Technical Report

The performance of the CPU, GPU, multi-GPU and heterogenous versions of SCD

Stijn Heldens (stijn.heldens@student.uva.nl)
Ana Lucia Varbanescu (a.l.varbanescu@uva.nl)
Informatics Institute, University of Amsterdam, The Netherlands

Arnau Prat-Pérez (aprat@ac.upc.edu)
Josep-Lluis Larriba-Pey (larri@ac.upc.edu)
DAMA-UPC, Universitat Politècnica de Catalunya, Spain

March 9, 2015

Abstract

Community detection has become a very active topic of research in the last decade due to its applications in many fields of research. One algorithm which can deal with the performance and accuracy requirements of real-world applications is the *Scalable Community Detection* (SCD) algorithm. We have designed and implemented the first GPU version of SCD. We have further extended this version to a heterogeneous version able to run on platforms consisting of one CPU and any number of GPUs. To test the performance and scalability of our solutions, we performed benchmarks on five real-world graphs (ranging from 0.9M to 1.8B edges) and six platforms (with GPUs having between 1.5GB and 6GB memory). In this report, we present the results of these benchmarks.

Contents

1 Introduction

In a network (or graph[1]), a *community* is informally defined as a group of densly interconnected vertices which are sparsely connected to the rest of the network [1]. Community detection, which is the partitioning of a network into communities, gives us valuable insight into the structure and properties of complex networks, such as those from biology and sociology [5]. For example, communities can correspond to proteins having similar functions in protein-protein interaction networks, people with the same interests in social networks, or researchers working on the same topic in academic collaboration networks.

Scalable Community Detection (SCD) [9] is one of the most recently proposed algorithms that aims to tackle both the quality and the scalability challenges of community detection. SCD partitions a network into communities by maximizing the *Weighted Community Clustering* (WCC) metric [8], a community quality metric based on counting triangles. It has been shown that WCC captures the properties of communities better than popular alternative metrics such as modularity [6] and conductance [2]. SCD has also been designed to be scalable and highly parallel. Due to these reasons, SCD outperforms the current state of the art community detection algorithms in terms of both quality and performance [9].

However, analyzing massive networks having millions of edges can still take tens of minutes, even on modern multi-core processors [9]. To overcome this problem, we have designed a GPU version of SCD and implemented the first prototype. Using a single GPU leads to higher performance, but it also means that the networks that can be processed are limited by the amount of memory of the GPU. To tackle this limitation, we have extended our GPU version to a heterogeneous version of SCD able to run on platforms consisting of one CPU and any number of GPUs.

To evaluate the performance of our solution, we have performed benchmarks on five real-world graphs (ranging from 0.9M to 1.8B edges) and six platforms (with GPUs having between 1.5GB and 6GB memory). In this report, we present the results of these benchmarks.

This report is structured as follows: Section 2 briefly discusses the SCD algorithm, Section 3 discusses the different implementations, Section 4 presents the results of our benchmarks and Section 5 presents conclusions.

[1]The terms *graph* and *network* have the same meaning and are used interchangeably throughout this paper

Figure 1: The three steps of the SCD algorithm

2 Scalable Community Detection (SCD)

The *Scalable Community Detection* (SCD) algorithm [9] partitions an undirected and unweighted network into communities by maximizing the *Weighted Community Clustering* (WCC) metric [8]. WCC is a quantitative metric that measures the quality of a network partition into communities. Given an undirected, unweighted graph $G = (V, E)$, a vertex x and a community C, let $t(x, C)$ be the number of triangles that vertex x closes with vertices in C and let $vt(x, C)$ be the number of vertices in C which close at least one triangle with x and a third vertex in C. The cohesion of vertex x to community C is defined as follows:

$$WCC_v(x, C) = \begin{cases} \frac{t(x,C)}{t(x,V)} \cdot \frac{vt(x,V)}{|C \setminus \{x\}| + vt(x,V) - vt(x,C)} & \text{if } t(x, V) \neq 0 \\ 0 & \text{if } t(x, V) = 0 \end{cases} \quad (1)$$

Now, let $\mathcal{P} = \{C_1, \ldots, C_n\}$ be a partition of the graph into communities such that $C_1 \cup \ldots \cup C_n = V$ and $C_i \cap C_j = \emptyset$ if $i \neq j$. Let C_x be the community to which vertex x belongs. The WCC of $\mathcal{P}$ is now defined as the average WCC over all the vertices:

$$WCC(\mathcal{P}) = \frac{1}{|V|} \sum_{x \in V} WCC_v(x, C_x) \quad (2)$$

SCD is an algorithm which takes a undirected and unweighted graph $G = (V, E)$ as its input and produces a partition of the graph into communities by maximizing the WCC in a greedy fashion. The algorithm consists of three steps: preprocessing, initial partitioning and partition refinement. These steps are shown in Fig. 1 and will be briefly discussed here.

During the preprocessing step, all edges which do not close any triangles are deleted from the graph. These edges do not affect the WCC, so removing them reduces the size of the graph without any effect on the WCC. Next, an initial partition of the network into communities is created using a simple heuristic. This initial partition serves as input to the refinement phase in which the partition is iteratively improved. In each iteration, a new candidate partition is created by evaluating each vertex of the graph independently and performing the action which leads to the largest increase in WCC. There are three possible types of actions:

(1) **No action**: leave the vertex in its current community.

(2) **Remove**: remove the vertex from its current community and place it alone in a newly created singleton community.

(3) **Transfer:** transfer the vertex from its current community to the community of one of its neighbors.

The best action for every vertex is decided and all vertices are then updated simultaneously. by applying these actions. At the end of one iteration, the WCC of the resulting partition is computed. The algorithm continues with a new iteration if the WCC has improved more than a given threshold.

