e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022

© 2012 IJFANS. All Rights Reserved

ANIMATED GRAPHICS: INTERPRETING SOLUTIONS TO NUMERICAL MODELS ON SUPERCOMPUTERS

SUBASHINI. S
Research Scholar
M.Phil Mathematics
Bharath Institute Of Higher Education And Research
Mail Id : mselvarajan77@gmail.com

Dr. N. RAMYA
Head of the Department, Department Of Mathematics
Bharath Institute Of Higher Education And Research

Address for Correspondence

SUBASHINI. S
Research Scholar
M.Phil Mathematics
Bharath Institute Of Higher Education And Research
Mail Id : mselvarajan77@gmail.com

Abstract

Present and future supercomputers offer many opportunities and ad vantages to attack complex and demanding industrial and applied mathematical problems, but provide also new challenges. In the Peta Flops regime, these concern both, the way to exploit the increasingly available power and the need of designing algorithms which are scal able and fault-tolerant at the same time. An example of a probabilistic domain decomposition method, which is indeed scalable and naturally fault-tolerant, is presented. Grid computing should also be mentioned as an increasingly popular way to perform massively distributed com puting: it represents a way to exploit computing power, aside the ex isting supercomputers. Beyond classical supercomputers there is the prospective quantum computer, in view of which it is advisable to start now a search for suitable algorithms for entire classes of problems. Due to the ever more challenging problems put forth by the needs of scientific computing, computing power increases continuously, making

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

it possible to cope with more complex applications. These problems include both, purely scientific advances and very practical industrial needs.

Hybrid Comparison Sorting

Sorting is a problem of fundamental importance in Computer Science with a rich history of algo rithm design, analysis, and engineering. Several parallel algorithms and their corresponding efficient implementations targeted at modern architectures including the Cell BE ,GPU based works including , the Intel MIC, are being studied. However, all of the above works utilize only a homogeneous device.

Given the importance and relevance of hybrid computing, in this work we propose a hybrid algorithm for sorting on a CPU+ GPU platform. We specifically consider comparison based sorting algorithms for reasons of wide applicability to settings such as variable length keys, and database records. We extend the algorithm presented in , which is a natural extension of the standard quick sort , to operate in a heterogeneous setting. The basic idea of sample sort is to choose $k - 1$ pivots, or *splitters*, from the input list. The input list is then split into k disjoint lists each containing roughly n/k elements. Each of the sub-lists can be sorted independently. Typically, a recursive approach is taken to reduce the size of the sublists further.

Motivation

We observe that in most GPGPU based computing, the CPU is practically idle in the computation process. This leads to inefficient resource usage, more so as the computational power of present gen eration multicore CPUs is on the rise. Hence, to improve performance, we use such a hybrid CPU and GPU system and target full resource utilization. We call this as *hybrid multicore computing*, or *hybrid computing* in short. Hybrid computing is gaining tremendous research attention of late given that issues such as power and performance dominate parallel computing.

Further, in a recent influential work , the authors argue and provide evidence for showing that on a diverse collection of 14 workloads, GPUs can offer only modest performance

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022

© 2012 IJFANS. All Rights Reserved

advantage compared to multicore CPUs. Sorting is one of the workloads considered in where it is shown that the GPU is on an average only 1.5 times faster than the CPU. We interpret the message of as not to compare one device against the other, but to study the benefits of using both the devices simultaneously. We call this as *hybrid computing*.

Related Work

Radix sort algorithms are some of the most efficient algorithms that have been implemented on GPUs and other multicore architectures. Radix sort is one of the easiest algorithms to be implemented in parallel machines because of its reducibility to a popular primitive which is the *scan* or parallel prefix operation. In works such as , the authors have shown the use of the scan and split operation for efficiently implementing the radix sort routine. A popular randomized parallel algorithm for radix sorting was proposed by Helman et al. in. The most popular recent implementation of radix sort was proposed by Merrill and Grimshaw . However, radix sort algorithms suffer from a basic bottleneck, where the sorting process becomes computationally more expensive with the increase in the size of the keys.

