2017 46th International Conference on Parallel Processing

Nearly Balanced Work Partitioning for Heterogeneous Algorithms

Hardhik Mallipeddi[a,1], Dip Sankar Banerjee [b,2]
Kiran Raj Ramamoorthy[a,3], Kannan Srinathan[a,4] and, Kishore Kothapalli[a,5]

[a]International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500 032.
[b]Indian Institute of Information Technology, Guwahati
Ambari, Gopinath Bordoloi (G.N.B.) Road, Guwahati, India 781001.

[1]mallipeddi.hardhik@research.iiit.ac.in, [2]dipsankarb@iiitg.ac.in,
[3]kiran.raj@research.iiit.ac.in, [4]srinathan@iiit.ac.in, [5]kkishore@iiit.ac.in

***Abstract*— The architectural trend towards heterogeneity has pushed heterogeneous computing to the fore of parallel computing research. Heterogeneous algorithms, often carefully handcrafted, have been designed for several important problems from parallel computing such as sorting, graph algorithms, matrix computations, and the like. A majority of these algorithms follow a work partitioning approach where the input is divided into appropriate sized parts so that individual devices can process the "right" parts of the input. However, arriving at a good work partitioning is usually non-trivial and may require extensive empirical search. Such an extensive empirical search can potentially offset any gains accrued out of heterogeneous algorithms. Other recently proposed approaches too are in general inadequate.**

In this paper, we propose a simple and effective technique for work partitioning in the context of heterogeneous algorithms. Our technique is based on sampling and therefore can adapt to *both* the algorithm used and the input instance. Our technique is generic in its applicability as we will demonstrate in this paper. We validate our technique on three problems: finding the connected components of a graph (`CC`), multiplying two unstructured sparse matrices (`spmm`), and multiplying two scale-free sparse matrices. For these problems, we show that using our method, we can find the required threshold that is under 10% away from the best possible thresholds.

I. Introduction

A noticeable architectural trend affecting parallel computing research currently is the shift towards a heterogeneous collection of devices involving CPUs and accelerators such as GPUs and the Intel Xeon-Phi. Hence, the design and development of heterogeneous algorithms aimed at such commodity heterogeneous computing platforms are gaining immense research interest. In recent years, several carefully handcrafted heterogeneous algorithms on various heterogeneous platforms are designed for a variety of fundamental and challenging problems from parallel computing such as sorting [3], sparse matrix operations [22], [17], dense linear algebra routines [21], graph algorithms [9], [5], and the like. Despite their success, a few important questions remain to be answered.

Several of the above algorithms use a work partitioning based approach where the input is divided into possibly unequal pieces according to the values (thresholds) of certain parameters. The individual devices in the computing platform process appropriate pieces of the input depending on various factors including architectural and execution characteristics of the device.

In general, however, it is not clear as to how to arrive at a good work partitioning. For certain workloads, in particular regular workloads, it may be possible to arrive at the required work partitions by using one of the several simple strategies. For instance, one can think of dividing the work based on the processing capability of the devices using metrics such as FLOPS, or can use the average obtained over a set of benchmark instances to guide the work division process. Such simple techniques tend to work on regular workloads such as dense matrix multiplication as our study in Figure 1 shows. The experiment in Figure 1 runs a heterogeneous algorithm for dense matrix multiplication based out of the Intel MKL [18] on a multi-core CPU and the corresponding CUDA library routine on an NVidia K40c GPU. The matrix multiplication work is divided across the CPU and the GPU in proportion of the FLOPS of the CPU and the GPU. Figure 1 shows the best possible threshold that is to be used obtained via an exhaustive search and the threshold that is obtained using the relative ratio of FLOPS. It can be seen that the thereshold obtained via the ratio of FLOPS is close to the best possible threshold. This helps us to partition the workload in a balanced manner.

The problem is accentuated when the workload is irregular in nature, one encounters two difficulties. Firstly, given the impact of itregularities in the workload, it is difficult to obtain a good mechanism to partition the work. Secondly, no direct mechanisms may exist to translate a given partition threshold to actual partitions of the input that can correspond to the work partition. In particular, it is often difficult to see which portion of the input will correspond to a 20% work volume. Computations on sparse graphs and sparse matrices fall under this category. These twin challenges make it extremely difficult to efficiently arrive at exact or near-exact work partitioning for irregular workloads in a heterogeneous setting.

In such cases, often, one has to resort to an exhaustive search over the parameter space to arrive at the best possible work partitioning. Such an exhaustive search has several disadvantages. Exhaustive search, given the time it takes, is

2332-5690/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPP.2017.14

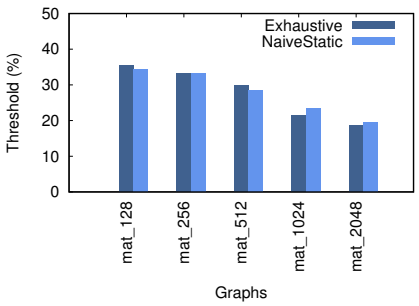


Fig. 1. This shows the threshold estimated by the sampling method in comparison to the best possible threshold obtained by an exhaustive search along with NaiveStatic estimate. We also show the time taken by Algorithm 1 using the threshold estimated by the sampling method in comparison to the time taken when using the best possible threshold along with the time taken with NaiveStatic partitioning. Elements of the matrices are chosen uniformly at random INTEGERS/REALS RANGE. The label "mat_n" on the X-axis indicates multiplication of square matrices each of size $n \times n$.

not an efficient technique and can potentially nullify the gains obtained by heterogeneous execution. Further, in most cases, a good work partition can be achieved only by taking into consideration the heterogeneous algorithm used and, *importantly*, the input instance. Thus, exhaustive search on one input may not throw light on a different input.

Other techniques such as recording historical profiles [20], [10], runtime scheduling approaches [2], static approaches [12], [19], semi-automatic approaches [16], [30] that require user-input for work partitioning, mitigate the problems of exhaustive search but come with other drawbacks. In particular, if the computation in question is non-uniform in nature and is sensitive to the nature of the input, then the above mentioned techniques fail to work.

The impracticality and inadequacy of the existing approaches therefore calls for general purpose and fully automatic techniques that can address the issue of work partitioning in the context of heterogeneous algorithms. We believe that deterministic approaches cannot address the problem at hand. Therefore, one has to look for randomized approaches.

