a linear increase of p by a constant step (of 1 by default) from pmin to pmax or a linear decrease of p by a constant step (of 1 by default) from pmax to pmin.

BASICEXP – an increase by multiplication of a current p value by 2 from pmin to pmax or a decrease by division of a current p value by 2 from pmax to pmin

SEARCHEXP – searching for the best value of a single parameter p with the following algorithm:

```
int direction=-1;
int directionchanged=0;
for(int p=pstart;p>=pmin
&& p<=pmax;)
{
  double time=
  runcommandgettime(...,p,...);
  if (time<besttime) {
    besttime=time;
    bestp=p;
  } else {
    if (directionchanged)
      break;
    direction=1;
    directionchanged=1;
  }
  if (direction==-1)
    p/=2;
    else
    p*=2;
}
```

<startval> denotes the starting value used by the tool – it is assumed to be between <min> and <max>.

The meaning of the parameters is as follows:

CPU_THREAD_COUNT – the number of computational threads running on CPU(s),

GPU_COUNT – the number of GPUs used,

GPU_THREADS_IN_BLOCK_COUNT – the number of threads per block, it is assumed that the number of thread blocks is obtained by the application using this variable and input data size,

APP_PARAM_X – parameter understood by the application,

COMMAND – invocation command for the application

TIME_OUTPUT – command for parsing application output to find out real execution time (the application

can measure a part of its execution optimized by the tool).

The proposed auto-tuning algorithm presented in Sections IV-B and IV-C is further called PCPARSEARCH.

B. Approach to auto-tuning of configuration parameters

The following steps are proposed for auto-tuning of *configuration parameters*, each of which searches for a value of a given parameter for which the smallest execution time is obtained, given fixed values of the others. This causes the approach to be heuristic rather than potentially extremely time-consuming search for a configuration out of all combinations of parameters.

1) GPU count `g` – checking for the `g` giving the smallest execution time. The number of computational CPU threads is equal to `cputhreadcountmax-2*g` assuming Hyperthreading so that physical cores are left out for management of GPUs. The starting grid configuration is used and starting values of *application parameters* are used. `bestg` is obtained and used in the next steps.
2) number of GPU threads per block `tpb` – checking for the `tbp` giving the smallest execution time. The number of computational CPU threads is equal to `cputhreadcountmax-2*bestg`. The starting grid configuration is used and starting values of *application parameters* are used. `besttpb` is used for subsequent computations.
3) number of computational CPU threads `cct` (apart from CPU threads managing computations on GPUs) – checking for the `cct` giving the smallest execution time. Starting values of *application parameters* are used.

Further *configuration parameters* can be used e.g. the number of CUDA streams as investigated in [25]. For the purpose of this work, the hybrid application used overlapping communication and computations using 2 CUDA streams per device. Additionally, thread affinity may have an impact on performance. Specifically, we may expect that balanced may typically be preferred for parallel computations on Intel Xeon Phis [26] with potential checks for scatter and compact while scatter vs compact should be checked for host CPUs. In the following experiments, default thread affinity was used.

C. Approach to auto-tuning of application parameters

For auto-tuning of *application parameters*, the following algorithm is used that traverses the search space in each dimension in two directions, as shown in Figure 1.

D. Reference fuller search algorithms

The proposed heuristic algorithm has been compared to two algorithms performing fuller searches of the combined configuration+application parameters, albeit still not exhaustive search.

The first, fuller search algorithm, that we will further refer to as SEMIFULLSEARCH1, executes the following steps:

1) GPU count `g` – checking for the `g` giving the smallest execution time. The number of computational CPU threads is equal to `cputhreadcountmax-2*g`. The starting grid configuration is used and starting values of *application parameters* are used. `bestg` is obtained and used in the next steps.
2) number of computational CPU threads `cct` (apart from CPU threads managing computations on GPUs) – checking for the `cct` giving the smallest execution time. Starting values of grid *configuration* and *application parameters* are used.
3) Performs a full search over all combinations of *application parameters* together with all considered values of grid configurations. The latter stem from different values of the number of threads within a block since we assume that the number of blocks stems from input data size and the block thread count. The rationale behind this assumption is that various combinations of *application parameters* and grid configurations may potentially result in various minima.

The second, semi full search algorithm, that we will further refer to as SEMIFULLSEARCH2, executes the steps optimizing *configuration parameters* exactly as the algorithm shown in Section IV-B. This is followed by performing a full search over all combinations of *application parameters* together with all previously obtained *configuration parameters* set.