Merge sort is another popular sorting algorithm which recursively merges multiple sorted sub sequences into a single sorted sequence. The first parallel merge sorting algorithm was proposed by Richard Cole . In the earliest works on merge sorting on the GPU, the work of Purcell et al., is of high importance. The current best result in comparison sort on GPUs is by Davidson et al. The work also provides several insights into efficient implementation on GPUs by reducing memory access latencies, improving register utilization and reducing segmentation.

In the following, we now describe the changes required for executing the algorithm in a hybrid manner. One important factor that our algorithm design addresses is to aim for load balance between the CPU and the GPU and also within the GPU. In all our algorithms, we have used labels such as CPU, GPU and CPU→ GPU. The label CPU refers to computations that take place in the CPU, and GPU refers to those on the GPU. The label CPU→ GPU refers to the data transfers that is happening between the CPU and the GPU.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

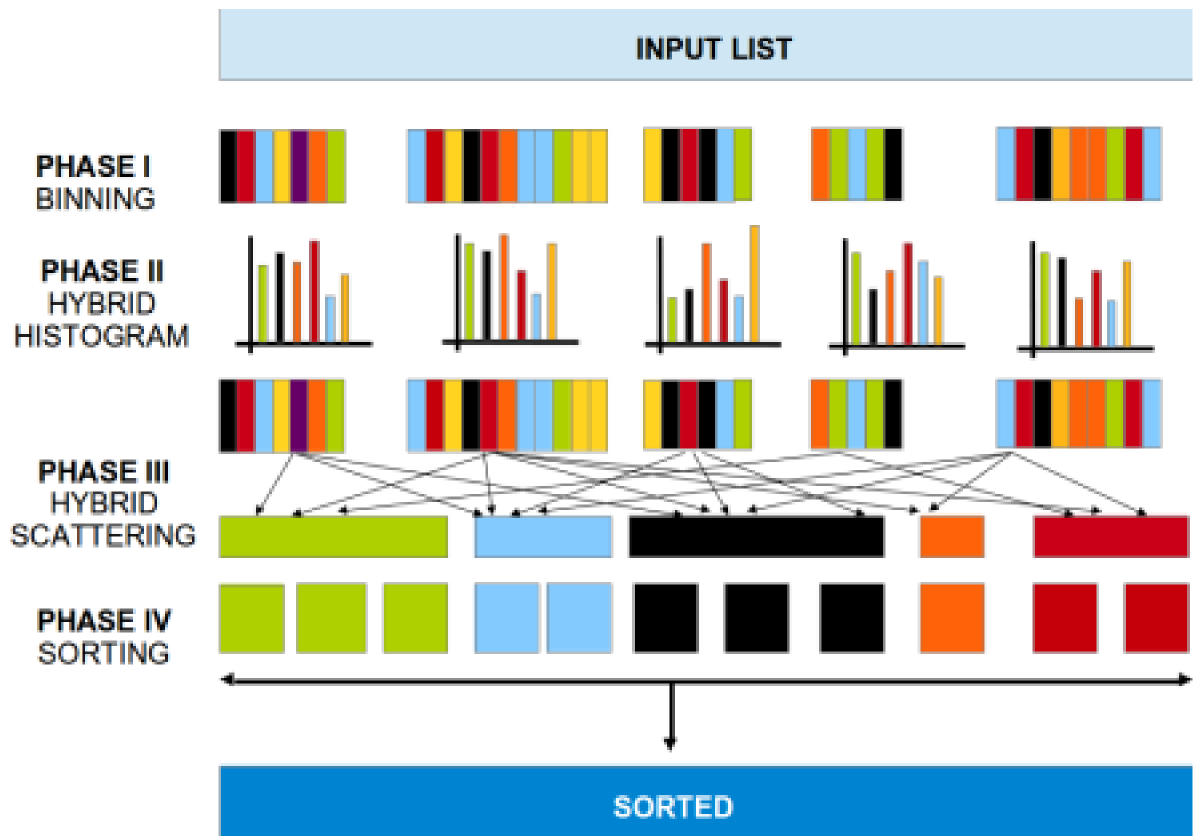
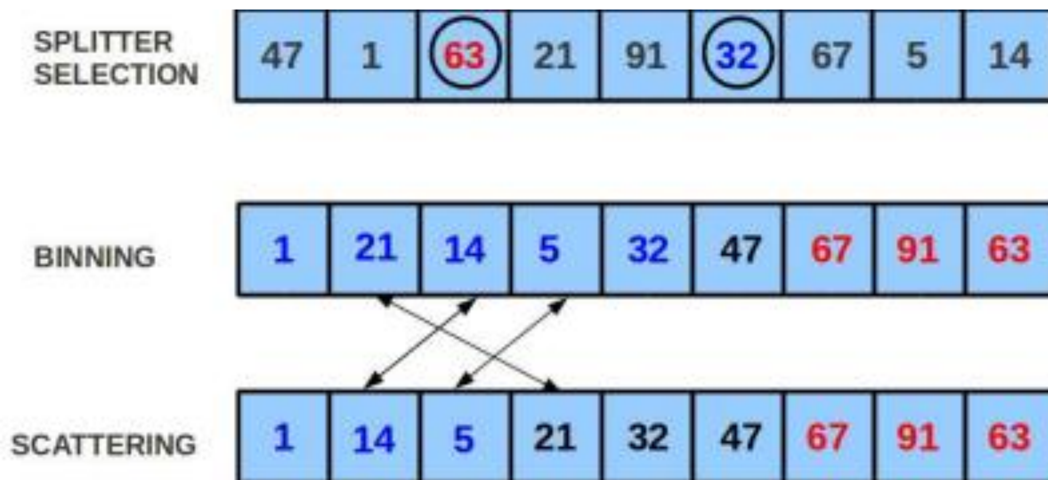