In this paper, we propose a simple randomized technique that has several advantages such as being *adaptable* to both the algorithm and the input instance, fast, and near-accurate. Our technique is based on the idea that to know the nearly correct work partition, we can *sample* the input and work with the sampled input to *identify* a good work partition on the sample input. Given this knowledge on the sampled input, we can then *extrapolate* the knowledge gained on the sampled input to achieve a (nearly) good work partitioning for the original input. Our technique is particularly useful when the work partitioning is input dependent. The fundamental motivation behind using this approach is that the generated samples have the properties of the input imbibed in them. This leads to a much smaller working set and considerable reduction in the work complexities. Further, our technique has a low overhead and hence can largely retain the benefits of heterogeneous algorithms.

Our main technical contributions can be summarized as follows. We introduce a framework (See Section II) for obtaining nearly balanced work partitioning in the context of heterogeneous algorithms. We then apply our technique to problems such as graph connected components (CC) and multiplying sparse matrices (`spmm`). The results of these case studies (See Sections III–V) are summarized in Table I.

Workload	Threshold Difference (%)	Time Difference (%)	Overhead %
CC	7.5%	4%	9%
`spmm`	10.6 %	19.1 %	13%
Scale-free `spmm`	5.25 %	6.01 %	1%

TABLE I

TABLE SUMMARIZING THE RESULTS OF APPLYING THE SAMPLING TECHNIQUE TO THREE DIFFERENT WORKLOADS.

In Table I, the column "Threshold Difference" indicates the percentage difference in the estimation achieved by the sampling approach compared to the best possible estimate. The column "Time Difference" shows the percentage difference in the time taken by the respective algorithm using the threshold estimated by the sampling approach in comparison to the time taken using the best possible estimate. The column "Overhead" indicates the percentage of overall time spent in the arriving at the estimation using the sampling method. The results from Table I indicate that our method spends less than 9% of the time in obtaining an estimate that is on average within 8% of the best possible estimate and sacrifices on average under 9% of the best possible runtime.

A. Related Work

We organize the related work into three sections that cover work related to work partitioning, CC, and `spmm`.

1) Work Partitioning: Work partitioning in the context of heterogeneous algorithms is a topic of immense research interest for its potential benefits. Static work partitioning using machine learning techniques were studied by Grewe et al. [12]. Their work does not introduce any input dependent features and the work partitioning is largely left to the runtime system that relies on static code features such as the number of integer and floating point operations, branches, and the like. The work of Grewe et al. [12] is extended by Kofler et al. [19] by incorporating a few dynamic code features. Both of the solutions work at the compiler level whereas our approach is compiler agnostic and therefore can be easily adapted to all heterogeneous platforms.

Predicting the behaviour of a workload and partitioning it accordingly is studied by Shen et al. [25]. Their study is however not input dependent and relies only on the general workload characteristic. Obtaining such a characteristic may not be straight-forward. Luk et al. [20] present an approach where the first run of a program is treated as a *training run* that is used to determine the task mapping for later runs. In contrast to our work, their model is not input dependent and hence can suffer on cases where the actual input plays a crucial role in the process. The work of Luk et al. builds upon the

work of Gregg et al. [10] that considers historical runtime data to arrive at work partitioning.

In a more recent work, Boyer et al. [6] use the current execution profile of a program to perform load balancing on the remaining part of the input. This method can however introduce communication overhead. Further, their method assumes that each "chunk" of the work requires (near) equal processing time. This uniformity assumption can be done away with in our approach as sampling can preserve the non-uniformity at least on expectation. Our method does not require any runtime communication.

Semi-automatic work partitioning where user inputs on the relative performance characteristics of the devices are used to arrive at the partitioning is proposed by Hou et al. [16]. On the other hand, our approach does not require any user inputs and is fully automatic. Work partitioning for power efficiency by using various system power usage parameters is proposed in [30]. Their approach requires one to use execution time characteristics on a given device for a given problem. Obtaining such metrics is not easy, and hence limits the applicability of works based on them.

A work partitioning framework using shared work queues in the context of heterogeneous systems is presented by Augonnet et al. [2]. Our approach differs from this approach as we do not need any runtime data sharing that may be difficult in an heterogeneous execution environment. Moreover, the work volume may not be directly related to the contents of the work queue and hence using work queues may not solve the problem of work partitioning effectively.

2) CC: Algorithms in the PRAM model for finding the connected components of a graph have been given by given by Hirschberg et al. [14], and by Shiloach and Vishkin [26]. The algorithm by Shiloach and Vishkin [26] is more suited for sparse graphs and has been the choice algorithm for most parallel implementations on multi-processor systems [11], GPUs [27], and heterogeneous systems [5], [4].

3) `spmm`*:* The multiplication of two matrices, dense and sparse alike, is an important primitive with applications to many areas of computing. The `spmm` workload is one of the important computations in parallel computing and is featured in the Berkeley dwarfs [1]. A considerable work is therefore targeted at improving the performance of `spmm` on modern parallel architectures including multi-core CPUs via the Intel MKL library [18], GPUs, and heterogeneous platforms.

Focusing on sparse matrices, one of the first notable works is that of Gustavson [13] that presents a Row-Row fashion `spmm` algorithm. Buluc et al. [7] conducted a detailed study of `spmm` on distributed memory systems. Sulatycke et al. [29] present cache optimized algorithms on sequential machines for sparse matrix multiplication. They explore Row-Row and Column-Row formulations of matrix multiplications. More recently, in [22], heterogeneous algorithms for `spmm` are designed on CPU+GPU systems.

II. Sampling Based Work Partitioning

In this section, we describe our approach for work partitioning in the context of heterogeneous algorithms. For simplicity of the exposition, we consider a simple heterogeneous system with one CPU attached to one GPU. However, our technique can be extended easily to other heterogeneous computing platforms.

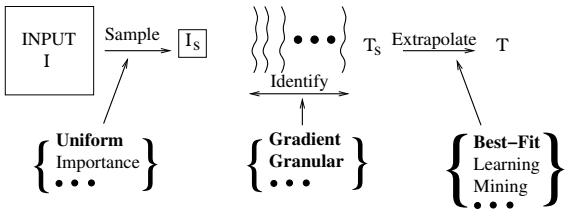


Fig. 2. An illustration of our sampling method. Each step has a choice of techniques some of which are listed in the bottom half of the figure. The techniques that we use in this paper are shown in bold-face font.

Let I be an input instance for a heterogeneous algorithm A for a problem P. We consider the situation that A partitions the input I based on some parameter (threshold) into two pieces I_1, I_2 so that each piece of the input is processed on a separate device in the computing platform. Our approach has the following steps.

