Noname manuscript No.
(will be inserted by the editor)

Parallelization of large vector similarity computations in a hybrid CPU+GPU environment

Paweł Czarnul

Received: date / Accepted: date

Abstract The paper presents design, implementation and tuning of a hybrid parallel OpenMP+CUDA code for computation of similarity between pairs of a large number of multidimensional vectors. The problem has a wide range of applications and consequently its optimization is of high importance, especially on currently widespread hybrid CPU+GPU systems targeted in the paper. The following are presented and tested for computation of all vector pairs: tuning of a GPU kernel with consideration of memory coalescing and using shared memory, minimization of GPU memory allocation costs, optimization of CPU-GPU communication in terms of size of data sent, overlapping CPU-GPU communication and kernel execution, concurrent kernel execution, determination of best sizes for data batches processed on CPUs and GPUs along with best GPU grid sizes. It is shown that all codes scale in hybrid environments with various relative performances of compute devices, even for a case when comparisons of various vector pairs take various amounts of time. Tests were performed on two high performance hybrid systems with: 2 x Intel Xeon E5-2640 CPU + 2 x NVIDIA Tesla K20m and latest generation 2 x Intel Xeon CPU E5-2620 v4 + NVIDIA's Pascal generation GTX 1070 cards. Results demonstrate expected improvements and beneficial optimizations important for users incorporating such types of computations into their parallel codes run on similar systems.

Keywords hybrid parallelism · OpenMP · CUDA · parallel programming · optimization

P. Czarnul
Dept. of Computer Architecture
Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology,
Poland
E-mail: pczarnul@eti.pg.gda.pl

1 Introduction

In the high performance computing landscape, more and more modern compute devices appear, including multicore CPUs such as Xeon series v4 devices with even 24 physical and 48 logical cores in the Intel Xeon E7-8894 v4 CPU, Pascal series GPUs and Knight Landing Xeon Phi many core coprocessors. Consequently, hybrid systems including these devices have become widespread. For instance, in the TOP500 list, two of the first three most powerful systems include an Intel Xeon+Xeon Phi system and Opteron+NVIDIA GPUs. CPU+GPU hybrid architectures are now also widespread among workstations and home PCs, many of which include multicore CPUs and GPUs capable of running CUDA and/or OpenCL codes.

The goal of this paper is to design a parallelized algorithm and present an efficient OpenMP+CUDA implementation for computation of similarity between pairs from a large set of multidimensional vectors, for the aforementioned hybrid CPU+GPU systems. This task is a challenge because of various performances of compute devices and consequently need for proper data partitioning, need for load balancing and management of computations on CPU cores and GPUs, potential optimizations such as minimization of host to device (GPU) communication across PCI Express, minimization of GPU memory allocation, overlapping CPU-GPU communication and computations on the GPU, concurrent kernel execution. This research follows consideration of the same problem for a hybrid CPU+Xeon Phi coprocessor environment presented in [5]. In comparison, this work dives into details of an environment with GPUs which require a different programming model and ways of optimization.

2 Related work

Applications of similarity computations are very broad and are often based on comparison of components such as images, words, text excerpts, documents (text, web pages) and various objects [24,22]. These components can be encoded as multidimensional vectors. Hence, the problem of finding similarities between a large number of multidimensional vectors using desired similarity metrics is of high importance, especially in areas where drawing conclusions from big data sets has become possible through emergence of new high performance computing devices as multicore CPUs and GPUs.

There are numerous applications of similarity computations including: providing hints to the user during web searching [22], finding data related to the current clinical case in medicine [17], disambiguation of entities [18], document clusterization [25], audio recognition [19], handwritten word image retrieval [23], linguistic information classification [21]. Paper [16] discusses benefits from parallelization of chemical similarity calculation and presents an algorithm for all-vs-all Tanimoto matrix calculation and nearest neighbor search using GPUs demonstrating considerable benefits over CPU based approaches.

In many papers, the problem of all pair similarity search is analyzed which is to obtain such pairs for which the value of a similarity metric is above a given threshold which is often implemented by algorithm level optimizations [4,9,1,3,27,15].

Various environments were considered for execution and optimization of similarity related research– typically this includes CPUs [4,15]. MapReduce has been used for computing similarity between words or objects on the Web [20,9,1]. Several works discuss using CPU and GPU environments. Paper [13] discusses parallelization of link based similarity measure computations on a GPU in comparison to a CPU based approach. Because of GPU memory limitations, matrix tiling is applied so that sub-matrices fit into GPU global memory. For optimization of data transfer between RAM and GPU memory, Doubly Compressed Sparse Row (DCSR) is used. For SimRank, rvs-SimRank and Inter-connection computations, CPU is better for a small number of iterations, but falls behind a GPU for larger numbers of iterations. Tiling is also used in paper [10] for parallel comparisons of bug reports in the context of automated bug triaging. 8192 reports are compared to 8192 reports after which results are transferred back to the system memory. A sequential CPU based version is compared to a parallel GPU version with the latter showing of up to over 80x speed-up compared to the former for parallel cosine similarity computations. Regression lines are found for tested data points to find projected results for even higher numbers of reports. Tesla K40 was used and compared to an Intel Xeon E5-2660v3 CPU. Paper [14] presents an approach for hybrid CPU/GPU parallel processing of similarity search in the context of content-based multimedia retrieval. Specifically, Signature Quadratic Form Distance is used. A CPU and GPU equipped workstation outperforms a 48-node NUMA server. In terms of implementation, a GPU is used mainly for computation of similarity between a query and a database entry. A block is sent to a GPU with the query and N signatures. Launching a kernel is asynchronous which allows spawning work on the CPU side as well. Furthermore, computation of a distance is also performed in parallel with use of local memory within a GPU. Combined CPU/GPU parallel similarity computations are also discussed in paper [12] in the context of ontology matching. The authors consider and address issues such as limited data types on the GPU side as well as limited memory. The former is solved using arrays and the latter by partitioning ontologies into smaller parts. An implementation uses a job queue from which work is distributed over a GPU and a CPU. The authors presented smaller execution times for a GTX660 compared to an Intel i5-2500 CPU and further improvement in a hybrid environment using both the GPU and the CPU. Next, distance computations can also be found in works in the context of content based medical image retrieval. In paper [26] authors present approximately 10x smaller execution times of a parallel GPU code on an NVIDIA GeForce 9500GT compared to an Intel E5500 CPU for X-ray images with sizes from 32x32 up to 800x800.

The problem considered in this work was also analyzed previously in [8] in the context of models of the processing algorithm and the hardware on which

the algorithm was run. Both were created in the MERPSYS environment that allows simulation of parallel application execution on large scale cluster and volunteer based systems and returns predicted application execution time, energy consumption and probability of successful execution. In [8] simulation results were validated against real execution times obtained in a parallel setting. Then, after proper calibration, the simulator allows to predict times for systems with larger numbers of nodes.