For more detailed descriptions of WCC and SCD, we refer the reader to [8] and [9] respectively.

3 Heterogeneous SCD

We designed a GPU version of SCD and have implemented a first prototype using CUDA (Compute Unified Device Architecture) [7], a framework for programming GPUs by NVIDIA. We have extended this version to a heterogeneous version able to run on any platform having one CPU and any number of GPUs. We present short descriptions of these versions in this section. The source code for our implementation[2] has been publicly released under the GNU GPL v3.0.

3.1 The GPU Version

The partition refinement phase (see Fig. 1) is the most computationally expensive phase of the algorithm, so we designed our GPU solution to perform this phase on the GPU. After preprocessing and creating the initial partition (performed on the CPU), each vertex is assigned a label corresponding to the identifier of the community. These labels together with the graph are transfered to the GPU. The values of $t(x, V)$ and $vt(x, V)$ for every vertex x are computed during the preprocessing and are also transferred to the GPU since they are necessary for calculating the WCC.

Each iteration of the refinement phase is performed entirely on the GPU. Every iteration consists of three steps which we will briefly discuss: update the labels of the vertices, collect community statistics and calculate the new WCC.

The first step is updating the label of the vertices. To maximize the parallelism of our implementation, we consider all possible actions for all vertices in parallel. Next, we chose the action for each vertex which leads

[2] `https://github.com/Het-SCD/Het-SCD`

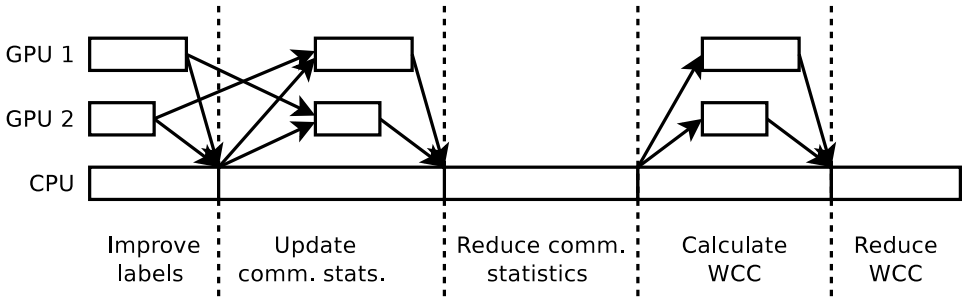

Figure 2: Steps of refinement phase for our heterogeneous version when using two GPUs.

to the largest improvement in WCC. A vertex adopts its new label if the improvement is positive, otherwise it keeps its current label.

The next step is collecting the community statistics of the new partition. We clear the statistics of the old communities and use atomic operations to count the number of vertices and the number of external and internal edges of each community in parallel

The final step is calculating the WCC of the new partition. For this step, we first replicate the entire graph and remove from this new graph all edges between different communities. Next, we assign one thread to every edge and calculate the values of $t(x, C_x)$ and $vt(x, C_x)$ for every vertex x. Finally, we compute the WCC for each vertex using Equation 1 and perform a reduction to calculate the average. The result is transfered back to the host, which decides whether another iteration needs to be performed.

3.2 The Heterogeneous Version

We have also implemented a heterogeneous version of SCD which can run using one CPU and/or one or more GPUs. This version requires each vertex to be assigned to either the CPU or a GPU, resulting in a partitioning of the network into components. The vertices that belong to the same component are the *core* vertices of that component and the vertices adjacent to the core vertices are the *halo* vertices. The whole graph is available in the memory of the CPU. For each GPU, we transfer the partition consisting of the core vertices and the halo vertices to the memory of the GPU before the refinement phase. For our initial prototype, we have used METIS [3] to find a partition of the graph with a low cut size.

Since all the steps in the refinement phase are vertex-centric and the graph has been divided over the CPU and GPUs by distributing the vertices, processing the vertices on CPU and GPUs in parallel is possible. The source code used to process the vertices on the CPU is parallelized using OpenMP. The vertices which are processed on the GPUs are masked out in the OpenMP implementation.

Table 1: Properties of networks used for experiments. The memory footprint on GPU is measured as the maximum amount of memory allocated by the GPU.

Name	Type	Vertices	Edges	Size in mem. (MB)	GPU footprint mem. (MB)
Amazon	Co-purchasing network	334,863	925,872	23.0	54.9
DBLP	Coauthorships network	317,080	1,049,866	27.2	69.8
YouTube	Social network	1,134,890	2,987,624	30.9	92.9
LiveJournal	Social network	3,997,962	34,681,189	562.8	1778.6
Orkut	Social network	3,072,441	117,185,083	1720.0	5026.6

Table 2: Properties of platforms used for experiments.

GPU Name	Micro-architecture	CUDA capability	SM count	Core clock rate (Mhz)	Memory (MB)	Bandwith (GB/s)	Host CPU
C2050	Fermi	2.0	14	575	2688	144.0	Intel Xeon E5620
GTX480	Fermi	2.0	15	700	1536	177.4	Intel Xeon E5620
GTX580	Fermi	2.0	16	772	3072	193.0	Intel Xeon X5650
GTX680	Kepler	3.0	8	1006	2048	192.3	Intel Xeon E5620
Tesla K20m	Kepler	3.5	13	706	5120	208.0	Intel Xeon E5-2620
GTX-Titan	Kepler	3.5	14	837	6144	288.0	Intel Xeon E5620

The CPU and GPUs need to communicate after each step of every iteration as illustrated Fig. 2. After updating the labels, all devices need to exchange data to update the labels of their halo vertices. After collecting the statistics of the communities in its partition, each GPU transfers their statistics to the CPU, the CPU adds all the results up, and transfers the final community statistics back to the GPUs. After calculating the average WCC of its vertices, each GPU sends its average to the CPU which, in turn, computes the overall average WCC for the entire graph.