1) **Sample:** Create an input I_s of size smaller than I using randomization and sampling.
2) **Identify:** On the input I_s, apply algorithm A and identify the right value(s) of the threshold(s) for input I_s.
3) **Extrapolate:** Extrapolate the value(s) of threshold(s) on I_s to those for I.

Figure 2 illustrates the above steps. As shown in Figure 2, there are multiple ways to accomplish each step. The choices we use in this paper are shown in bold-face font. The exact nature of sampling may also impact answers to the above questions. In this work, we limit ourselves to uniform sampling. We leave the scope for other sampling methods, e.g., importance sampling [23], that have found favor in randomized sampling for future work.

Further, the extrapolation required in Step 3 may depend on the nature of the heterogeneous algorithm and the nature of the computation. We observe that in some cases Step 3 may require one to deploy tools from other domains such as data mining or learning theory. In particular, the larger question in Step 3 is to "find" the relation between work partitioning that can be achieved on the sampled input I_s to that of the original input I. This relation may be studied offline on a sample dataset and the knowledge of the relation can then be used for "any" input instance.

Some of the benefits of our approach are as follows. Firstly, Steps 1 and 3 of our approach, i.e., Sample and Extrapolate, can be seen to be input dependent, and Step 2 can be seen to be algorithm dependent. Therefore, our method caters to *both* the algorithm and the input instance. Secondly, as we use randomization and sampling, our method can work well for non-uniform and irregular workloads also. In a way, randomization and sampling help us obtain a faithful miniaturization I_s (of I) that can provide insurance against some parts of the input biasing a deterministic work partitioning methodology in use.

Finally, since the size of the sampled input is expected to be small, our method allows us the freedom to conduct multiple runs of the algorithm on the sampled input to understand the behavior of the original input.

Thus, for our technique to work, one has to devise mechanisms for sampling, identifying the work partitioning for the sampled input, and extrapolating the knowledge to the original input. In particular, some questions that need to be addressed in the context of our technique are the following.

- What should be the size of I_s and how to sample I to I_s so that the characteristics of I are preserved in I_s?
- What is the time taken in identifying the value(s) of the threshold(s) on I_s?
- How to extrapolate the value(s) of the threshold(s) on I_s to that of I?

Some of these questions can naturally lead to trade-offs. For instance, it is intuitive to expect that the larger the size of I_s, the more accurate the extrapolation in Step 3 can be performed. However, working with a larger I_s may increase the time spent in Step 2. In a similar fashion, a small I_s offers scope for a very small time spent in Step 2, but the accuracy of the estimation may suffer. Since the size of the sample may also depend on the problem, we conduct this study in later sections. In our case studies, we apply the framework and also study the above questions in detail. Our case studies concern irregular workloads such as finding the connected components of a graph and `spmm`. We also show subsequently (cf. Figure 7) that randomness is an essential ingredient in our proposed technique.

Our technique can be extended to other heterogeneous platforms naturally. In a way, the values of the threshold(s) now can be treated as a vector, unlike a scalar in the simple CPU+GPU case. We do not need any additional assumptions such as the availability of advanced profiling, compilation, or run-time systems.

III. Case Study I – Graph Connected Components

Finding the connected components of a given graph is an important computation with applications to several other graph algorithms. In this work, we use an algorithm similar to the algorithm of Banerjee et al. [5] to study our sampling technique from Section II.

The algorithm starts off by dividing the input graph $G = (V, E)$ into two graphs G_{GPU} and G_{CPU} that are processed on the GPU and the CPU respectively. An implementation of the Shiloach and Vishkin algorithm [26] and the sequential depth-first search algorithm [8] are used on the GPU and the CPU respectively. Edges of G such that one end point is in G_{GPU} and another in G_{CPU} are called as *cross edges*. Once the connected components of G_{GPU} and G_{CPU} are identified, the algorithm processes the cross edges so that the connected components of G_{GPU} and G_{CPU} can potentially be merged.

We refer the reader to Algorithm 1 for an algorithmic description. In Algorithm 1, the labels CPU :: and GPU :: refer to the computations done in an overlapped manner on the CPU and the GPU respectively.

As can be observed, the algorithm uses work partitioning in Phase I (See Algorithm 1, Line 1–6) where the parameter t dictates the graphs G_{GPU} and G_{CPU}. However, it is not clear as to what is a good value for t. In fact, the CC workload is such that beyond exhaustive experimentation, there is no reported way to find the best value of t. In particular, even if we know the time taken for finding the connected components of *a* given graph instance on the CPU and the GPU exclusively, it is not clear as to the value of t since the correlation between the runtime and the work volume is not directly established. Using our sampling based framework from Section II, we will find a good value for t with a small overhead as follows.

Algorithm 1 Connected Components(G)

Input: A graph $G = (V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$
Output: The number of components

{**/* PHASE I: Partition */**}

1: Find a threshold t between 0 to 100 using sampling.
2: $n_{cpu} := \frac{n \times t}{100}$, $n_{gpu} = n - n_{cpu}$
3: Partition G into two subgraphs G_{GPU} and G_{CPU} with $V(G_{\mathrm{CPU}}) = \{v_1, v_2, \cdots, v_{n_{cpu}}\}$ and $V(G_{\mathrm{GPU}}) = V \setminus V(G_{\mathrm{CPU}})$;
4: Set $E(G_{\mathrm{CPU}}) = E(G) \cap (V(G_{\mathrm{CPU}}) \times V(G_{\mathrm{CPU}}))$.
5: Set $E(G_{\mathrm{GPU}}) = E(G) \cap (V(G_{\mathrm{GPU}}) \times V(G_{\mathrm{GPU}}))$.
6: Divide G_{CPU} into equal parts $G_{\mathrm{CPU}1}, G_{\mathrm{CPU}2}, ...G_{\mathrm{CPU}c}$ when using c threads on the CPU.

{**/* PHASE II: Heterogeneous Computation */**}

7: GPU :: Find the connected components of G_{GPU}
8: CPU :: Find the connected components of G_{CPU} in parallel
9: GPU :: Merge the components found on the CPU and the GPU using cross edges

A. Sampling Framework

1) Sample: We choose a set S of $\sqrt{n}$ vertices of G uniformly at random. We then set G' as the graph induced by S in G, i.e., $G' = G[S]$.