V. Experiments

A. Testbed Environments

For experiments, we used three modern multicore CPU(s) + GPUs systems:

- eagle (HPC workstation, 2x Xeon CPU + 2x GTX GPU),
- des19 (poweful desktop, 2x i7 CPU + 2x GTX GPU),
- apl11 (HPC server, 2x Xeon CPU +2x Tesla GPU).

Specifications of the systems are listed in Table I.

B. Testbed Application

For experiments performed in this paper, we used a hybrid parallel OpenMP+CUDA application for parallel computations of similarity measures among pairs of multidimensional vectors [27], [28]. Paper [27] presented manual steps of optimization of this application by finding good values of both configuration and *application parameters*, performing full search for the latter. Specifically, the application uses 2 *application parameters* that denote size of the so-called first vector batch size and second vector batch size respectively. Each vector from the first batch size is compared to each vector of the second batch size. The same application was also optimized in a parallel environment with Intel Xeon CPUs and Intel Xeon Phi coprocessors [26]. This paper presents automation of these optimization steps, applicable to a multi CPU + multi GPU environment.

The following *configuration parameters* were used for performing tests for 32768 vectors with 4096 vector size on system 1:

```
CPU_THREAD_COUNT 0 32 BASICEXP 1
GPU_COUNT 1 2 LINEAR 1
```

```
for(i=0;i<appparamcounter;i++) {
  current[i]=appparamstart[i];
  bestappparamvals[i]=current[i];
  sprintf(appparamvals[i],"%d",current[i]);
}
while (1) {
  int cont=0;
  for(i=0;i<appparamcounter;i++) {
    // for each of the parameters perform 1 step and go to the next one
    if (!appparamstop[i]) {
      cont=1; // change in at least one dimension
      if (appparamsteptype[i]==STEP_LINEAR) current[i]--;
      if (appparamsteptype[i]==STEP_EXP) current[i]/=2;
      if (appparamsteptype[i]==STEP_SEARCH_EXP)
        if (appparamdirection[i]==-1)
          current[i]/=2; else current[i]*=2;
      sprintf(appparamvals[i],"%d",current[i]);

      double time=runcommandgettime(bestgpucount,bestcputhreadcount,
      bestgputhreadsinblockcount,appparamnames,appparamvals,
      appparamcounter,command,timeoutput);

      if (time<besttime) {
        besttime=time;
        bestappparamvals[i]=current[i];
      } else {
        if (appparamsteptype[i]==STEP_SEARCH_EXP) {
        // in this mode change direction
        // and if direction has already been changed then terminate
          if (appparamdirectionchanged[i])
            appparamstop[i]=1;
          else {
            appparamdirection[i]=1;
            appparamdirectionchanged[i]=1;
            current[i]=appparamstart[i]; // reset the initial value
            }
          }
        }
      if ((current[i]==appparammin[i]) && (appparamdirection[i]==-1)) { // change
        // direction
          appparamdirection[i]=1;
          appparamdirectionchanged[i]=1;
          current[i]=appparamstart[i]; // reset the initial value
        }
        if ((current[i]==appparammax[i]) && (appparamdirection[i]==1)) // terminate
          appparamstop[i]=1;
      }
    }
    if (!cont) break;

  }
}
```

Fig. 1. Auto-tuning of application parameters in PCPARSEARCH

TABLE I
TESTBED CONFIGURATIONS

Testbed	1	2	3
CPUs	2 x Intel Xeon CPU E5-2620v4 @ 2.10GHz	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz	2 x Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
CPUs – total number of physical/logical cores	16/32	4/8	12/24
System memory size (RAM) [GB]	128	16	64
GPUs	2 x NVIDIA GTX 1070	2 x NVIDIA GTX 1060	2 x NVIDIA Tesla K20m
GPUs – total number of CUDA cores	2 x 2048	2 x 1280	2 x 2496
GPU Compute capability	6.1	6.1	3.5
GPU memory size [MB]	2 x 8192	2 x 6144	2 x 5120
Operating system	Linux 4.15.0-36-generic	Linux 4.4.155-68-default	Linux 3.10.0-862.9.1.el7.x86_64
Compiler/version	Cuda compilation tools, release 9.1, V9.1.85, gcc 7.3.0	Cuda compilation tools, release 9.1, V9.1.85, gcc 4.8.5	Cuda compilation tools, release 9.1, V9.1.85, gcc 4.8.5

```
GPU_THREADS_IN_BLOCK_COUNT 64 1024 \
SEARCHEXP 128
APP_PARAM_0 32 4096 SEARCHEXP 128
APP_PARAM_1 32 4096 SEARCHEXP 128
COMMAND ./program-onegpumemalloc-\
gpu-comm-opt-overlapping1 \$GPU_COUNT \
\$CPU_THREAD_COUNT 32768 4096 \
\$APP_PARAM_0 \$APP_PARAM_1 \
\$GPU_THREADS_IN_BLOCK_COUNT
TIME_OUTPUT grep FINAL \
| cut -d' ' -f 10
```