4 Evaluation

To evaluate the performance and scalability of our solution, we have performed benchmarks on a set of five networks and six platforms. The networks used for these benchmarks are from the SNAP [4] repository and are listed in Table 1. The platforms we used are listed in Table 2. We only focus on the performance of the *refinement phase* in this section, since this

All our results are averaged over 5 runs and errors bars have been omitted since results are stable. The improvement threshold of the algorithm was set to 1% [9].

4.1 The CPU Version

We will first evaluate the performance of the CPU version, which has been parallelized using OpenMP and is based on the source code provided by Prat et al. [9] [3]. Fig. 3, 4, 5, 6 and 7 show the average time per iteration for five different networks, two different dual-quad-core CPUs and the number of threads ranging from a single thread to 16 threads. The results show that in all cases, the Intel Xeon E5620 is slightly faster than Intel Xeon X5650. This is expected since the X5650 has more memory bandwidth and a higher core clock rate. Using more threads reduces the execution time and the best performance is always obtained when occupying all cores by using 16 threads.

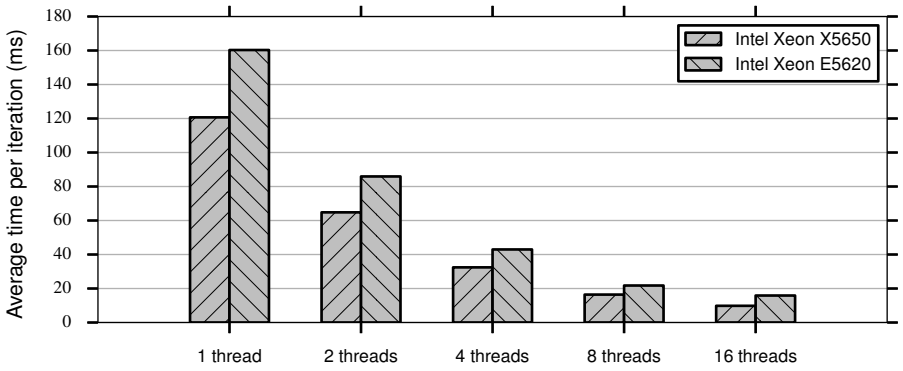

Figure 3: CPU performance on Amazon

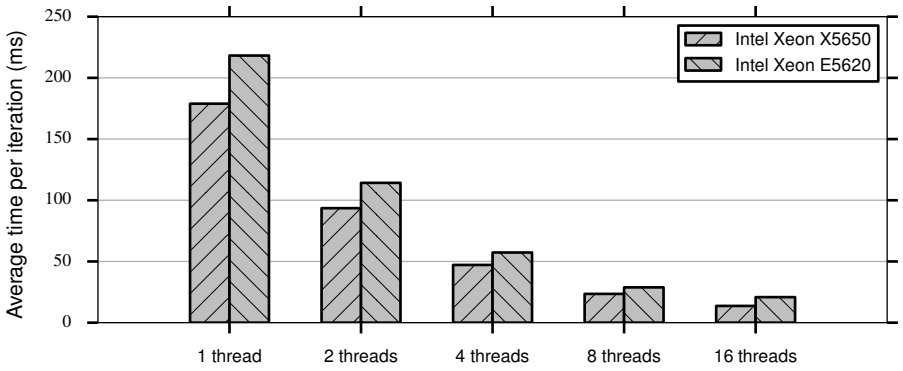

Figure 4: CPU performance on DBLP

[3] `https://github.com/DAMA-UPC/SCD`

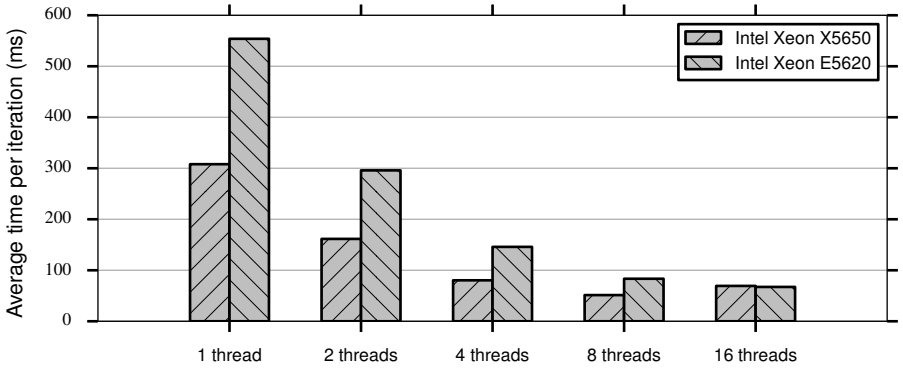

Figure 5: CPU performance on YouTube

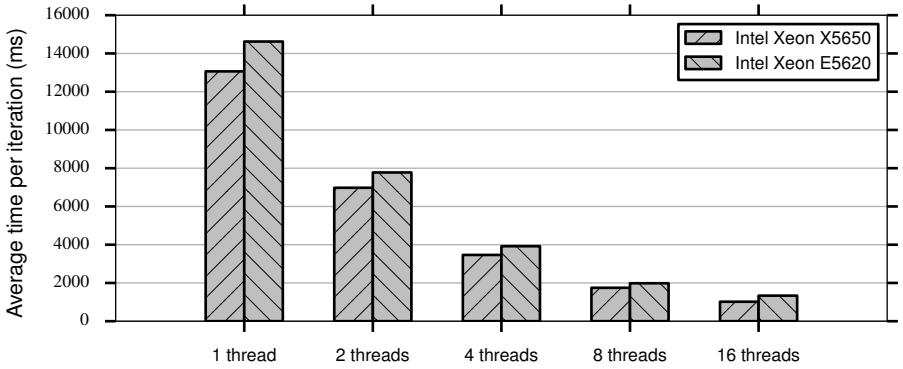

Figure 6: CPU performance on LiveJournal

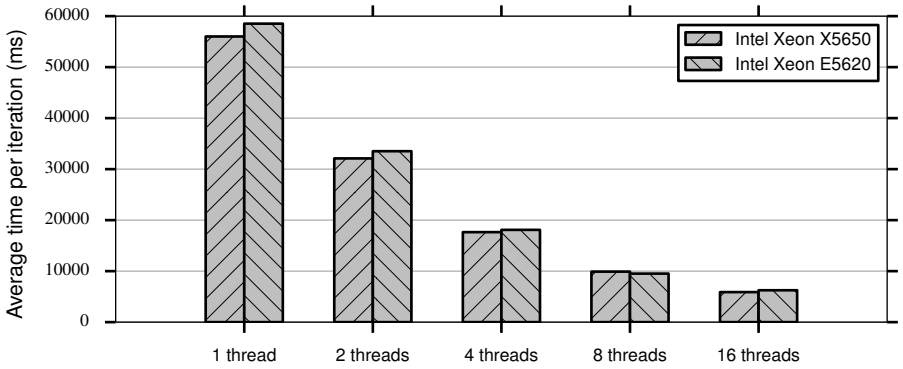

Figure 7: CPU performance on Orkut

4.2 The GPU Version

Next, we will evaluate the performance of the GPU version. Fig. 8, 9, 10, 11 and 12 show the average time per iteration for the five different network and the six different GPUs. The results for the two CPUs from the previous section have been included as well for comparison reasons. The results show that in almost all cases, the GPU is faster than the CPU, the only exception is the C2050 which is slower than the Intel Xeon X5650 for the two smallest graphs. The C2050 is always the slowst GPU, this is expected since it has the smallest bandwidth and lowest clock frequency. The GTX-Titan is always the fastest GPU, except for Youtube where the K20m is slightly faster. This is unexpected since the GTX-Titan has larger bandwidth, higher clock frequency and more SMs than the K20m. Note that LiveJournal and Orkut are too large to be processed on some GPUs.