Implementation of parallel similarity measure computations between vectors in a hybrid CPU/Xeon Phi environment was optimized by the author previously and described in detail in paper [5]. Specifically, best sizes of vector batches sent for computations were obtained experimentally including dynamic second vector batch size, optimizations including setting `MIC_USE_2MB_BUFFERS` and overlapping communications were implemented and verified as giving benefits in a parallel hybrid environment.

Compared to the aforementioned works, this paper contributes by optimization of computing similarities for pairs of multidimensional vectors in a hybrid CPU+GPU environment with specific performance oriented optimizations including: overlapping communication and computations, memory management optimizations, data partitioning and best GPU grid size determination. As such, conclusions found in this paper are directly useful for a considerable number of programmers using such modern hardware for application of vector similarity search in many applications.

3 Problem statement and approach to parallelization

The problem considered in this paper can be stated as follows: design, implement and optimize a parallel code for computation of a similarity measure between every pair of a large number of high dimensional vectors such that the code scales in a hybrid environment with compute devices such as CPUs and GPUs of various relative performances. Similarly to [5], a testbed similarity measure was a square root of the sum of differences of values in corresponding dimensions power 2. All similarities across vector pairs can then be reduced to a single value according to a given operator. For the following tests, except those in Section 4.3.7, the maximum operator is used which requires all pair similarity computations.

The code should also scale in a case in which pairs of vectors that meet certain criteria are considered. The latter makes load balancing more difficult because computations required for some pairs of vectors are different than for the others. This is the case for the considered scenario when the goal is to consider pairs for which similarity exceeds a given threshold.

Parallelization of the problem involves the following steps:

1. Partitioning – how data and computations should be broken into data packets for processing by available compute devices. Compute devices in a hybrid environment differ in performance.

2. Assignment – how data packets are assigned to particular compute devices in the system.
3. Execution – actual launching computations and running code on compute devices.

It should be noted that partitioning and assignment can be performed either statically i.e. before computations start or dynamically, at runtime.

3.1 Partitioning

For partitioning of input data, the following algorithm is applied, which is depicted in Figure 1. The figure presents the result space represented by a matrix in which each vector is paired with each other vector for comparison. Along each side of the square there are the input vectors lined up. Consequently, the shaded triangle (excluding the diagonal) represents a result space. Rectangles of predefined size are generated and aligned to cover the triangle of results. Each rectangle has <first vector batch size> × <second vector batch size> size. The order in which the result space is covered is shown in the figure. Because of various performances of compute devices, faster devices would compute more result rectangles than slower ones in the same amount of time. A sufficient number of rectangles is needed to balance load. The sufficient number of rectangles is the smallest number that allows the rectangles to be distributed among compute devices so that execution times of the rectangles on various devices differ by no more than a predefined threshold. The greater differences in compute performances between devices the larger number of rectangles is required.

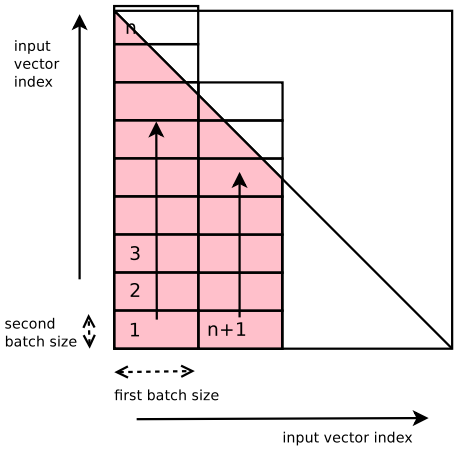

Fig. 1: Partitioning of the result space

3.2 Assignment and execution

Execution of computations corresponding to each rectangle can be regarded as spawning a computational kernel, in terms of technological terms in APIs such as CUDA. Similarly, running a local function in a separate thread on a multicore CPU can also be regarded as launching such a kernel. This is similar to offloading computations to a coprocessor such as Xeon Phi using OpenMP's constructs which can also be regarded as such.

Finally, then, assignment and execution requires proper management of the rectangles and it is accomplished similarly to the approach in [5] i.e.: within a node a number of top level threads is launched each of which is responsible for running a kernel on a compute device. Note that each kernel is parallel within itself:

- a CUDA kernel on a GPU,
- a number of threads running on cores of a multicore CPU.

4 Code optimizations and results

4.1 Testbed environments

For the following tests, two hybrid parallel environments described in Table 1 were used that differ mainly in CUDA capabilities of cards and their target: desktop and server as well as different relative CPU/GPU performances. Interestingly, it is the newer desktop GTX 1070 card in system 1 that offers higher CUDA compute capability than the Tesla K20m in system 2. Both systems feature 2 multicore Intel Xeon family CPUs apart from 2 NVIDIA GPUs in each.

4.2 Testbed application

The testbed problem was described in previous sections of the paper. Its implementation needs to consider, among others, storage of arrays, allocation of space for arrays on a device and further code optimizations, described in subsequent sections. Specifically, the testbed application uses/assumes that vectors are allocated as a single array with vectors stored one after another.

At a high level, pseudocode of the initial implementation executed on the host can be presented as follows:

```
enable nested parallelism in OpenMP;
(...)
// generate input data
generatedata(&data,vectorcount,vectorsize);
(...)
for(i=0;i<requesteddevicecount;i++) {
  allocate memory on host corresponding to given device i;
  cudaSetDevice(i);
  allocate memory on device i;
  create stream for device i;
```

Table 1: System configurations

System	1	2
CPUs	2 x Intel Xeon CPU E5-2620v4 @ 2.10GHz	2 x Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz
CPUs – total number of physical/logical cores	16/32	12/24
System memory size (RAM) [GB]	128	64
GPUs	2 x NVIDIA GTX 1070	2 x NVIDIA Tesla K20m
GPUs – total number of CUDA cores	2 x 2048	2 x 2496
GPU Compute capability	6.1	3.5
GPU memory size [MB]	2 x 8192	2 x 5120
Operating system	Ubuntu Linux version 4.4.0-57-generic	CentOS Linux version 2.6.32-642.6.2.el6.x86_64
Compiler/version	CUDA compilation tools, release 8.0, V8.0.44, gcc 6.2.0	CUDA compilation tools, release 8.0, V8.0.44, gcc 4.4.7

```
}

#pragma omp parallel <data scoping and reduction clauses> num_threads(
    requesteddevicecount+((cputhreadcount>0)?1:0))
{
  (...)
  int i=omp_get_thread_num();
  if (i<requesteddevicecount) { // threads managing execution on GPU(s
      )
    (...)
    cudaSetDevice(i);
    do {
      finish=0;
      #pragma omp critical
      {
        if ((firstbatchcounter<vectorcount) && (secondbatchcounter<
            vectorcount)) {
          determine next batches for computations;
        } else
          finish=1;
      }
      if (finish==0) {
        copy data to device;
        // start computations on the GPU using particular streams (use
            dynamically allocated shared memory)
        slavegpu<<<blocksingrid,threadsinblock,threadsinblock*sizeof(
            double),stream[i]>>>(...);
        copy results from a GPU to the host;
        wait for completion of processing on the GPU;
        // and merge results
        gpuresultmergecpufunction(...);
        (...)
      }
```

```
    } while (!finish);
  } else { // parallel execution on CPU(s)
41  do {
      finish=0;
      #pragma omp critical
      {
        if ((firstbatchcounter<vectorcount) && (secondbatchcounter<
            vectorcount)) {
46        determine next batches for computations;
        } else
        finish=1;
      }
      if (finish==0) {
51      // start computations on the CPU(s)
        slavecpu(...,cputhreadcount);
        merge results;
      }
    } while (!finish);
56 }
}
release resources;
```