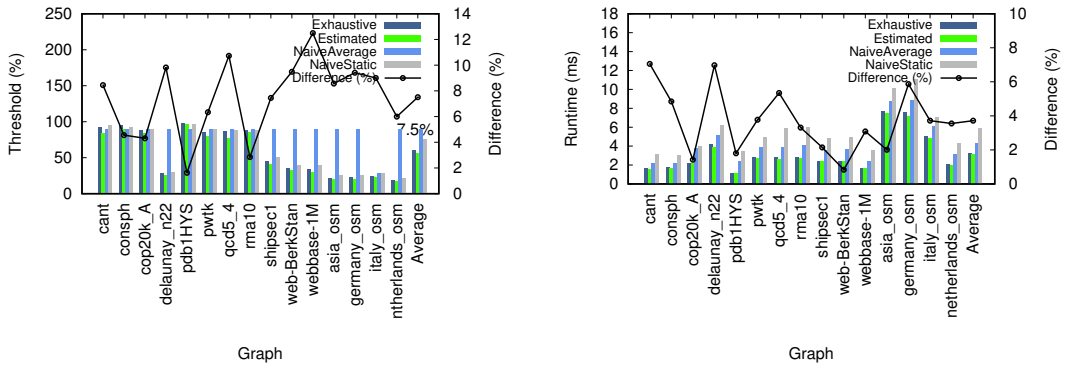
2) Identify: We run the above heterogeneous algorithm, Algorithm 1 with various values of the threshold on the graph G'. To minimize the number of runs, we use a coarse-grained estimation followed by a fine-grained process. To this end, we run with values of t' that differ by 8, and once the best value of t' is identified, we then run on values of t' that differ by 1.

3) Extrapolate: In this case, if G' preserves the properties of G, then we expect that t should be identical to t'. Indeed that is what we observe in our experimentation also and hence use the value of t' to partition the graph G as G_{GPU} and G_{CPU}.

B. Results

In this section, we study the results of our sampling based approach. We use graphs listed in Table II for our experiments. These datasets are drawn from standard datasets from the University of Florida collection [28].

1) Experimental Platform: Our heterogeneous computing platform consists of an Nvidia K40c GPU attached to an Intel(R) Xeon E5-2650 CPU via the PCI Express link. The CPU has 128 GB RAM and is a dual processor with each



(a) Threshold

(b) Runtime

Fig. 3. Figure (a) shows the threshold estimated by the sampling method in comparison to the best possible threshold obtained by an exhaustive search. In Figure (b), we show the time taken by Algorithm 1 using the threshold estimated by the sampling method in comparison to the time taken when using the best possible threshold. In the latter, the time taken for the exhaustive search is not included.

Graph/Matrix	n	m or, NNZ
cant	62,451	4,007,383
consph	83,334	6,010,480
cop20k_A	121,192	2,624,331
delaunay_n22	4,194,304	25,165,738
pdb1HYS	36,417	4,344,765
pwtk	217,918	11,634,424
qcd5_4	49,152	1,916,928
rma10	46,835	2,374,001
shipsec1	140,874	7,813,404
Web Graphs		
web-BerkStan	685,230	7,600,595
webbase-1M	1,000,005	3,105,536
Road Networks		
asia_osm	11,950,757	25,423,206
germany_osm	11,548,845	24,738,362
italy_osm	6,686,493	14,027,956
netherlands_osm	2,216,688	4,882,476

TABLE II

LIST OF GRAPHS USED IN OUR EXPERIMENTS. IN THE TABLE, n REFERS TO THE NUMBER OF NODES AND m REFERS TO THE NUMBER OF EDGES IN THE GRAPH. WHEN VIEWED AS A MATRIX, n REFERS TO THE NUMBER OF ROWS, AND NNZ REFERS TO THE NUMBER OF NON-ZEROS IN THE MATRIX.

processor having 10 cores operating at 2.34 GHz. Using SMT, the CPU can run 40 threads simultaneously. The Tesla K40c GPU is a current generation Kepler micro-architecture from NVidia that has 15 streaming multi-processors (SMX) with each having 192 cores for a total of 2880 compute cores. Each compute core is clocked at 745 MHz. Each SMX has a hardware scheduler which schedules 32 threads at a time. This group is called a warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU has a fully cached memory access via a L2 cache of size 1.5 MB. To program the GPU we use the CUDA API Version 6.5.

2) Results: We study three aspects of our approach. One is the closeness of the estimated threshold to that of the best possible threshold obtained via an exhaustive search. Secondly, we study the time overhead of our approach. Finally, we study the trade-off between the sample size and the accuracy of the approach.

In Figure 3(a), we show the threshold identified by our method and the best possible threshold obtained by exhaustive search. The label "Exhaustive" ("Estimated") in Figure 3(a) refers to the threshold obtained via an exhaustive search (*resp.* the sampling method). The secondary Y-axis in Figure 3(a) shows the absolute difference between the estimated and the one obtained by exhaustive search as a percentage. As can be noted, the above difference is 7.5% on average. We additionally compare our estimated results with two naive partitioning mechanisms which provides shows the usefulness of our mechanism. Towards this we compare our results against two naive partitioning mechanisms that are labled in Figure 3(a) and Figure 3(b) as "NaiveStatic" and "NaiveAverage". NaiveStatic refers to the partitioning of the input graph between the CPU and the GPU based on the FLOPS ratio. Clearly, the GPU having a higher FLOPS rating gets the bigger of the two partitions which is 88% on average. The second "NaiveAverage" partitioning is done based on prior experimentations. Through several rounds of prior exhaustive runs we arrive at an ideal partitioning threshold. The thresholds arrived at for all the datasets under consideration are then averaged and treated as the threshold percentage for all of the input graphs. For the graphs that we have considered, this averaged value of threshold comes to 90%.

We anticipate that the time taken by our algorithm using the estimated threshold t', would be close to the time taken by our algorithm using the best possible threshold obtained via exhaustive search.

In Figure 3(b) we compare the time taken by Algorithm 1 when using our approach with respect to the time taken by the algorithm when using the threshold, labeled as "Exhaustive" in Figure 3(b). The label "Estimated" refers to the time taken by Algorithm 1 when using the threshold t' estimated by

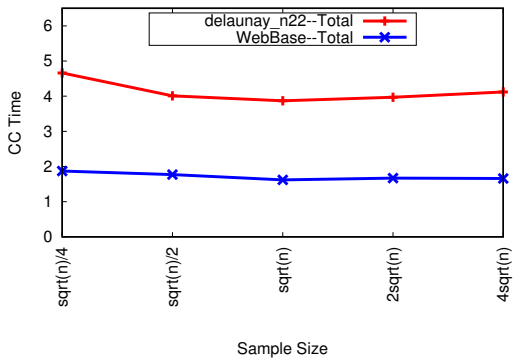


Fig. 4. Figure shows the trade-off between the size of the sample graph and the overall time taken and the time taken in estimating the threshold by Algorithm 1.

the sampling approach. The label "Naive" represents the time taken by the naive homogeneous solutions where there is no partitioning involved and algorithm is run on the GPU only. Note that the primary Y-axis in Figure 3(b) is on a log-scale. The secondary Y-axis of Figure 3(b) shows the percentage increase in the runtime of Phase II of Algorithm 1 when using the threshold t' to the time taken when using the best possible threshold.