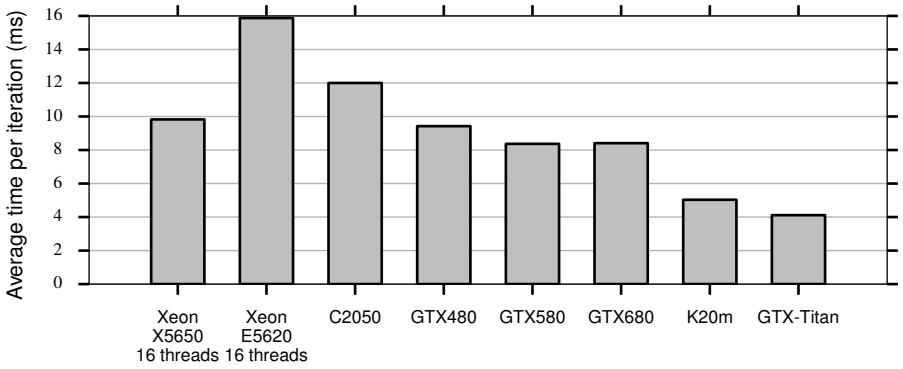

Figure 8: GPU performance on Amazon

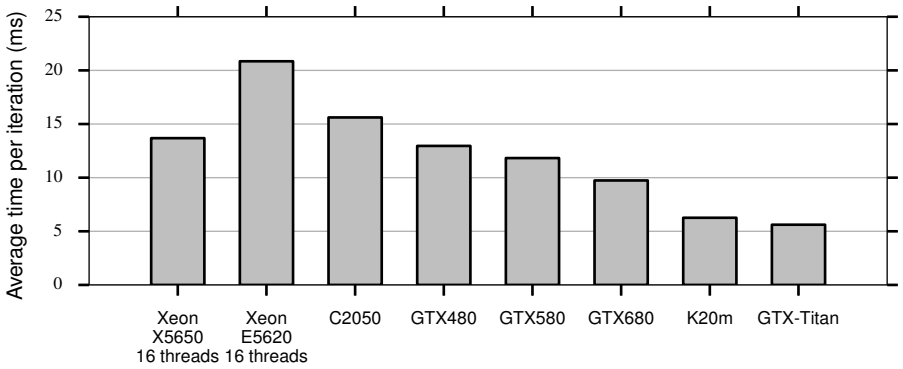

Figure 9: GPU performance on DBLP

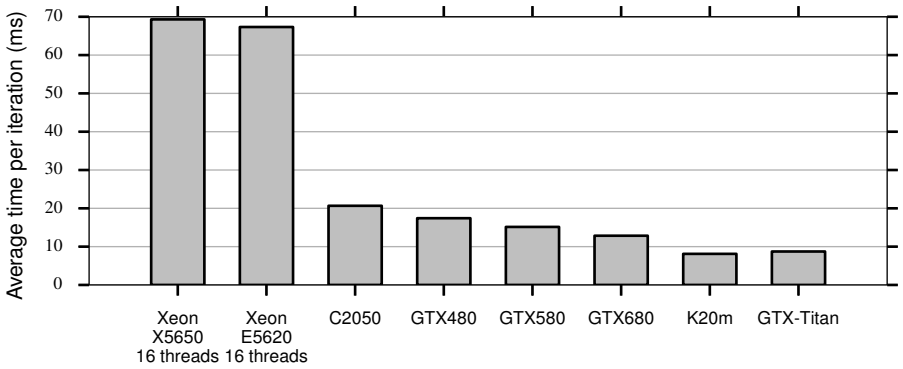

Figure 10: GPU performance on YouTube

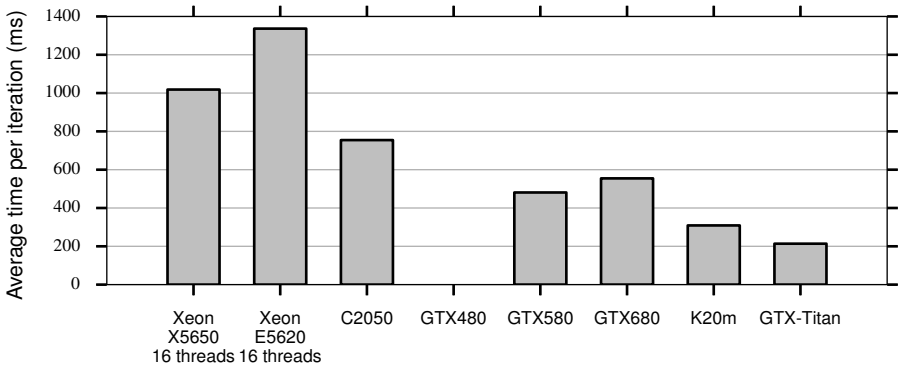

Figure 11: GPU performance on LiveJournal. Missing bars indicate failures due to insufficient memory.

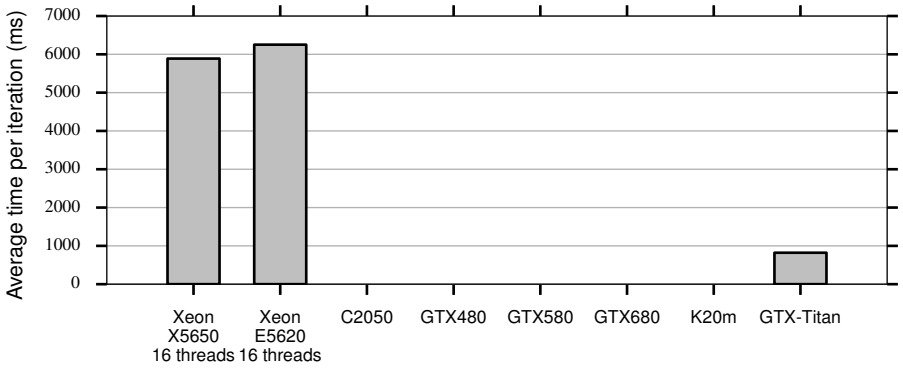

Figure 12: GPU performance on Orkut. Missing bars indicate failures due to insufficient memory.

4.3 The Multi-GPU Version

The heterogeneous version allows us to use multiple GPUs. Using multiple GPUs does not always increase the performance due to communication between devices. Increasing the number of devices also increases this communication