Figure 3.1 Different phases in hybrid sorting. The different colors represent the different bin labels which are brought together by scattering.



e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

Figure 3.2 An example run of our algorithm on a sample input of 9 elements. In this, we show the first pass of the algorithm on our input list, that creates the first set of bins.

Algorithm 1 HybridSort()

1: ***PHASE I :*** Creating bins.
2: ***PHASE II :*** Hybrid Histograms on CPU and GPU.
3: ***PHASE III :*** Scattering elements on both CPU and GPU.
4: ***PHASE IV :*** Recurse and sort small bins.

Phase I

Phase I involves selecting splitters as a uniform sample of the input list and then identifying the bin into which each element can be assigned to based on the splitters. we build a binary search tree of the splitters. The bin to which an element belongs to can then be efficiently identified by searching in the binary search tree. Having such a binary search tree is efficient since it reduces thread divergence of a block of threads in a GPU. Such techniques have been found to be useful in also multicore CPUs . As we see in Algorithm 2, we now partition the list of labels into two parts each of which can be handled by the CPU and the GPU respectively. Let us call these lists as L_c and L_g respectively. As of now we employ the static partitioning strategy.

The static scheme of distribution of work have often been used in several popular benchmarks such as ScaLAPACK. The static strategy has been found to be good because of its optimal communication costs, lesser synchronization overheads and scalable load balancing properties. we show how the threshold of separation of the work between the two devices have a bearing on the final result. Also, this parameter helps in another way by enabling us to auto tune the entire application.

After the separation of the data between the two devices, each of them now completes the binning process by using the binary tree formed using the splitters. To facilitate a hybrid execution of the other phases, we now associate each element with its bin number. We call these

© 2012 IJFANS. All Rights Reserved

bin numbers as *labels*. This allows us to treat some of the following phases on inputs in the range [1, k] thereby simplifying the later phases.

Algorithm 2 Phase I($Integer * I, Integer\ n, BlockSize\ BLOCK$)

1: ***PHASE I : Create Bins***
2: Select elements from the list at intervals of $\sqrt{n}$ and from a binary search tree. n is the input number of elements.
3: CPU :: Set Threshold which determines the ratio of division of L
4: CPU :: Divide L as L_g for GPU and L_c for CPU
5: CPU→ GPU :: Transfer L_g to GPU
6: CPU :: Use tree to optimally divide L_c among several bins in overlap of transfer.
7: GPU :: Use tree to optimally divide L_g among several bins after transfer is complete.