As can be seen, using our technique incurs a slowdown of no more than 4% on average. The time taken to estimate the threshold is around 9% of the overall time on average. This indicates that the threshold we estimate can result in a near balanced work partition. Our result should also be seen in light of the fact that it is not practical to obtain the best possible threshold whereas the sampling based approach offers a practical solution. Further, our method performs much better on large graphs as compared to small graphs indicating that minor deviations from the best possible threshold can be tolerated as the size of the graph increases.

Sensitivity: We proceed to study the sensitivity of our results as we vary the sample size. To this end, we vary the size of the sampled graph between $\sqrt{n}/4$ to $4\sqrt{n}$ and note the total time taken (Phase I + Phase II of Algorithm 1). The results of this study for two graphs is shown in Figure 4. As can be seen, for both these graphs, the total time has a (near) concave behavior. This is intuitive as a larger sample can provide for better work partitioning at the expense of taking more time to perform the estimation. At the sample size of $\sqrt{n}$, the total time is minimum, so our choice of $\sqrt{n}$ as the sample size is justified.

IV. Sparse Matrix Multiplication on Heterogeneous Systems

On a heterogeneous CPU+GPU platform, the current best performing algorithm is to use the row-row based matrix multiplication method on both the CPU and the GPU as shown in [22]. The rows of matrix A, in the computation of $A \times B$, are partitioned across the CPU and the GPU such that the computation for an optimal $r\%$ of the overall workload is done on the CPU and the rest is done is on the GPU. This parameter r will be referred to as the split percentage in the rest of this section.

The algorithm from [22] is summarized as Algorithm 2 below. For `spmm`, it is actually possible to estimate the work volume as follows. Consider the matrix product $A \times B$ and obtain the vector V_B with $V_B[i]$ set to the number of nonzeros in the ith row of B. When B is of size $n \times p$, V_B has size $n \times 1$. The product $A \times V_B$ will be a vector L_{AB} such that $L_{AB}[i]$ equals the work volume of the ith row of A in the product $A \times B$. We make use of this observation in Algorithm 2.

As can be observed, Algorithm 2 uses a work partitioning approach. Given a work partition percentage r, the algorithm computes L_{AB} and divides the matrix A horizontally into two matrices A_1, A_2 such that $A_1 \times B$ has an $r\%$ of the work volume of $A \times B$ and $A_2 \times B$ has the $(100-r)\%$ of the work volume of $A \times B$.

Algorithm 2 SPMM(A)

Input: A matrix A with n rows and m columns, a matrix B with m rows and k columns, split percentage r

Output: Resultant Matrix C with n rows and k columns, $C = A \times B$

{**Phase I: Partitioning** }

1: Compute the load vector for L_{AB}, $L = |||L_{AB}|||_1$ on the GPU.
2: Calculate the load on CPU and GPU as $L_{CPU} = (r \times L/100$ and $L_{GPU} = L - L_{CPU}$ on the GPU.
3: Find out the split row index i where $V_L[i]$ is closest L_{CPU} on the GPU.
4: Split the matrix A such that $A_1 = A[0 \cdots i]$ and $A_2 = A[i+1 \cdots n]$

{**Phase II: Heterogeneous Computation**}

5: Compute $A_1 \times B$ using [22, Algorithm 1] on the CPU.
6: Compute $A_2 \times B$ [22, Algorithm 1] on the GPU.
7: Transfer results of GPU to CPU and add that to CPU's results.

The split percentage r in Algorithm 2 is hard to find analytically. Given the architectural peculiarities in a heterogeneous setting, there is often very little relation between the system metrics posted by the manufacturer and the throughput achieved (or time spent) on a *given* instance of sparse matrix multiplication. Some of the reasons for this difficulty stem from the fact that the computation is quite irregular in nature more so for sparse matrices that exhibit an unstructured nature of sparsity. In fact, small changes to the optimal work split percentage can result in significant changes to the overall time taken to complete the computation. In the following, we show that sampling can be used to *however* arrive at a near balanced partitioning of the computation across the CPU and the GPU.

A. Sampling Method

We use the following sampling method for identifying the split percentage in the `spmm` computation on a CPU+GPU heterogeneous platform.

a) Sample: We choose a submatrix A' of size $n/k \times n/k$ from matrix A uniformly at random. This ensures that the number of nonzeros in the matrix A', denoted NNZ', is also scaled appropriately as $NNZ'_i = NNZ_i/K$ where NNZ'_i denotes the number of non-zeroes in the i^{th} row of A' and NNZ_i is the number of non-zeroes in the i^{th} row of A, and K is a constant. We use 4 as the value of K in our experiments. This helps us to preserve the sparsity structure of A in the sampled matrix A'.

b) Identify: We run the heterogeneous algorithm, Algorithm 2, with various values of split percentage r' on A' and obtain an optimal split percentage for A'. To minimize the number of runs, we use a coarse estimation followed by a fine-grained process. The coarse estimation is done by multiplying the sample matrices A' and B' on CPU and GPU independently in parallel and stop when either of them finishes. At this step, by observing the amount of work processed, we can roughly estimate the split percentage for $A' \times B'$. This is followed by a finer search to narrow down on the exact work partition percentage for $A' \times B'$. (Note that this method cannot be used on the computation $A \times B$ due to its large time requirement).

c) Extrapolate: In this case, if A' preserves the sparsity structure of A, then we expect that r should be identical to r'. Indeed that is what we observe in our experimentation also and hence use the value of r' as the split percentage to divide the work between the CPU and the GPU in the computation of $A \times B$.

B. Experimental Results

In this section, we study the results of our sampling based approach for `spmm`. For experimentation we use the same platform mentioned in Section III-B.1. We use the matrices listed in Table II for our experiments. We multiply the matrix A by itself for reasons of compatibility of matrix multiplication.