overhead. Fig. 13, 14, 15, 16 and 17 show the results for one to eight GTX580 GPUs when evenly dividing the graph over the devices.

The results clearly show the effect of the communication overhead. The number of GPUs to use to obtain best performance differs per graph. For Amazon, a single GPU is ideal. For DBLP and YouTube, three GPUs give the best performance, although the improvement compare to a single GPU is not significant (less than 15% improvement). For LiveJournal, the best number of GPUs is five (62% improvement). Orkut requires at least six GPUs to run and using all eight GPUs give best performance.

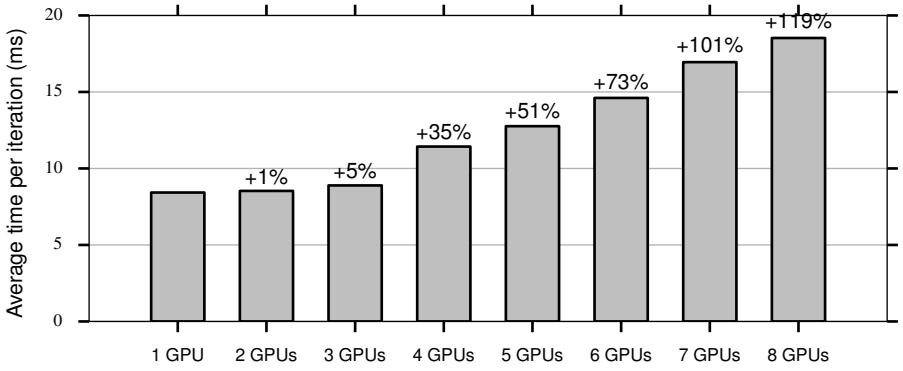

Figure 13: Multi-GPU performance on Amazon. Percentages indicate improvements in execution time over a single GPU.