It uses:

1. OpenMP for top level threads, each of which is responsible for management of input data, launching computations and fetching results from a compute device – either a GPU – one management thread per GPU or CPU(s) – one management thread per all cores. Each management thread fetches new batches of input data from the input array in a critical section denoted by `#pragma omp critical`. The initial implementation does the following, in terms of management of computations on GPUs: before input data is sent to a GPU, memory is allocated for input vectors, output results (one double per thread block), input data is sent, kernel is launched and output data is copied back to the host memory.
2. CUDA for launching computations on a GPU. The initial implementation uses one stream per device, optimized versions described below use two streams per device to allow overlapping of data/result copying and processing of another kernel at the same time.
3. OpenMP nested parallelism for launching and parallelization of computations within CPU cores. The thread responsible for management of CPU cores uses `#pragma omp parallel for` for parallelization of an outer loop that iterates over vectors in the first batch. Each iteration then goes over second batch vectors. Since we calculate only results for pairs of vectors denoted by the shaded triangle in Figure 1, various first batch vectors may have various numbers of results and iterations of the outer loop may take various amounts of time. In this case it is important to note that `schedule(dynamic,1)` is used for efficient assignment of iterations to threads.

In order to balance load among compute devices that potentially differ in performance, the number of similarly sized (differences can arise at the end of batches of vectors) batches must be considerably larger than the number of compute devices.

For subsequent tests, the application was compiled with the `-O3` flag for high optimization. Three runs per each test configuration, unless otherwise noted, were run with selection of best results.

4.3 Tests, results and discussion

4.3.1 Determination of CPU-GPU thread management configuration

Firstly, for a selected first and second batch sizes as well as grid size configuration we aim at determination of how many computing threads on CPUs should be launched apart from top level threads in charge of GPU management. It should be noted that 2 threads are used for high level management of launching computations on two GPUs. Figure 2 presents results for various numbers of additional *computing threads* on CPUs on system 1. The first batch size equal to 512, second batch size equal to 256 and 1024 threads per block were used as an example at this point for an initial parallel implementation. Optimization of the latter as well as parameter tuning is performed in subsequent sections, in particular in Section 4.3.5. The reason for the observed

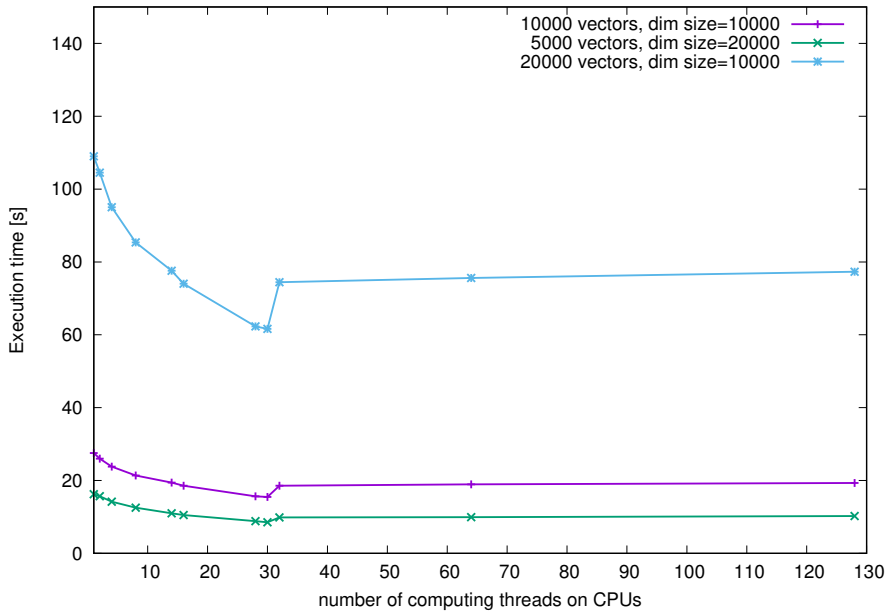

Fig. 2: System 1: Execution time [s] vs the number of computing threads for CPUs; 2xGPUs used

increase in execution time from 30 to 32 threads computing on the CPUs is

that they need to share cores with the two threads managing computations on the GPUs i.e. launching host to device communication, kernel and device to host communication. Consequently, based on the previous results, the number of computing threads for CPUs was selected as <the number of logical processors-2>. This formula also turned out to be the best for system 2. Such values are used for subsequent tests, for the two platforms, both with 2 GPUs.

4.3.2 Optimization of the GPU kernel

The initial kernel followed the CPU implementation i.e. each thread was assigned a pair of vectors for which a similarity measure was to be computed. While this approach works correctly, on a GPU several threads of a single block would be computing pairs for which obviously one of the threads would be different and thus would be accessing very dispersed locations in the global memory. In order to improve this implementation, the author proposed and implemented the idea shown in Figure 3.

The main steps and logic of this solution are as follows:

1. As before, each thread is assigned a pair of vectors. That is, each thread is responsible for a different pair of vectors from the rectangle of <first vector batch size> × <second vector batch size> size. This is determined using the thread's id $blockIdx.x * blockDim.x + threadIdx.x$ considering the rectangle size. However, in this solution, threads in a thread block compute a similarity measure of each of the pairs assigned to this block's threads in parallel. This means that x threads of a thread block ($x \leq 1024$ in CUDA) are assigned x pairs of vectors and parallel computation of each of these pairs is performed one by one.
2. A loop iterates over pairs of vectors. In each iteration j following from 0 to $blockDim.x - 1$ it is determined which pair would be computed i.e. the pair for which the j-th thread in a block is responsible. This information is put into shared memory before `__syncthreads()` so that each thread in a block knows the pair to be compared.
3. An internal loop (executed in each thread) allows each thread to compute $(a_i - b_i)^2$ elements in parallel. Each thread uses stride $blockDim.x$ which optimizes memory access (memory coalescing).
4. A parallel reduction sum is run for all threads in a block using shared memory (like in [11]) with log complexity (the number of steps in terms of the number of threads).
5. A square root of the result of the previous step is stored in a cell of another shared memory array at the index to which the pair of vectors was initially assigned.
6. After the outer loop has finished, the latter shared memory array contains results for the x pairs of vectors. Now, another parallel reduction is run with the given operator.