For the other systems, the range of `CPU_THREAD_COUNT` varied from 0 to the number of logical CPUs in the system.

C. Results and Discussion

Firstly, after running SEMIFULLSEARCH1 and SEMIFULLSEARCH2, we have noticed, as shown in Table II, that the best configurations have been obtained for the same CUDA grid configurations. While it does not have to be the case in general, we have selected SEMIFULLSEARCH2 as the faster algorithm as a baseline for comparison of relative performance and execution time of PCPARSEARCH.

In order to assess the range of application execution times for various combinations of *configuration* and *application parameters* and the three systems, Table III presents the following metrics:

1) best application running times,
2) worst application running time across parameter sets (for reference on how search improves compared to worst case scenario),
3) average application running time across parameter sets (for reference on how search improves compared to average case scenario).

As an example, Figure 2 presents successive, descreasing application execution times of configurations tested by PCPARSEARCH in successive time steps. Figures 3, 4 and 5 present relative execution times of the best application configurations obtained by PCPARSEARCH compared to SEMIFULLSEARCH2, for system 1, 2 and 3 respectively, averaged from 5 runs. The same figures show relative execution time of PCPARSEARCH compared to SEMIFULLSEARCH2. It can be seen that for the analyzed application and 3 various CPU+GPU systems, PCPARSEARCH found application execution times worse only up to 8% compared to SEMIFULLSEARCH2 while the time to find the solution was only up to 26% of the execution time of SEMIFULLSEARCH2. This makes it an interesting alternative to fuller search algorithms at a small performance cost.

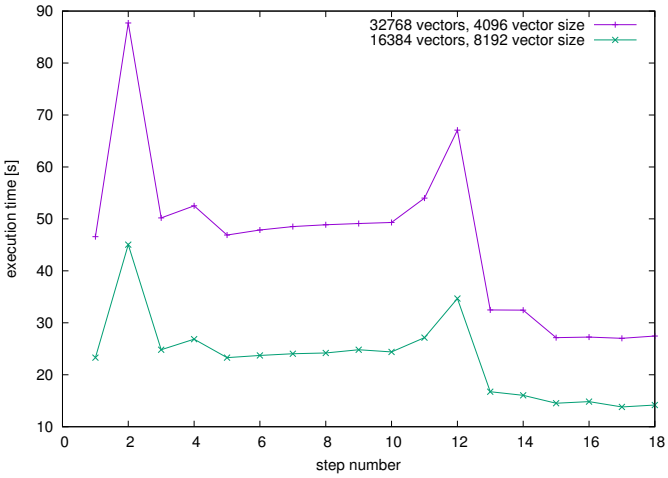

Fig. 2. System 1 – application execution times in successive steps of the auto-tuning algorithm

VI. SUMMARY AND FUTURE WORK

In the paper, we presented a heuristic algorithm for searching for combinations of *configuration* and *application parameter* sets that aim at minimization of parallel application execution times on modern hybrid CPU+GPU systems. As the

TABLE II
OBTAINED CONFIGURATIONS

Number of vectors / vector size	system	auto-tuning algorithm	number of GPUs	number of computational CPU threads	number of threads per block	first batch size	second batch size
32768 / 4096	1	SEMIFULLSEARCH1/2	2	28	128	1024	128
32768 / 4096	1	PCPARSEARCH	2	28	128	1024	512
16384 / 8192	1	SEMIFULLSEARCH1/2	2	16	128	1024	64
16384 / 8192	1	PCPARSEARCH	2	28	128	1024	512
32768 / 4096	2	SEMIFULLSEARCH1/2	2	4	128	1024	256
32768 / 4096	2	PCPARSEARCH	2	4	128	1024	1024
16384 / 8192	2	SEMIFULLSEARCH1/2	2	4	128	1024	128
16384 / 8192	2	PCPARSEARCH	2	4	128	1024	512
32768 / 4096	3	SEMIFULLSEARCH1/2	2	20	256	1024	32
32768 / 4096	3	PCPARSEARCH	2	20	256	512	512
16384 / 8192	3	SEMIFULLSEARCH1/2	2	20	256	1024	32
16384 / 8192	3	PCPARSEARCH	2	20	256	512	256