Phase II

Algorithm 3 Phase II($Integer * L_c, Integer * L_g, BlockSizeBLOCK$)

1: ***PHASE II : Hybrid Histograms***
2: **For** each list L_g and L_c do in parallel
3: CPU :: Compute histogram H_c of L_c for $LEN/BLOCK$ elements
4: GPU :: Compute block-wise histogram H_g of L_g for $LEN/BLOCK$ elements
5: **endfor**

At the end of Phase I, elements that have a common label are scattered across the input. In Phase II, described in Algorithm 3 for each label, we count the number of elements that have this label. This is done by computing the histogram of the list L_c on the CPU and the histogram of the list L_g on the GPU. In our application we use histograms in the second phase for getting the frequency of each bin label that is present on each block.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

Phase I

The computation in this phase involves finding the bin number to which each input element belongs to. The bin number of an element is the number of splitters that are smaller than the element. On the GPU, if all the elements are stored in contiguous locations, then one can benefit from a large number of coalesced accesses in this computation. A further optimization described in builds a height balanced binary search tree out of the splitters. At this point, finding the bin number of an element translates to a search in the binary search tree of splitters. One can also reduce thread divergence using this technique .

For the above reasons, we notice that indeed this phase can be executed entirely on the GPU. In fact, the time taken when this phase is executed on the GPU entirely is less than 2% of the overall time even for large inputs. However, we choose to perform this step also on the CPU and the GPU. This is justified by the fact that the input array is available only at the CPU at the beginning. For the GPU to start executing, the input array has to be made available at the GPU. In addition, since the other phases run simultaneously on both the CPU and the GPU, the output of this phase has to be sent to the CPU.

This involves an unnecessary data transfer step, which can be avoided if the bin numbers of a portion of the input is computed on the CPU.

We therefore choose a certain threshold and split the input array I into two parts, I_c and I_g. We choose splitters in I, and also find the bin numbers of elements in I_c on the CPU. The array I_g and the splitters are simultaneously transferred to the GPU. The GPU then computes the bin numbers of elements in I_g. At the end of the Phase 1 we have a list of bin labels that correspond to each of the input elements and is now called L_g and L_c for the GPU and the CPU parts respectively.

Phase II

Phase II computes the histogram of the bin labels. Computing histograms on the GPU is a well researched problem. Briefly, the entire computation is split into computing local histograms

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

at each of the SMs and then combining these histograms to arrive at a global histogram. One of the factors that affect the GPU performance is the number of threads that should be launched on the GPU so as to use to entire bandwidth on the GPU. Within each SM, to compute a local histogram, threads use shared memory to improve memory coalescing. A fundamental bottleneck in the histogram computation on GPUs is that GPUs offer very low throughput when using atomic operations.

This however, helps the hybrid computing model, since as described in Algorithm 3, the CPU also computes the histogram on an independent input, L_c. This computation is data is simply added using atomic primitives supported by OpenMP . As we are having a constant block size, it is com paratively simpler to parallelize across all the available cores of the CPU. A manual unrolling of the histogram serves this purpose and uses all the six cores that are available on the CPU and also gives us a significant performance benefit. In addition to this, we ensure to carefully optimize the histogram computation of the histogram on the CPU. We read in a certain tile of data into the L2 cache of the CPU. This tile size is based on the size of the L2 cache on the Westmere CPU. We iterate in steps of this tilesize so that the entire data on the L2 cache is used and does not require to be used afterwards. Inside each of the unrolled loops, we now use a second tile size that reads data from the L2 cache into the local cache of each of the cores. These cores now maintain a histogram store on the shared local cache and performs the atomic increments. Each of the instructions issued for increments are vectorized so as to ensure proper SIMD execution. We ultimately get a bandwidth bound performance from the CPU which is marginally better than the GPU. At the optimal threshold, the GPU stays idle for only 2% of the entire Phase II time.