The new kernel was compared to the initial one for selected numbers of vectors and dimension sizes, for the two systems. Results, shown in Figures

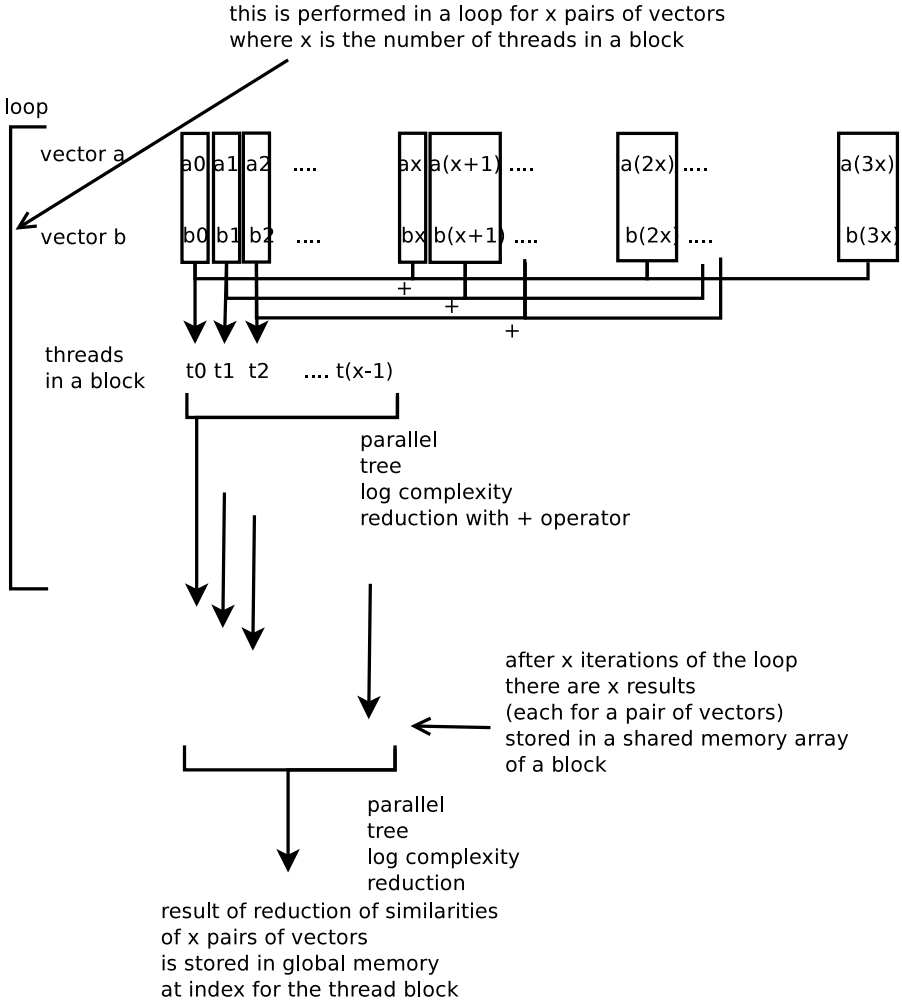

Fig. 3: Improved kernel algorithm

4 and 5 clearly indicate benefits of the improved solution, relatively better for Tesla. The following configurations were used – for system 1: first batch size=1024, second batch size=64, 128 threads per block and for system 2: first batch size=2048, second batch size=128, 256 threads per block. These configurations are best configurations obtained for the two systems according to the analysis of batch and grid sizes shown in Section 4.3.5. Such configurations, unless otherwise noted are used for following tests as well.

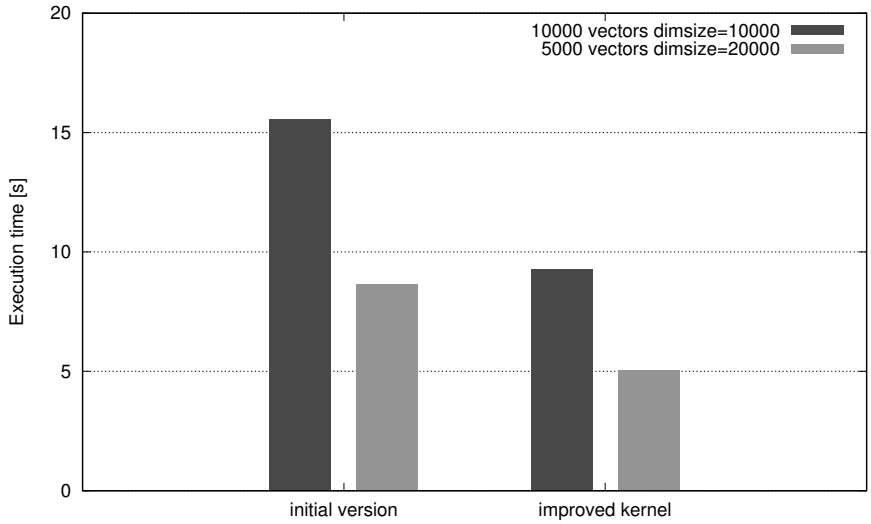

Fig. 4: System 1: gain from the improved kernel implementation

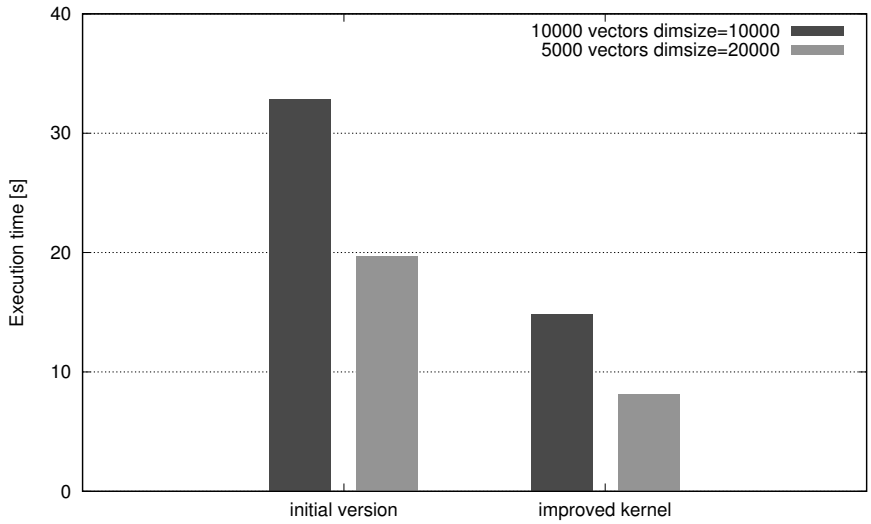

Fig. 5: System 2: gain from the improved kernel implementation

4.3.3 Optimization of GPU memory allocation and CPU-GPU communication

In this step, compared to the allocation scheme of the initial implementation described in Section 4.2, the following optimizations were introduced and tested in the code:

1. There is no need for recurrent memory allocation on the GPU side. It can be done once and reused later if only large enough buffers are allocated. This is known from predefined first and second vector batches. It can also be noted that memory allocation optimization has been found important in a GPU implementation for speech recognition [2].
2. According to the partitioning scheme outlined in Figure 1, a host thread iterates first through first batch vectors and then through second batch vectors. Consequently, it is often the case in subsequent kernel invocations that the first vector batch does not change and there is no need for sending it again. This is detected through keeping track of which first vector batch was sent last time. If the current to be analyzed first vector batch is the same, it is not sent as it is already in the GPU memory and can be reused.