1) Results: We study three aspects of our approach as described in Section III-B.2. In Figure 5(a), we show the split percentage identified by our method r_{est} and the best possible threshold r_{best} obtained by an exhaustive search. The label "Exhaustive" ("Estimated") in Figure 5(a) refers to the split obtained via an exhaustive search (*resp.* the sampling method). The secondary Y-axis in Figure 5(a) shows the absolute difference between the estimated and the one obtained by exhaustive search as a percentage. As can be noted, the above difference is 10.6% on average. As already explained in the discussion of the previous results on graph connected components, in this case too we consider two naive partioning mechainsims namely "NaiveStatic" and "NaiveAverage". The former being the statically determined threshold based on FLOPS ration while the later is the average of exhaustive thresholds arrived at through multiple prior runs over all the datasets under consideration.

We anticipate that the time taken by our algorithm using the estimated split percentage, r', would also be close to the time taken by our algorithm using the best possible threshold obtained via an exhaustive search. In Figure 5(b) we compare the time taken by Algorithm 2 when using our approach with respect to the time taken by the algorithm when using the optimal split ratio r, labeled as "Exhaustive". The label "Estimated" ("Exhaustive") refers to the time taken by Algorithm 2 when using the split ratio r', (*resp.* the best possible split percentage). The primary Y-Axis shows the time taken for the computation in milliseconds. The secondary Y-axis of Figure 5(b) shows the percentage increase in the runtime of Algorithm 2 when using the split percentage r' to the time taken when using the best possible split percentage.

As can be seen, using our technique with a sample matrix of size $N/4 \times N/4$ incurs a slowdown of no more than 19% on average. The time taken to estimate the threshold (overhead) is 13% of the overall time on average. This indicates that the threshold we estimate can result in a near balanced work partition. In fact, our approach suffers more on web and road networks compared to the other matrices. This behavior can be explained likely by the artefact effects of sampling on web and road networks. Our result should also be seen in light of the fact that it is not practical to obtain the best possible threshold whereas the sampling based approach offers a practical solution.

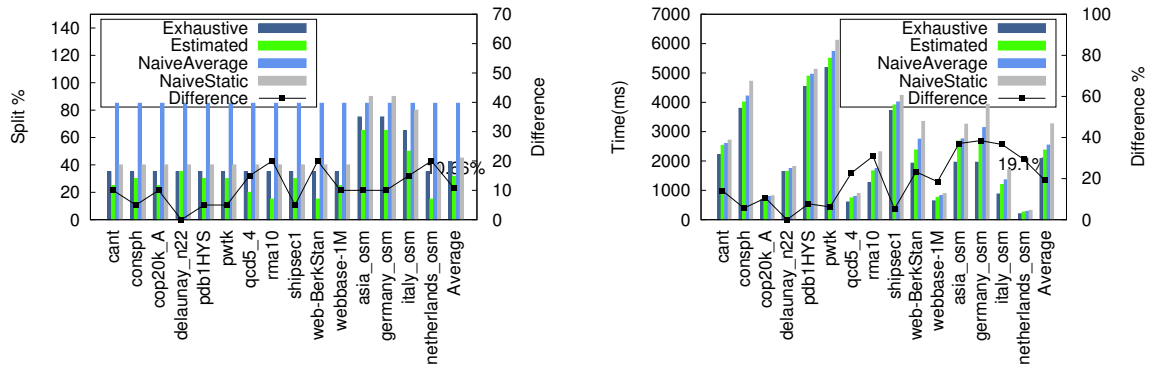
Sensitivity: We proceed to study the sensitivity of our results as we vary the sample size. To this end, we vary the size of the sampled graph between $n/10 \times n/10$ and $4n/10 \times 4n/10$ and note the total time taken (Phase I + Phase II of Algorithm 2). The results of this study for two graphs is shown in Figure 6. As can be seen, for both these matrices, the time taken using the sampling based approach has a near concave behavior. This is intuitive as using a larger sample size can increase the time taken to perform the estimation whereas a smaller sample size can lead to inaccuracies in estimating the right threshold. At the sample size of nearly $N/4$, we observe that we can estimate the split percentage within a reasonable overhead. So our choice of $N/4$ as the sample size is justified.

Role of Randomness: We finally note that randomization in Step 1 of the sampling based technique is essential to its success. To this end, we choose the sample matrix A' as a predetermined $n/4 \times n/4$ submatrix of the input matrix A. We perform the next two steps of our techniqe on the matrix A'. We also repeat these steps on four different predetermined submatrices of A. The results of this experiment are shown in Figure 7 for two matrices cant and cop_20k. As can be seen from Figure 7, predetermined samples tend to be inaccurate in estimating work partition threshold.

V. Case Study 3 – `spmm` on Scale-Free Sparse Matrices

In the context of sparse matrices, it can be observed that several sparse matrices arising in practical scenarios exhibit a scale-free nature. A matrix exhibiting a scale-free nature has several rows with very few nonzero elements and very few rows with a large number of nonzero elements. In other words, the row densities follow a power-law distribution.

Such a concentration of the nonzeros in a small fraction of the rows can have significant implications for algorithm design and implementation as shown in [24]. The algorithm from [24] is shown below as Algorithm 3. The algorithm can be



(a) Split Percentage

(b) Runtime

Fig. 5. Figure (a) shows the split percentage estimated by the sampling method in comparison to the best possible threshold obtained by an exhaustive search. In Figure (b), we show the time taken by Algorithm 2 using the threshold estimated by the sampling method in comparison to the time taken when using the best possible threshold. In the latter, the time taken for the exhaustive search is not included.

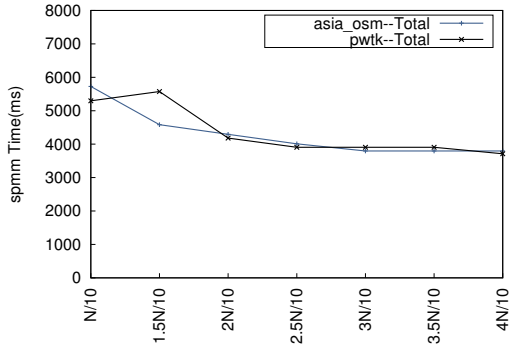


Fig. 6. Figure shows the trade-off between the size of the sample matrix and the overall time and the time spent in estimating the split percentage by Algorithm 2.