At the end of this histogram computation, we synchronize the two devices using the CUDA event synchronization functions. This is required since Phase III can start only after the histogram computation is completed on both the devices.

• Staggered: In this the input of n numbers is arranged in p buckets. In the buckets numbered 1 to $p/2$ (both inclusive), all the numbers are chosen uniformly at random from $[(2i -$

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

$1) \cdot 2^{31}/p,(2i)\cdot 2^{31}/(p-1)]$, where i is the bucket number. For buckets numbered $p/2+1$ to p, all the elements are chosen uniformly at random from $[(2i-p-2)\cdot 2^{31},(2i-p-1)\cdot 2^{31}/(p-1)]$.

• Bucket Sorted: In this an input of n numbers is organized into p buckets as follows. The first n/p^2 elements in each bucket are chosen uniformly at random from $[0, 2^{31}/p - 1]$, the second n/p^2 elements in each bucket are chosen uniformly at random from $[2^{31}/p, 2^{32}/(p-1)]$, and so on.

2. Variable length keys: For sorting strings we mainly use two different data sets. First we exper iment with a Protein Sequence Database which is 650 MB in size and has many protein sequences that are represented using the popular "FASTA". In this format, each of the sequences are up to 120 characters long. Apart from this, we also experiment with a data set of 500 MB size that is containing a set of random strings and are up to 500 characters in length.

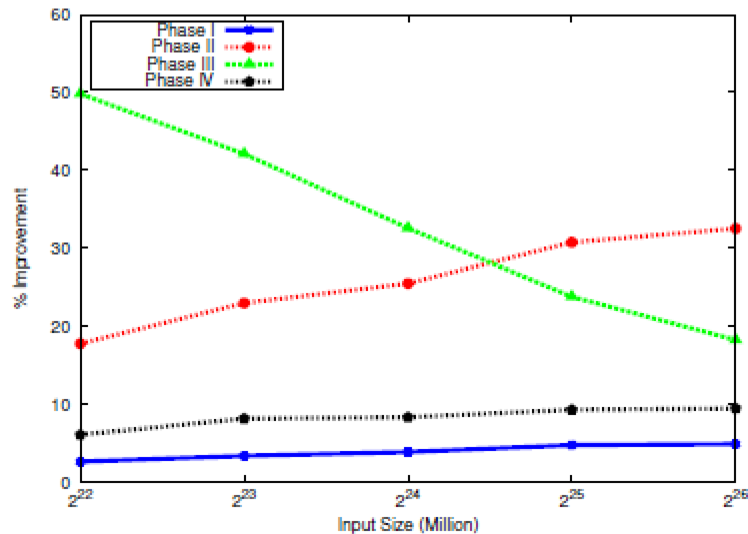


Figure 3.5 Percentage improvement over sample sort [81] at various phases.

Profiling, Resource Utilization, and Idle Times

One aspect of our hybrid implementation is that each phase of our hybrid algorithm runs faster than the corresponding phase in the pure GPU version from [81]. This is shown in Figure 3.5. In Figure 3.5, the times for Phase II and Phase III are only over the first iteration. This is justified since the overall time taken by Phase IV, which includes recursive calls to Phases I-III,

 © 2012 IJFANS. All Rights Reserved

is a very small portion of the entire runtime of the implementation. Also, this recursive calls happens over data that is present in each of the bins that has been created in the first iteration. Hence, the overall impact of the hybridization is felt mostly in the first iteration of the application. In Figure 3.5, we show the percentage improvement of our hybrid implementation over the pure GPU sample sort implementation. We now analyze the results of Figure 3.5.

In Phases II and III, we notice that there is a significant improvement that is achieved over the corresponding pure GPU phases. In both of these phases, we notice that there is a benefit of around 25%. Further, the improvement of Phase II increases as the size of the input increases. This is because of two reasons. Firstly, as the input size increases, the distribution of the numbers happen over a larger number of bins. Hence, the conflicts arising during the atomic computations are reduced. Also, the larger number of bins are now also divided among the CPU and GPU which leads to a higher degree of work distribution. This leads to a better works sharing and consequent increase in the gain of Phase II.