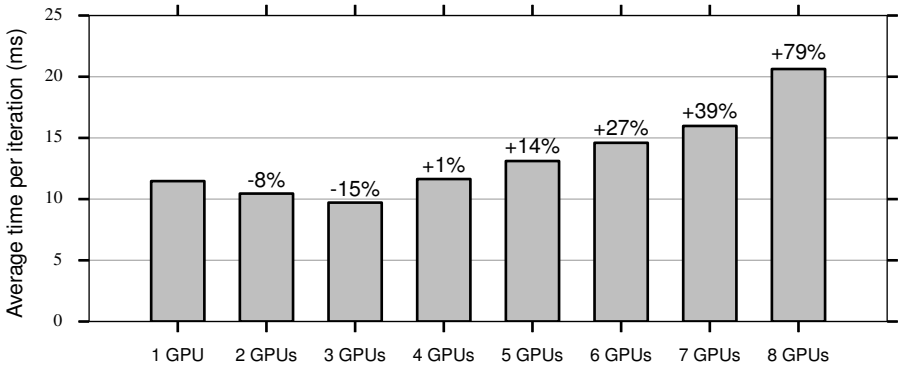

Figure 14: Multi-GPU performance on DBLP. Percentages indicate improvements in execution time over a single GPU.

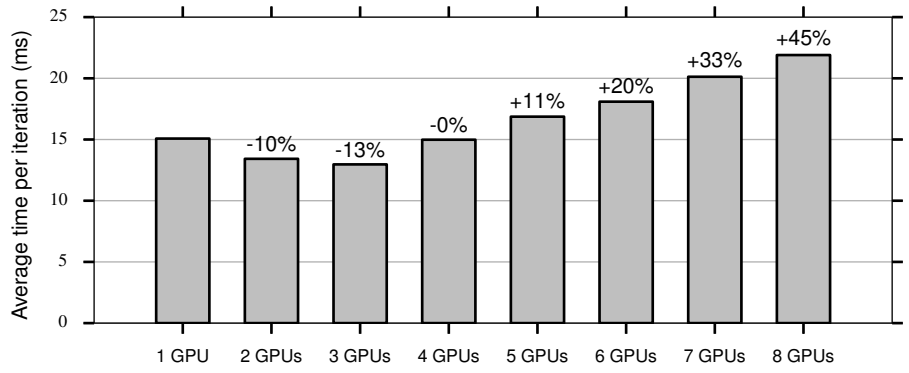

Figure 15: Multi-GPU performance on YouTube. Percentages indicate improvements in execution time over a single GPU.

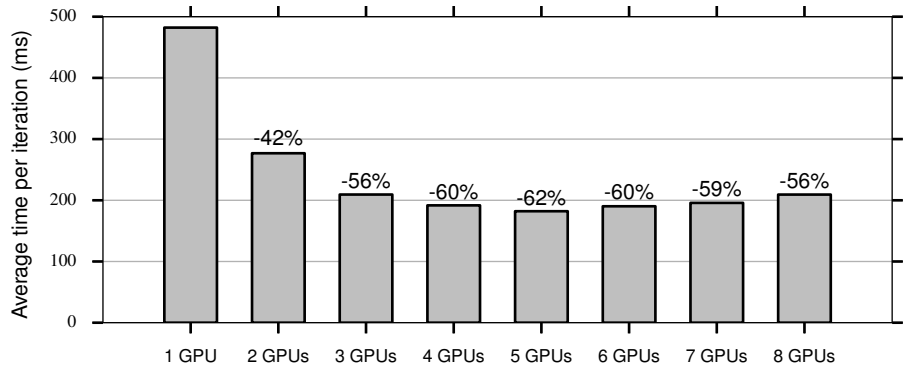

Figure 16: Multi-GPU performance on LiveJournal. Missing bars indicate failures due to insufficient memory. Percentages indicate improvements in execution time over a single GPU.

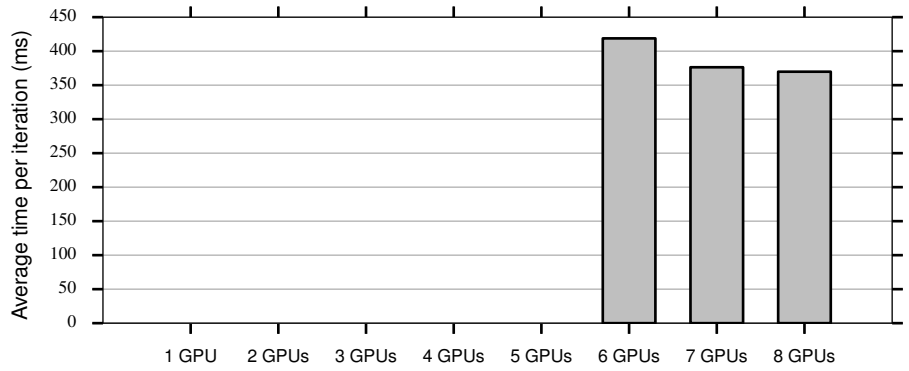

Figure 17: Multi-GPU performance on Orkut. Missing bars indicate failures due to insufficient memory.

4.4 The Heterogeneous Version

The heterogeneous version also allows us to assign a fraction of the vertices to the GPU and process the remaining vertices on the CPU. Fig. 18, 19, 20, 21 and 22 show the results for the five graphs and six platforms when varying the fraction of vertices on the CPU from 0% to 100%. Note that the platforms have different host CPUs which leads to different results for the CPU (see Table 2).

The results show that for the three smallest graphs, the best performance is always obtained by assigning all vertices to the GPU and not using the CPU. For these graphs, we can also very clearly see a drop between 90% and 100% vertices on the host. This is caused by the communication overhead which disappears when the GPU is not used. For LiveJournal, we can see that best performance is obtained when between 10% and 30% of the vertices are on the CPU, although the improvement over not using the CPU is not significant.