TABLE III
EXECUTION TIMES

Number of vectors / vector size	system	best execution time [s]	average execution time [s]	worst execution time [s]
32768 / 4096	1	26.8	56.4	461.9
16384 / 8192	1	12.6	29.3	153.2
32768 / 4096	2	37.5	75.1	427.4
16384 / 8192	2	18.7	39.5	164.1
32768 / 4096	3	73.9	120.5	776.6
16384 / 8192	3	34.2	55.1	281.7

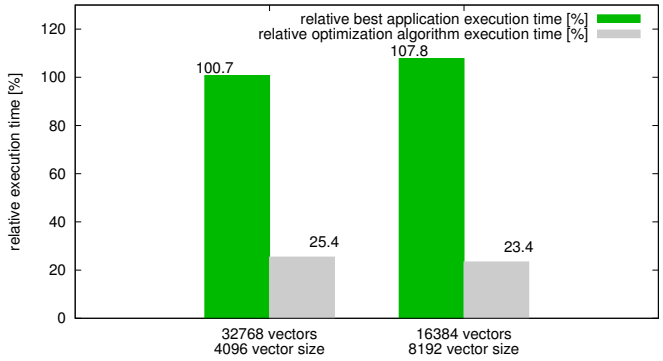

Fig. 3. System 1 – relative application and auto-tuning algorithm execution times

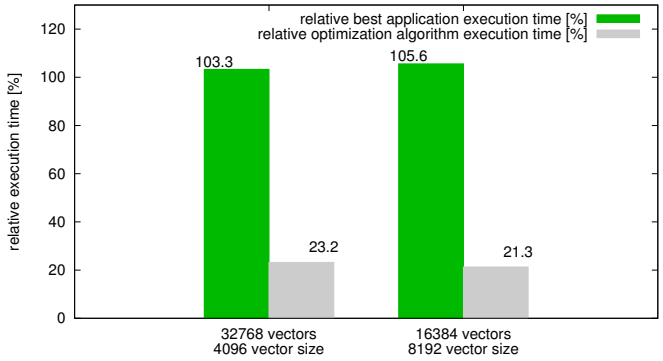

Fig. 4. System 2 – relative application and auto-tuning algorithm execution times

problem is computationally demanding, we showed that the algorithm offers application execution times worse by only up to 8% compared to an algorithm that performs a full search over combinations of *application parameters* while taking only up to 26% of auto-tuning time of the latter for an application computing similarity among pairs of multidimensional vectors. Tests have been performed on three different CPU+GPU parallel systems.

The presented work will be extended in the future in several ways:

1) other parallel applications will be tested,
2) additional tools such as nvprof etc. can be used to predetermine better starting values for GPU related parameters,
3) more HPC systems will be tested, including larger numbers of GPUs and various ratios of CPU/GPU performance.

ACKNOWLEDGMENTS

Work was supported partially by the Polish Ministry of Science and Higher Education.

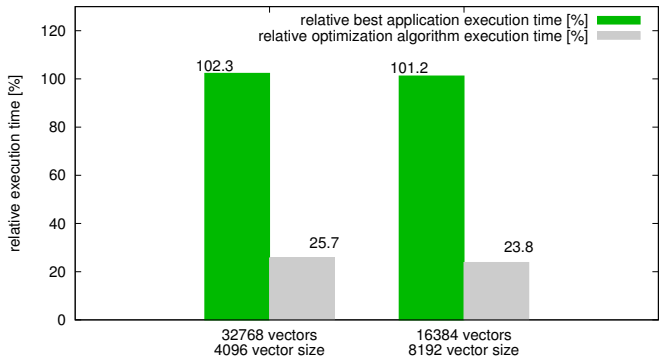

Fig. 5. System 3 – relative application and auto-tuning algorithm execution times

REFERENCES

[1] Pawel Rosciszewski. Optimization of hybrid parallel application execution in heterogeneous high performance computing systems considering execution time and power consumption. *CoRR*, abs/1809.07611, 2018.