Notice from Figure 3.5 that performance gain of our hybrid implementation in Phase I and Phase IV is not very significant. This can be explained as follows. Phase I is not highly compute intensive. Indeed in our experimentation, we notice that Phase I if run entirely on the GPU takes only less than 1% of the total time. However, even this phase is done in a hybrid manner as that reduces the data communication

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

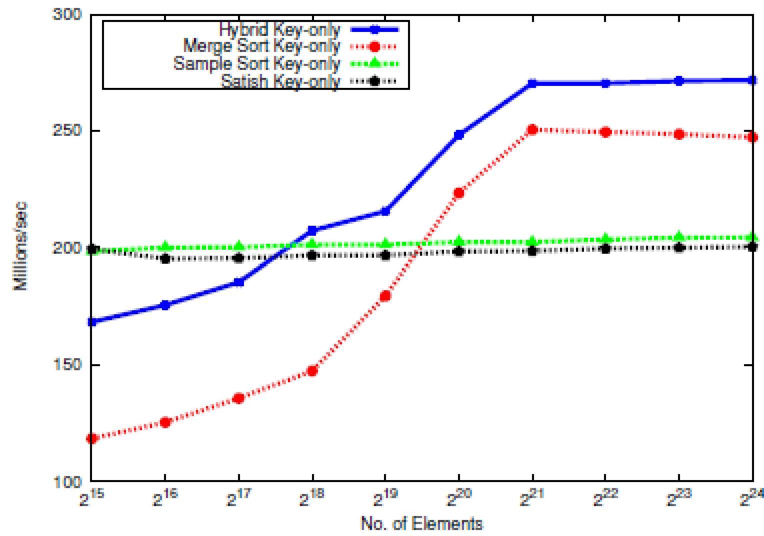


Figure 3.6 Performance on uniformly random keys sorting in high-end platform.

. Doing so has the other benefit that the CPU utilization improves, and hence the idle time reduces. This can be noticed also from Figure 4.4 where the phase-wise timings are shown for an input of size 4 M. We define idle time of a device as the total time for which the device is idle. The idle time of an implementation is then the maximum idle time experienced by any of the devices. In our algorithm, the GPU is idle for at most 5% of the total runtime, and the CPU is idle for at most 12% of the total runtime. So, the idle time of our implementation is 12%.

Results of Sorting

Results on Fixed Length Keys

In this section, we show the results of our implementation on various experimental datasets of fixed length keys. Since most of the GPU sorting algorithms are designed on 32 bit numbers, we have exper imented using the 32 bit integers as well. Results on 64 bit integers are shown later.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

The input is generated as a uniformly random dataset described earlier. In Figure 4.6, we report the performance of the 32 bit key only sorting. In this case, we see that we achieve almost a 23% improvement on an average over the closest best known result .

We now look at the sorting results for key-value pairs in Figure 3.7. As can be noticed from Figure 3.7, our hybrid implementation is on average 40% faster than the result of , and on average 20% faster than the result . This improvement can be attributed to the fact that the majority of the computation involves operations such as histogram and scattering which are very amenable to a hybrid execution environment.

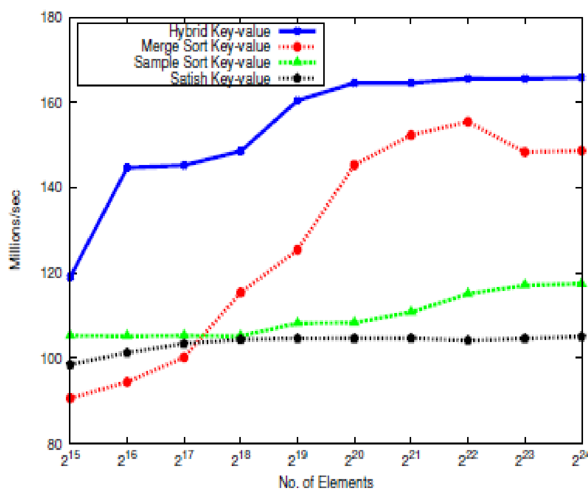


Figure 3.7 Performance on key-value high-end platform.