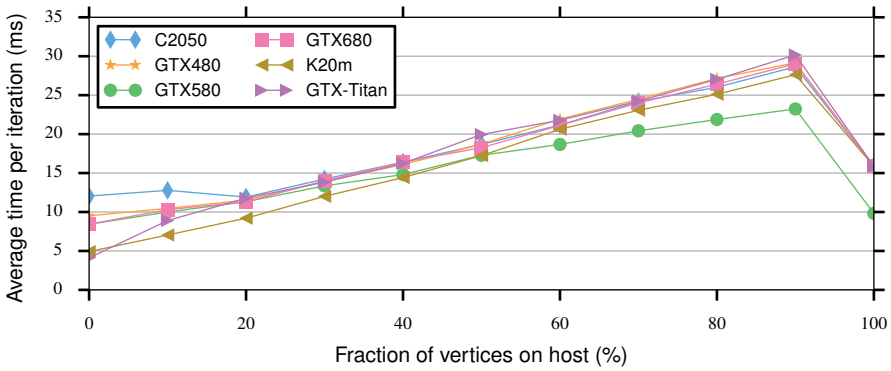

Figure 18: Heterogeneous performance on Amazon

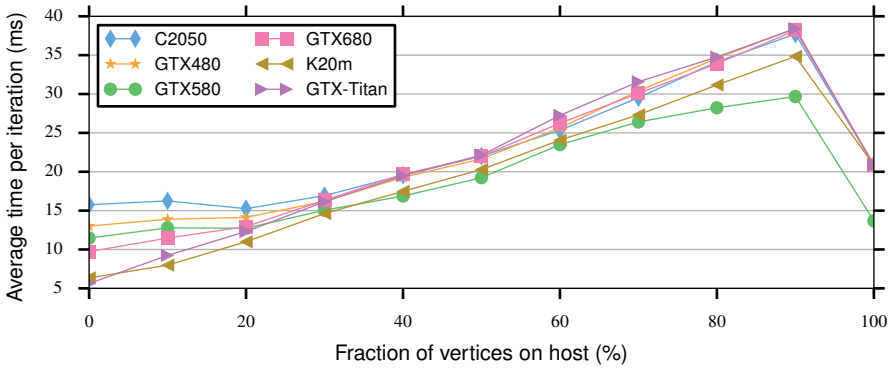

Figure 19: Heterogeneous performance on DBLP

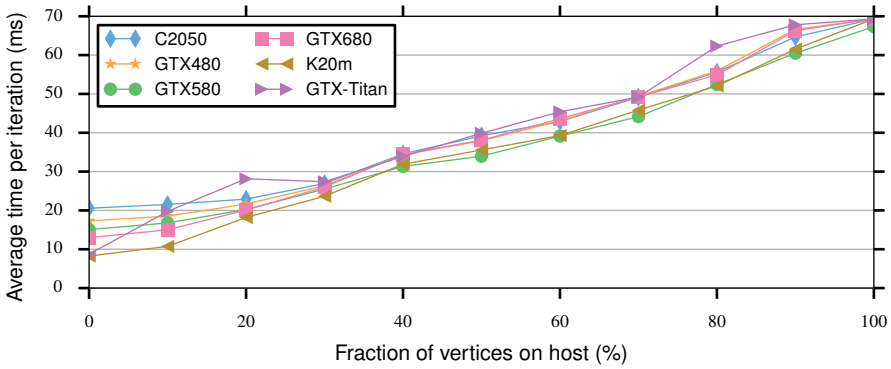

Figure 20: Heterogeneous performance on YouTube

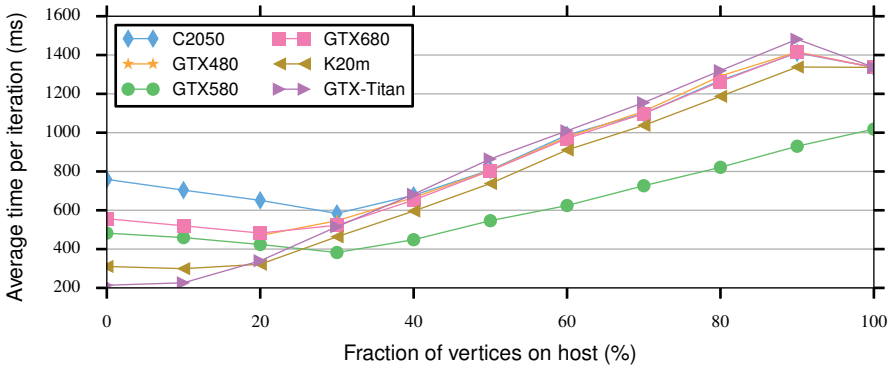

Figure 21: Heterogeneous performance on LiveJournal. Missing points indicate failures due to insufficient memory.

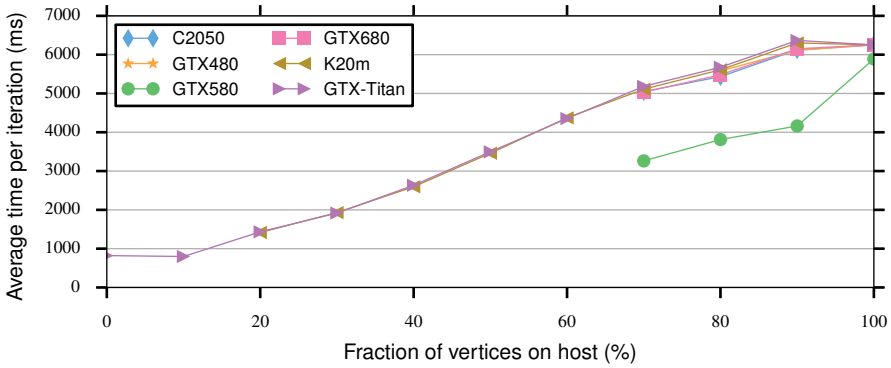

Figure 22: Heterogeneous performance on Orkut. Missing points indicate failures due to insufficient memory.