Gains from these code changes are clearly visible and shown in Figure 6, for the two configurations, compared to the previous improved kernel version.

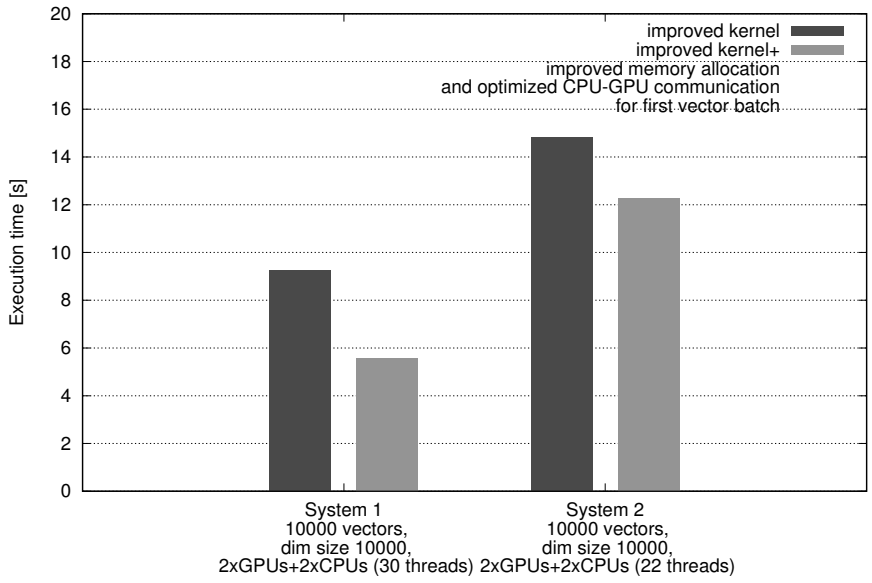

Fig. 6: Gain from optimized GPU memory allocation and CPU-GPU communication, 10000 vectors, dim size = 10000

4.3.4 Overlapping communication and computations as well as concurrent kernel execution on GPUs

In the initial implementation, a host thread manages host to device copying, kernel execution on the device and device to host copying. It is possible to overlap copying and kernel execution in two streams or even kernel execution is supported by the card. Because of that in the next step, the previous implementation was modified in such a way that the host thread creates two streams. Subsequently it puts host to device communication, kernel launch, device to host communication into the first stream, the same sequence of commands for a new second vector (and possibly first vector) batches into the second stream, all launched asynchronously. This results in overlapping of communication and kernel execution in the two streams as well as concurrent kernel execution. Analysis of GPU execution in the NVIDIA Visual Profiler proved the intended overlap. Overlapping also required allocation of host side buffers – vector data (through `cudaHostRegister(...)` for previously allocated data) as well as output result buffers through `cudaHostAlloc(...)` as pinned memory. Allocation as pinned memory may take more time than regular allocation but later results in performance gains through the proposed implementation. These improvements require more memory allocated for the two streams – which is important especially on the GPU side. Figure 7 presents visible improvements from the optimizations for two selected cases for the two analyzed systems. For system 1, 30 CPU threads and 2 threads for GPU management were used. For system 2, 22 CPU threads and 2 threads for GPU management were used.

4.3.5 Optimization of vector batch sizes and GPU grid size configuration

In the next step, using a version with all the aforementioned improvements, we ran performance tests in order to evaluate how batch sizes as well as the number of threads per block impact performance.

Figures 8 and 9 present execution times for a given first vector batch size, second vector batch size for the number of threads per block giving the best execution time (tests were run for 64, 128, 256, 512 and 1024 threads per block for each configuration), for the two systems. There is a trade-off between potential imbalance among compute devices (for large batch sizes) and performance cliff for very small batch sizes which result in a very small number of blocks and under-utilization of a GPU.

It turns out that the best configurations for the two platforms were similar – the first batch either 1024 or 2048, second batch size 64 or 128 (multiples of 2 were tested so that numbers of threads divided by 32 – the number of threads in a warp). It can be noticed that these are very similar to best batch sizes obtained for the same problem benchmarked by the author on a different hybrid system with 2 multicore CPUs and 2 Xeon Phi cards [5].

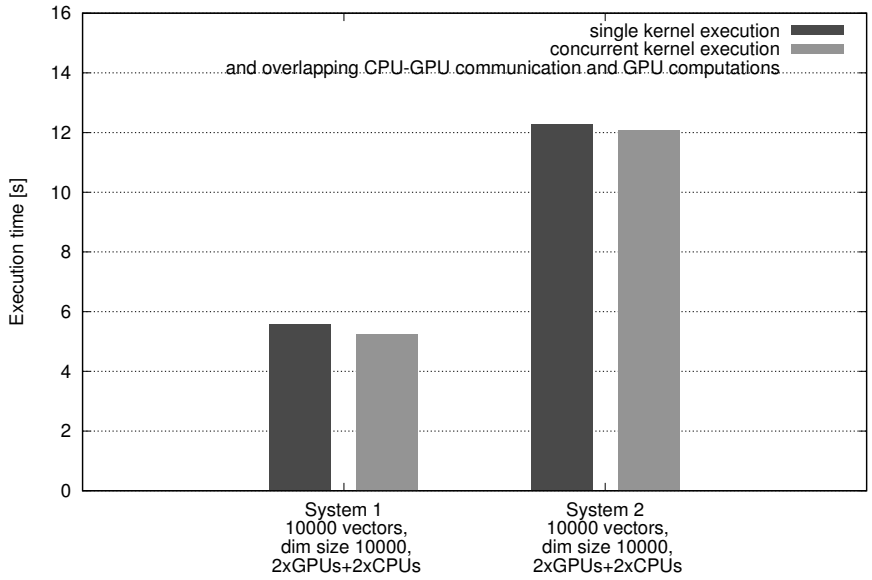

Fig. 7: Gain from overlapping CPU-GPU communication and GPU computations and concurrent kernel execution

4.3.6 Results for various configurations

Based on the previous assessment, a configuration with first batch size=1024, second batch size=64, number of threads per block=128 was used for system 1 and first batch size=2048, second batch size=128, number of threads per block=256 for system 2. Tests were performed for various hardware configurations i.e.: 2xCPU, 1xGPU, 2xGPU and 2xCPU+2xGPU. Results are presented in Figures 10 and 11 for systems 1 and 2 respectively. In comparison, the 2xCPU+2xGPU execution times shown in the figures are up to 8.4% larger compared to theoretical times computed analytically from measured performance of 2xCPU and 2xGPU cases assuming perfect load balancing.

The following conclusions can be drawn from these results:

1. The code scales well i.e. can effectively use more resources for computations of vector similarities on the two platforms.
2. There is a visible difference in GPU to CPU performance for the two systems. On system 1, GPUs are considerably more powerful than CPUs for this code which in turn results in relatively lower gain from 2xGPU to 2xCPU+2xGPU configuration.
3. Gain is visible for various input data sizes, in particular various ratios of the number of vectors to dimension size. This proves that the code scales well for various input data configurations.