[2] Paweł Czarnul and Paweł Rościszewski. Optimization of Execution Time under Power Consumption Constraints in a Heterogeneous Parallel System with GPUs and CPUs. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314, pages 66–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] Paweł Rościszewski. Modeling and Simulation for Exploring Power/-Time Trade-off of Parallel Deep Neural Network Training. *Procedia Computer Science*, 108:2463–2467, 2017.

[4] Pawel Rosciszewski, Pawel Czarnul, Rafal Lewandowski, and Marcel Schally-Kacprzak. Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. *Concurrency and Computation: Practice and Experience*, 28(9):2586–2607, 2016.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[6] Ronal Muresano, Dolores Rexachs, and Emilio Luque. Methodology for Efficient Execution of SPMD Applications on Multicore Environments. pages 185–195. IEEE, 2010.

[7] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[8] Paweł Czarnul. Parallelization of Divide-and-Conquer Applications on Intel Xeon Phi with an OpenMP Based Framework. In Jerzy Swiatek, Leszek Borzemski, Adam Grzech, and Zofia Wilimowska, editors, *ISAT (3)*, volume 431 of *Advances in Intelligent Systems and Computing*, pages 99–111. Springer, 2015.

[9] J. A. Guevara, E. Cesar, J. Sorribes, A. Moreno, T. Margalef, and E. Luque. A performance tuning strategy for complex parallel application. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 103–110, Feb 2010.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[11] V. Tabatabaee, A. Tiwari, and J.K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. pages 57–57. IEEE, 2005.

[12] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-wei Liao. Machine Learning-Based Configuration Parameter Tuning on Hadoop System. In *2015 IEEE International Congress on Big Data (BigData Congress)*, pages 386–392, June 2015.

[13] Roman Wyrzykowski, Łukasz Szustak, and Krzysztof Rojek. Parallelization of 2d MPDATA EULAG algorithm on hybrid architectures with GPU accelerators. *Parallel Computing*, 40(8):425–447, August 2014.

[14] S. Memeti and S. Pllana. Combinatorial optimization of work distribution on heterogeneous systems. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 151–160, Aug 2016.

[15] Suejb Memeti and Sabri Pllana. A machine learning approach for accelerating dna sequence analysis. *The International Journal of High Performance Computing Applications*, 32(3):363–379, 2018.

[16] Mohamad Chaarawi, Jeffrey M. Squyres, Edgar Gabriel, and Saber Feki. A Tool for Optimizing Runtime Parameters of Open MPI. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205, pages 210–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[17] Nhat-Phuong Tran, Myungho Lee, and Jaeyoung Choi. Parameter based tuning model for optimizing performance on gpu. *Cluster Computing*, 20(3):2133–2142, Sep 2017.

[18] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of mpi applications using ptf. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, pages 31–38, New York, NY, USA, 2016. ACM.

[19] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, page 4. IEEE/ACM, 2008.

[20] Robert V. Lim, Boyana Norris, and Allen D. Malony. Autotuning gpu kernels via static and predictive analysis. In *ICPP*, pages 523–532. IEEE Computer Society, 2017.

[21] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 57–57, Nov 2005.

[22] Bugra Gedik. Auto-tuning similarity search algorithms on multi-core architectures. *International Journal of Parallel Programming*, 41(5):595–620, 2013.

[23] Elias Jääsaari, Ville Hyvönen, and Teemu Roos. Efficient autotuning of hyperparameters in approximate nearest neighbor search. In Qiang Yang, Zhi-Hua Zhou, Zhiguo Gong, Min-Ling Zhang, and Sheng-Jun Huang, editors, *PAKDD (2)*, volume 11440 of *Lecture Notes in Computer Science*, pages 590–602. Springer, 2019.

[24] Thomas Karcher and Victor Pankratius. Run-time automatic performance tuning for multicore applications. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 3–14, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] Pawel Czarnul. Benchmarking overlapping communication and computations with multiple streams for modern gpus. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Communication Papers of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznań, Poland, September 9-12, 2018.*, pages 105–110, 2018.

[26] Paweł Czarnul. Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors. *International Journal of Parallel Programming*, 45(5):1091–1107, Oct 2017.

[27] Paweł Czarnul. Parallelization of large vector similarity computations in a hybrid cpu+gpu environment. *The Journal of Supercomputing*, 74(2):768–786, Feb 2018.

[28] P. Czarnul, P. Rościszewski, M. Matuszek, and J. Szymański. Simulation of parallel similarity measure computations for large data sets. In *2015 IEEE 2nd International Conference on Cybernetics (CYBCONF)*, pages 472–477, June 2015.