Fig. 16 and 17 show that using more GPUs reduces the execution time for LiveJournal and Orkut. Fig. 21 and 22 show that combining the CPU and one GPU also reduces the execution time for these two graphs. In Fig. 23 and 24 we can see the effect of combining these two strategies. A fraction of the vertices is assigned to the CPU and the remaining vertices are evenly divided over one, two, four or eight GTX580 GPUs. For LiveJournal, this approach does not seem to be beneficial. When using two or more GPUs, processing some vertices on the CPU does not increase performance. For Orkut, on the other hand, this approach is very beneficial. Since this graph is very large and requires at least six GTX580 GPUs to be processed (See Fig. 17), less than six GPU can also be used by assigning as much vertices as possible to each GPU and the remaining vertices to CPU. For example, when using four GPUs, the CPU only needs to process 50% of the vertices to be able to fit into memory.

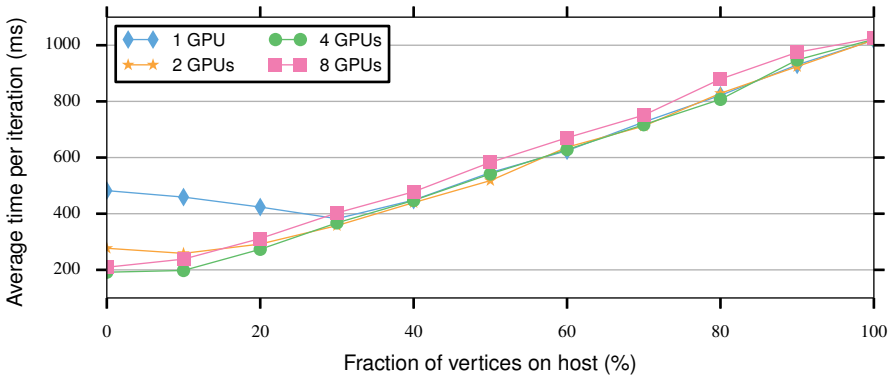

Figure 23: Heterogeneous performance on LiveJournal with multiple GPUs. Missing points indicate failures due to insufficient memory.

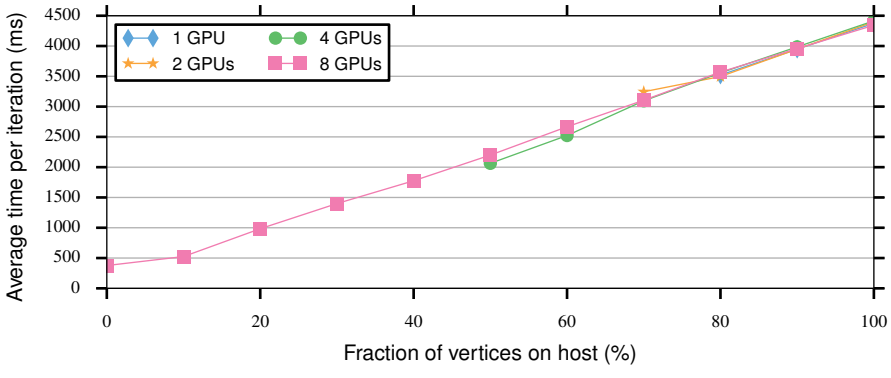

Figure 24: Heterogeneous performance on Orkut with multiple GPUs. Missing points indicate failures due to insufficient memory.

5 Conclusions

Community detection has become an important topic in the last decade due to its applications in many different fields of research. SCD is a community detection algorithm which can deal with the performance and accuracy requirements of real-world application. We have designed the first GPU version of SCD and extended this version to a heterogeneous version. We have performed benchmarks to measure the scalability and performance of our solutions. In this work, we have presented the results of our benchmarks of our version for six platforms and five networks.

The results show that the GPU almost always outperforms the CPU and these speedups are significant. For example, comparing the performance of the GTX-Titan to the Intel E5620 shows speed-ups of ∼4x on small networks (Amazon, 0.9M edges) to ∼6.5x on larger networks (Orkut, 117M edges). Based on these observations we suggest that one should always use the GPU instead of the CPU. The only exception might be networks which are much smaller than the networks we have used for our benchmarks. Based on our intuition, we estimate that this threshold might be around 0.1M edges, which is an order an magnitude less than our smallest graph (Amazon).

The results also show that using multiple GPU can lead to an increase in performance. However, using too many GPUs decreases performance due to communication overhead. For the three smallest networks (Amazon, DBLP, YouTube, below 3M edges) from our benchmarks, using more multiple GPUs never leads to a significant improvement in performance (less than 15%). For larger graphs, the number of GPUs to use to obtain the best performance differs. Based on these observations and our intuition, we propose the following rule of thumb: partition the network over the available GPUs such that each GPU has at least 5M edges.

Finally, the results also show that it is almost never beneficial for performance to assign a fraction of the vertices to the GPU and the remaining vertices to the CPU. In all cases, the increase in performance compared to the GPU-only version is not significant. Based on these results, we suggest to assign as much vertices as possible the GPU by filling its memory completely and process the remaining vertices on the CPU.

6 References

[1] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[2] Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515, 2004.

[3] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[4] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[5] Mark EJ Newman. Detecting community structure in networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):321–330, 2004.

[6] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[7] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[8] Arnau Prat-Pérez, David Dominguez-Sal, Josep M Brunat, and Josep-Lluis Larriba-Pey. Shaping communities out of triangles. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1677–1681. ACM, 2012.

[9] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-LLuis Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. International World Wide Web Conferences Steering Committee, 2014.