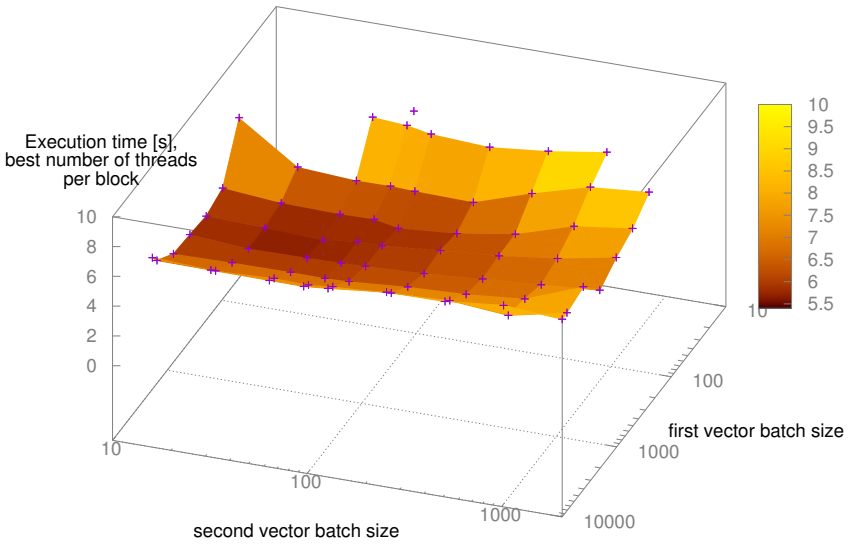

Fig. 8: System 1: Execution time [s], best number of threads per block for each tuple of first batch size and second batch size, 10000 vectors, dim size = 10000, 2 GPUs, 30 computing CPU threads, all optimizations deployed

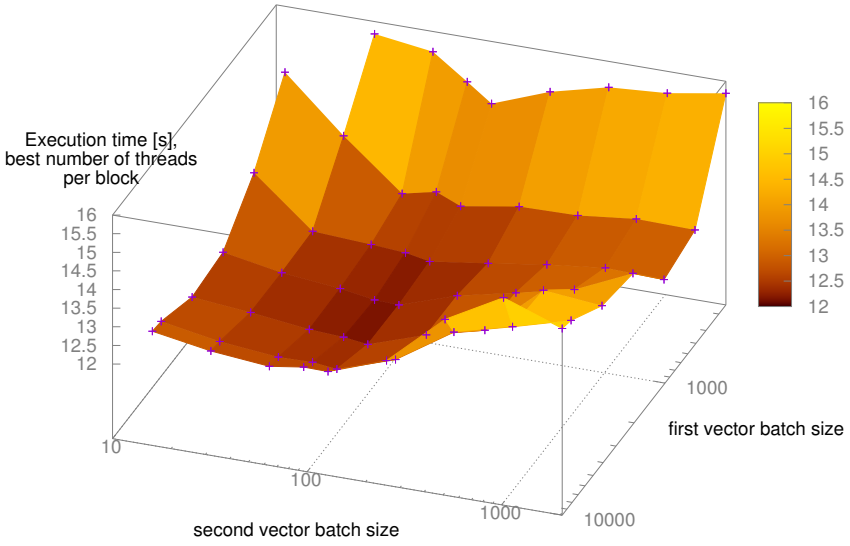

Fig. 9: System 2: Execution time [s], best number of threads per block for each tuple of first batch size and second batch size, 100000 vectors, dim size = 10000, 2 GPUs, 22 computing CPU threads, optimized memory allocation and CPU-GPU communication

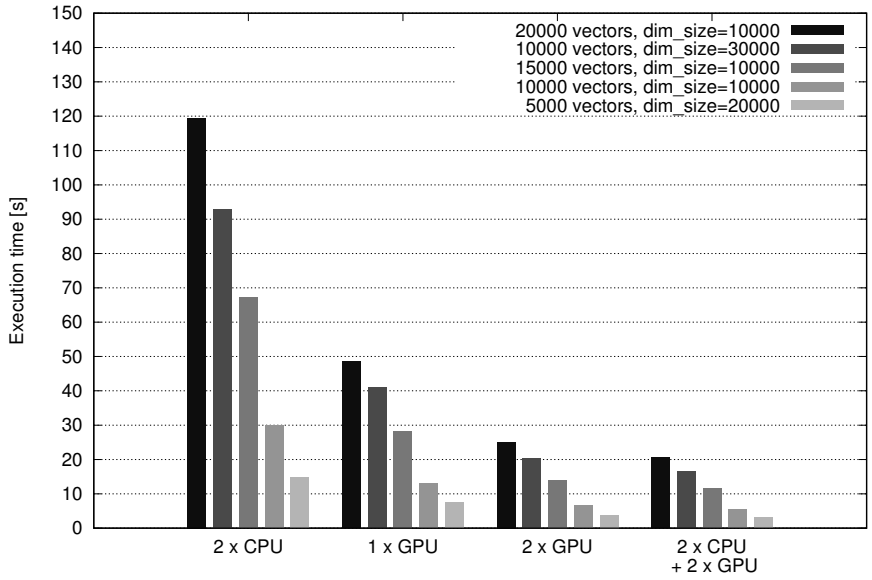

Fig. 10: System 1: results for various configurations for the optimized code, first batch size=1024, second batch size=64, number of threads per block=128

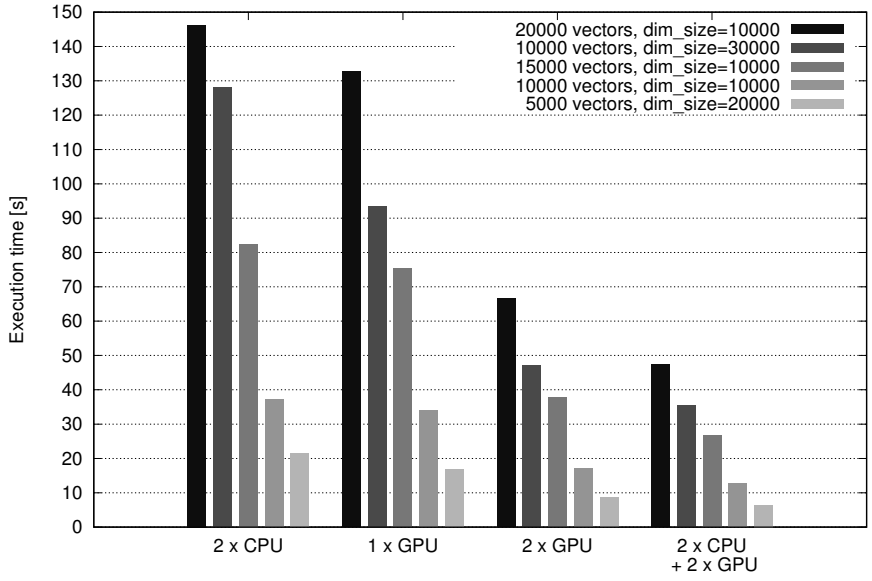

Fig. 11: System 2: results for various configurations for the optimized code, first batch size=2048, second batch size=128, number of threads per block=256

It can be noticed that the code scales for hybrid systems with increasing computational power, with differences in times more visible for systems for which additional resources (such as CPUs in the 2xCPU+2xGPU environment) that are of higher computational power compared to the already included devices – GPUs, such as for system 2. Additionally, Figure 12 presents execution times for various numbers of vectors and dim size=10000 and analogous growth for various hardware configurations.