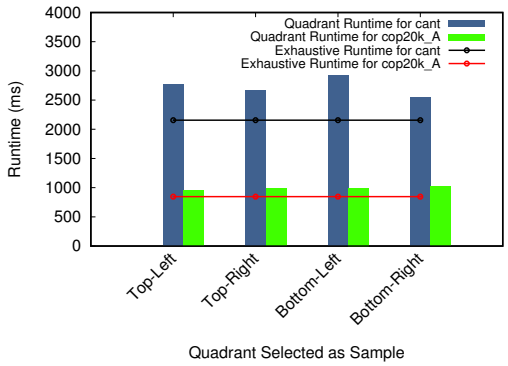


Fig. 7. Figure showing the importance of randomization in our technique.

summarized as follows. We show that having such knowledge about the nature of sparsity can also help in efficient sampling. In particular, the size of the sample required in this case will be smaller than the case of the earlier section. This results in a less overhead in the sampling and estimation phase. This section also shows that the sampling technique can be used when the work partitions are based on indirect parameters rather than the work volume directly.

Given two scale-free matrices A and B, we wish to compute their product $C = A \times B$. (Throughout, we assume that A and B are compatible for multiplication). Algorithm HH-CPU [24] has four phases as shown in Algorithm 3. In the first phase, the algorithm uses a threshold t to prepare matrices A_H, A_L and B_H, B_L that contain the high dense and low dense rows of A and B respectively. A row of a matrix is said to be a high (*resp.* low) dense row if the row has more (*less*) than t non-zeros. In Phase II, the algorithm computes the product of A_H with B_H on the CPU and the product of A_L with B_L on the GPU. In Phase III, the products A_H with B_L and that of A_L with B_H are computed on both the CPU and the GPU. The results of the computations done in Phases II and III are combined in Phase IV.

As can be observed, Algorithm 3 requires a threshold t such that in Phase II, rows with more than t nonzeros are processed on the CPU, while rows with less than t nonzeros are processed on the GPU. We apply our sampling technique from Section II to obtain the value of the threshold t.

Algorithm 3 Algorithm HH-CPU

1: **/* Phase I */** Identify thresholds t_A, t_B and the matrices A_H, A_L, B_H, and B_L.
2: **/* Phase II */** Compute $A_H \times B_H$ on the CPU, and $A_L \times B_L$ on the GPU using [22, Algorithm 1]
3: **/* Phase III */** Compute $A_H \times B_L$ and $A_L \times B_H$ on the CPU and the GPU.
4: **/* Phase IV */** Combine the results of Phases II and III using both the CPU and the GPU

A. Sampling Framework

We apply the sampling technique to obtain a good estimate for the threshold t as follows. For uniformity in scalability we consider the same matrix for both A and B. So in effect, B and A are the same matrices.

1) Sample: We first create a submatrix A' of A with $\sqrt{n}$ rows. The elements of this submatrix are chosen such that A' has a sparsity pattern that is similar to that of the sparsity pattern of A on expectation. This is achieved by sampling $\sqrt{n}$ rows of A uniformly at random to be included in A'. In each such row, we sample elements from the corresponding row of A and ensure that the column indices are transformed so that the column indices are within 1 to $\sqrt{n}$.

2) Identify: In the second step, we use a gradient descent based approach to find the best threshold that works for A', say t'. This requires us to run our algorithm on A' multiple times with various thresholds. However, since the size of A' is smaller than that of A, the time incurred in these additional runs is often a small fraction of the time taken on A.

3) Extrapolate: Finally, we extrapolate t' obtained for A' to get a good threshold t_A for A. For this, we note that the relation between t_s and t_A can depend on the nature of computation and the nature of the algorithm that is being used. For the case of `spmm`, we use an off-line best-fit strategy that finds the most plausible relation between t_s and t_A. We find that $t_A = t_s \times t_s$ and therefore use t_A as the threshold in Algorithm 3.

B. Experimental Results

In this section, we study the results of our sampling based approach for `spmm`on scale-free matrices. We use the scale-free matrices (Matrices in rows till 1 through 11 excluding 4 and 7), listed in Table II for our experiments. The matrices excluded are not scale-free in nature and hence Algorithm HH-CPU is not suitable as discussed in [24]. For reasons of compatibility, we multiply each matrix in Table II with itself. (In other words, we compute $A \times A$). The experimental platform we use is as described in Section III-B.1.

We start by analyzing the sampling based approach and its ability to identify the right value of the threshold t in Phase I of Algorithm 3. In Figure 8[a], we plot the threshold obtained by the sampling method, labeled as "Estimated", and the best possible threshold, labeled as "Exhaustive". The secondary Y-axis shows the absolute difference between these two as a percentage. It can be observed that indeed the average difference is around 5.25%. The labels "NaiveStatic" and "NaiveAverage" denotes the naive partitioning values arrived using techniques already discussed in the previous workloads.

In Figure 8(b) we compare the time taken by Algorithm HH-CPU using the estimated threshold, t_A, to that of the time taken by the algorithm when using a best possible threshold obtained via an exhaustive search. As earlier, the label "Exhaustive" refers to the runtime of Algorithm HH-CPU using the threshold identified via an exhaustive search. The secondary Y-axis in Figure 8(b) shows the percentage difference between the runtime of Algorithm HH-CPU when using the thresholds t_A and the one obtained by an exhaustive search. As can be seen, the time taken when using the threshold t_A is only 6% away from the time taken using the best possible threshold. The better performance compared to `spmm` from Section IV is due to the fact that the size of the sample is much smaller now thereby reducing the time spent in Phase I of Algorithm 3. The sampling based approach therefore offers a practical way to find the near best threshold whereas the best possible threshold obtained by an exhaustive search is not a practical alternative.

Sensitivity: To understand the trade-off between the sample size required and the accuracy of estimating the threshold, we conduct the following experiment. We vary the number of rows in the sampled matrix A' as $\sqrt{n}/4, \sqrt{n/2}, \sqrt{n}, 2 \cdot \sqrt{n}$, and $4 \cdot \sqrt{n}$. We then record the time taken by Algorithm 3 when using the threshold obtained from Phase I. Intuitively, the accuracy of estimating the threshold improves significantly as we increase the size of A'. However, the time taken to estimate the threshold also increases as the time taken in each run of Algorithm HH-CPU on input A' increases with the size of A'.

The results of this study on two matrices is shown in Figure 9. In Figure 9, the Y-axis shows the total time taken by Algorithm HH-CPU over varying sample size. We observe that the overall time has a minimum when the size of the sampled matrix is $\sqrt{n} \times \sqrt{n}$.

VI. Concluding Remarks

In this paper, we have proposed sampling as an effective and practical method to help in work partition in the context of heterogeneous algorithms. We have applied the technique to three important problems and studied the advantages of the approach. It is interesting to adapt and apply our techniques to other emerging heterogeneous computational platforms and also to other problems that require work partitioning.

References

[1] K. Asanovic et al. A View of the Parallel Computing Landscape Comm. ACM, 2009 V. 52, pp. 56–67.

[2] C. Augonnet, S. Thibault, N. Raymond, P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Conc. and Comp.: Prac. & Exp., v. 23 n. 2, pp.187–198.

[3] D. S. Banerjee, P. Sakurikar, and K. Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. In *Proc. AsHES, 2013*, pp. 1060-1069.

[4] D. S. Banerjee, S. Sharma, and K. Kothapalli. Work efficient parallel algorithms for large graph exploration. In HiPC, 2013 pp. 81-93.

[5] D. S. Banerjee, and K. Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components in Proc. of HiPC, 2011. pp. 1-10.

[6] M. Boyer, K. Skadron, S. Che and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. in Proc ACM Computing Frontiers, 2013. pp. 21:1–21:10.

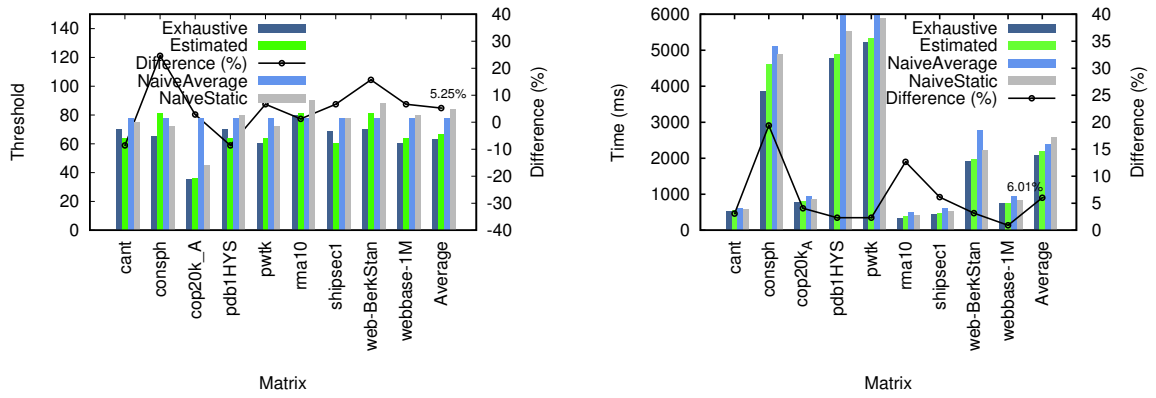
[7] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In Proc. ICPP'08, pp 503–510, 2008.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms MIT Press, 2001.

[9] A. Gharaibeh, B. Costa, E. Santos-Neto, and M. Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In in Proc. of IEEE IPDPS (2013). pp. 851-862.

[10] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. in Workshop on Applications for Multi- and Many-Core Processors, 2011.

[11] J. Greiner. A comparison of parallel algorithms for connected components, in Proc. SPAA, 1994, pp. 16–25.



(a) Threshold

(b) Runtime

Fig. 8. Timings of HH-CPU using the estimated threshold versus the empirical threshold. The plot also shows the time taken for the estimation of the threshold. The line anchored to the secondary Y-axis shows the percentage difference between the time taken by Algorithm 3 using the estimated threshold and the empirical threshold.

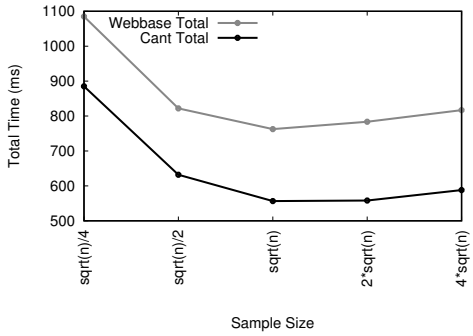


Fig. 9. Figure shows the trade-off between the sample size and the overall time and the time taken for estimating the threshold by Algorithm 3 for two matrices.

[12] D. Grewe and Michael F. P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL in Intl. Conf. on Compiler Construction, 2011. pp. 286-305.

[13] F. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM T. Math. Soft.,4(3):250–269, 1978.

[14] D. S. Hirschberg, A. K. Chandra, D. V. Sarwate. Computing connected components in parallel computers CACM 22, 8 (1979), 461–464

[15] S. Hong, N. C. Rodia, and K. Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *Proc. of SC'13* (2013) pp. 92:1–92:11.

[16] Q. Hou, K. Zhou, and B. Guo. SPAP: A programming language for heterogeneous many-core systems. Technical report, Zhejiang University Graphics and Parallel Systems Lab, 2010.

[17] S. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and Workload-aware algorithms for Spare Matrix- Vector Multiplication, in Proc. ACM Compute, 2014. pp. 3:1–3:9.

[18] Intel Math Kernel Library https://software.intel.com/en-us/intel-mkl

[19] K. Kofler, I. Grasso, B. Cosenza and T. Fahringer An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning in ACM ICS, 2013. pp. 149-160.

[20] C-K. Luk, S. Hong, and H. Kim Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. in Proc. of Intl. Symp. on Microarchitecture (MICRO), 2009. pp. 45-55.

[21] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra, Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators, IEEE T. on Par. and Dist. Comp, 2010.

[22] K. Matam, S. Indarapu, and K. Kothapalli. Sparse Matrix Matrix Multiplication on Hybrid CPU+GPU Platforms, in Proc. of HiPC, 2012.

[23] R. Motwani and P. Raghavan Randomized Algorithms Cambridge Univeristy Press, 2000

[24] K. R. Ramamoorthy, D. S. Banerjee, K. Srinathan and K. Kothapalli A Novel Heterogeneous Algorithm for Multiplying Scale-Free Sparse Matrices IPDPS Workshops 2015, pp. 637–646.

[25] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. in Proc. ACM ICS, pp. 241–250, 2014.

[26] Y. Shiloach and U. Vishkin,. An $O(\log n)$ parallel connectivity algorithm. J. Algorithms, pp. 57–67, 1982.

[27] J. Soman, K. Kothapalli, and P. J. Narayanan. Some GPU Algorithms for Graph Connected Components and Spanning Tree, Parallel Processing Letters, vol. 20, no. 4, pp. 325–339, 2010

[28] Stanford Network Analysis Platform dataset , http://www.cise.ufl.edu/research/sparse/matrices/SNAP/

[29] P. D. Sulatycke, and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices, in Proc. of IPDPS, pp.117-123, 1998.

[30] G. Wang and X. Ren Power-efficient work distribution method for CPU-GPU heterogeneous system. in Proc. ISPA, 2010. pp. 122-129.