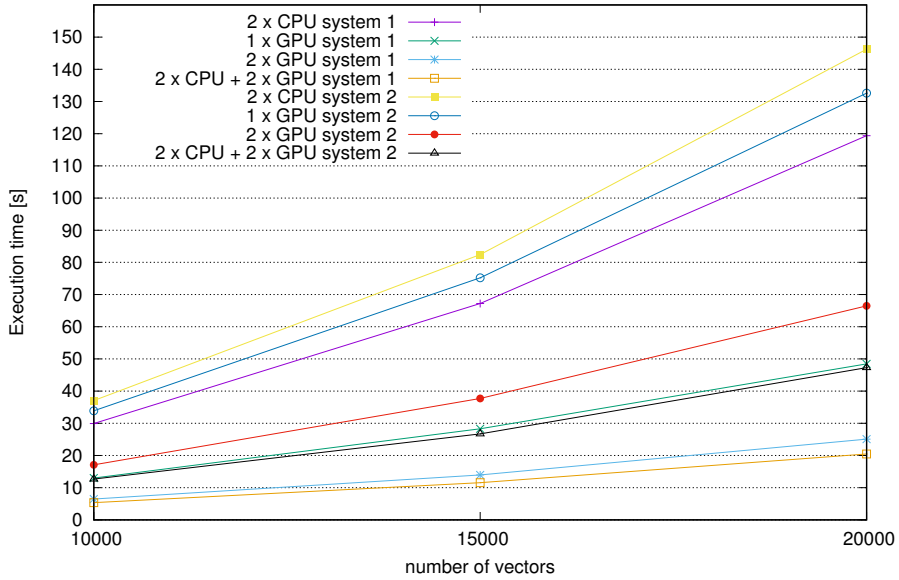

Fig. 12: Execution times vs number of vectors, dim size=10000, optimized code, system 1: first batch size=1024, second batch size=64, number of threads per block=128, system 2: first batch size=2048, second batch size=128, number of threads per block=256

4.3.7 Scaling for all pairs above a threshold

Finally, the implementation was checked in terms of scalability for a case when we search for vectors for which the similarity value exceeds a certain threshold. This results in smaller execution times compared to full search but actually makes load balancing harder because computations for a given pair of vectors can be stopped at the moment when a partial sum exceeds a threshold. In this case, the minimum operator is used.

Firstly, it was implemented in the code in the following way. The loop iterating through vector indices was split into two loops – an outer one and

an inner one – the latter adding a number of elements (the same in the GPU and CPU implementations) after which a check was performed whether the sum exceeded a threshold. This did not involve too much overhead which was important because in the GPU case, checking against a threshold had to be preceded by a parallel reduction over threads in a block.

Figure 13 finally shows that the code scales for such a case, in which processing of various pairs of vectors may require checking various numbers of elements before reaching the threshold.

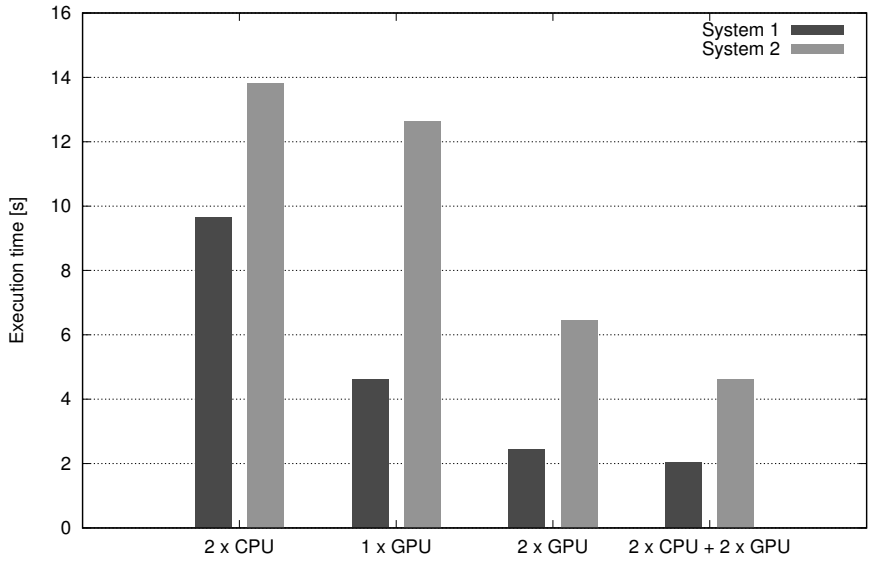

Fig. 13: System 1 and 2, times for a version that searches for all vector pairs above a given threshold, 10000 vectors, dim size = 20000

5 Summary and future work

The paper presented and discussed a parallel OpenMP+CUDA implementation for parallel computation of similarity between pairs of a large number of multidimensional vectors. It allows efficient parallelization and scaling for hybrid CPU+GPU systems that was proved through experiments on two real systems, each with 2 x Xeon CPU + 2 x NVIDIA GPUs, each with a different relative performance of CPU/GPU. Several code improvements were proposed, tested and proved beneficial through experiments, including: kernel tuning by engaging threads for optimized memory access and utilization of

shared memory, minimization of data sent between CPU and GPU, minimization of memory allocation on the GPU side, overlapping CPU-GPU communication and GPU kernel execution, concurrent kernel execution, determination of best vector batch sizes for which computations are requested and best grid configuration on GPUs.

Future work includes taking up other important algorithms for such hybrid systems as well as tuning for the latest Xeon Phi Knight Landing systems, as continuation of research presented in [5] and in this paper. Furthermore, the author plans to incorporate power consumption and energy usage models [6, 7] into optimization of these computations.

References

1. Alabduljalil, M.A., Tang, X., Yang, T.: Optimizing parallel algorithms for all pairs similarity search. In: S. Leonardi, A. Panconesi, P. Ferragina, A. Gionis (eds.) WSDM, pp. 203–212. ACM (2013). URL http://dblp.uni-trier.de/db/conf/wsdm/wsdm2013.html#AlabduljalilTY13
2. Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Hannun, A.Y., Jun, B., Han, T., LeGresley, P., Li, X., Lin, L., Narang, S., Ng, A.Y., Ozair, S., Prenger, R., Qian, S., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, C., Wang, Y., Wang, Z., Xiao, B., Xie, Y., Yogatama, D., Zhan, J., Zhu, Z.: Deep speech 2 : End-to-end speech recognition in english and mandarin. In: M. Balcan, K.Q. Weinberger (eds.) Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, *JMLR Workshop and Conference Proceedings*, vol. 48, pp. 173–182. JMLR.org (2016). URL http://jmlr.org/proceedings/papers/v48/amodei16.html
3. Awekar, A., Samatova, N.F.: Fast matching for all pairs similarity search. Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on **1**, 295–300 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/WI-IAT.2009.52
4. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, WWW '07, pp. 131–140. ACM, New York, NY, USA (2007). DOI 10.1145/1242572.1242591. URL http://doi.acm.org/10.1145/1242572.1242591
5. Czarnul, P.: Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors. International Journal of Parallel Programming pp. 1–17 (2016). DOI 10.1007/s10766-016-0455-0. URL http://dx.doi.org/10.1007/s10766-016-0455-0
6. Czarnul, P., Kuchta, J., Rościszewski, P., Proficz, J.: Modeling energy consumption of parallel applications. In: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 855–864 (2016)
7. Czarnul, P., Rościszewski, P.: Optimization of Execution Time under Power Consumption Constraints in a Heterogeneous Parallel System with GPUs and CPUs, pp. 66–80. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). DOI 10.1007/978-3-642-45249-9_5. URL http://dx.doi.org/10.1007/978-3-642-45249-9_5
8. Czarnul, P., Rościszewski, P., Matuszek, M., Szymański, J.: Simulation of parallel similarity measure computations for large data sets. In: 2015 IEEE 2nd International Conference on Cybernetics (CYBCONF), pp. 472–477 (2015). DOI 10.1109/CYBConf.2015.7175980
9. De Francisci, G., Lucchese, C., Baraglia, R.: Scaling out all pairs similarity search with mapreduce. Large-Scale Distributed Systems for Information Retrieval p. 27 (2010)
10. Dunn, T., Banerjee, N.K., Banerjee, S., undefined, undefined, undefined, undefined: Gpu acceleration of document similarity measures for automated

bug triaging. 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) **00**(undefined), 140–145 (2016). DOI doi.ieeecomputersociety.org/10.1109/ISSREW.2016.27
11. Harris, M.: High performance computing with cuda. optimizing cuda. In: SC07 (2007). Http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf
12. Hartung, M., Kolb, L., Groß, A., Rahm, E.: Optimizing Similarity Computations for Ontology Matching - Experiences from GOMMA, pp. 81–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-39437-9_7. URL http://dx.doi.org/10.1007/978-3-642-39437-9_7
13. Jo, Y., Bae, D., Kim, S.: Efficient computations of link-based similarity measures on the GPU. In: 3rd IEEE International Conference on Network Infrastructure and Digital Content, IC-NIDC 2012, Beijing, China, September 21-23, 2012, pp. 261–265. IEEE (2012). DOI 10.1109/ICNIDC.2012.6418756. URL http://dx.doi.org/10.1109/ICNIDC.2012.6418756
14. Kruliš, M., Skopal, T., Lokoč, J., Beecks, C.: Combining cpu and gpu architectures for fast similarity search. Distributed and Parallel Databases **30**(3), 179–207 (2012). DOI 10.1007/s10619-012-7092-4. URL http://dx.doi.org/10.1007/s10619-012-7092-4
15. Lam, H.T., Dung, D.V., Perego, R., Silvestri, F.: An incremental prefix filtering approach for the all pairs similarity search problem. In: W.S. Han, D. Srivastava, G. Yu, H. Yu, Z.H. Huang (eds.) APWeb, pp. 188–194. IEEE Computer Society (2010). URL http://dblp.uni-trier.de/db/conf/apweb/apweb2010.html#LamDPS10
16. Ma, C., Wang, L., Xie, X.: GPU accelerated chemical similarity calculation for compound library comparison. Journal of Chemical Information and Modeling **51**(7), 1521–1527 (2011). DOI 10.1021/ci1004948. URL http://dx.doi.org/10.1021/ci1004948
17. Mabotuwana, T., Lee, M.C., Cohen-Solal, E.V.: An ontology-based similarity measure for biomedical data – application to radiology reports. Journal of Biomedical Informatics **46**(5), 857 – 868 (2013). DOI http://dx.doi.org/10.1016/j.jbi.2013.06.013. URL http://www.sciencedirect.com/science/article/pii/S1532046413000889
18. McInnes, B.T., Pedersen, T.: Evaluating measures of semantic similarity and relatedness to disambiguate terms in biomedical text. Journal of Biomedical Informatics **46**(6), 1116 – 1124 (2013). DOI http://dx.doi.org/10.1016/j.jbi.2013.08.008. URL http://www.sciencedirect.com/science/article/pii/S1532046413001238. Special Section: Social Media Environments
19. Obin, N., Roebel, A.: Similarity search of acted voices for automatic voice casting. IEEE/ACM Transactions on Audio, Speech, and Language Processing **24**(9), 1642–1651 (2016). DOI 10.1109/TASLP.2016.2580302
20. Pantel, P., Crestan, E., Borkovsky, A., Popescu, A.M., Vyas, V.: Web-scale distributional similarity and entity set expansion. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2, EMNLP '09, pp. 938–947. Association for Computational Linguistics, Stroudsburg, PA, USA (2009). URL http://dl.acm.org/citation.cfm?id=1699571.1699635
21. Phong, P.H., Son, L.H.: Linguistic vector similarity measures and applications to linguistic information classification. International Journal of Intelligent Systems **32**(1), 67–81 (2017). DOI 10.1002/int.21830. URL http://dx.doi.org/10.1002/int.21830
22. Pushpa, C., Girish, S., Nitin, S., Thriveni, J., Venugopal, K., Patnaik, L.: Computing semantic similarity measure between words using web search engine. In: D.C. Wyld, D. Nagamalai, N. Meghanathan (eds.) Third International Conference on Computer Science, Engineering & Applications (ICCSEA 2013), pp. 135–142. Delhi, India (2013). ISBN : 978-1-921987-13-7, DOI: 10.5121/csit.2013.3514
23. Rodriguez-Serrano, J.A., Perronnin, F., Llados, J., Sanchez, G.: A similarity measure between vector sequences with application to handwritten word image retrieval. In: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pp. 1722–1729 (2009). DOI 10.1109/CVPR.2009.5206783
24. Szymanski, J.: Mining relations between wikipedia categories. In: Networked Digital Technologies - Second International Conference, NDT 2010, Prague, Czech Republic, July 7-9, 2010. Proceedings, Part II, pp. 248–255 (2010)
25. Szymanski, J.: Comparative analysis of text representation methods using classification. Cybernetics and Systems **45**(2), 180–199 (2014)

26. Yadav, K., Mittal, A., Ansari, M.: Parallel implementation of similarity measures on gpu architecture using cuda. Indian Journal of Computer Science and Engineering (IJCSE) **3**(1) (2012). ISSN: 0976-5166
27. Zadeh, R.B., Goel, A.: Dimension independent similarity computation. Journal of Machine Learning Research **14**(1), 1605–1626 (2013). URL http://dl.acm.org/citation.cfm?id=2567715