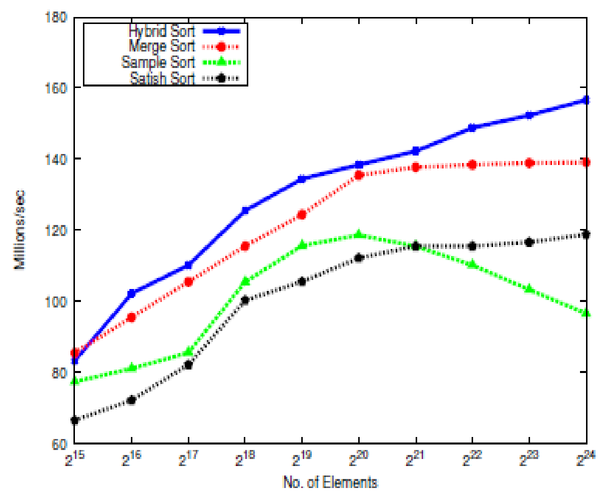


Figure 3.8 Performance on 32 bit pairs in Gaussian in put.

We report the behavior of our algorithm on the inputs such as Gaussian distributed, Deterministic Duplicates, Staggered, and Bucket sorted in the following. In Figure 3.8, we see the performance of our algorithm on the Gaussian distributed input. We achieve an improvement of 10% on an average on the Gaussian input because of the added overhead in the scattering phase. As the number of bins created in this case is higher than the other inputs, the scattering in the first phase of the algorithm consumes more time. In both Figure 3.7 and Figure 3.8, we notice a significant performance improvement only when the input size cross the threshold of 2^{18}

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022

© 2012 IJFANS. All Rights Reserved

elements. This can be attributed to the fact that on inputs greater than 2^{18} elements, the number of sublists that are created before the threshold of sorting is reached enables the GPU to achieve bandwidth saturation. Hence, the number of threads employed towards the whole GPU scattering process are higher which helps get higher gains.

Results on Other Platforms We also experimented on the Hybrid-Low platform described. We see the result of the sorting in Figure 3.15. We experiment using 64 bit key-value pairs on this platform and achieve a benefit of 18% on an average over the best known merge-sort implementation.

In the Figures 3.16, 3.17, we see the performance of our key only and key-value sorting across three different platforms that we have already explained. The sorting method performs the best with the K20c GPU as the GPU has a bigger sized L2 cache and offers a much higher degree of

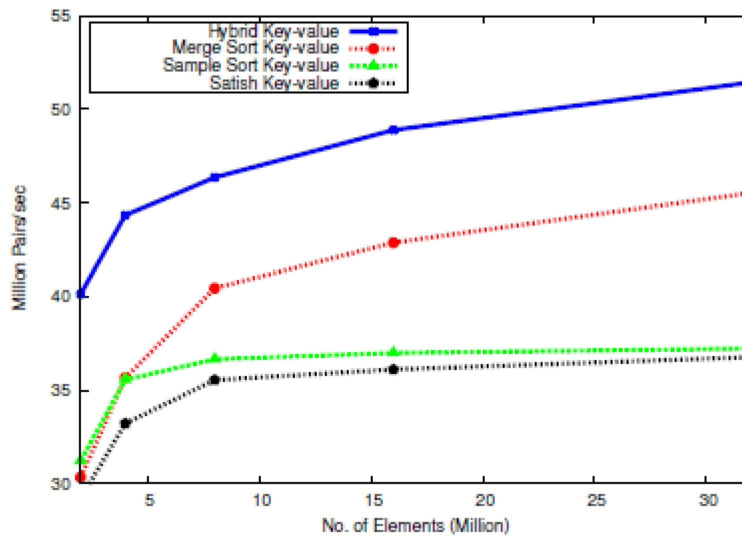


Figure 3.15 Performance on key value pairs on low-end platform.

parallelism. So, due to the L2 caching of the data, the overhead suffered during Phase III is significantly offset.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

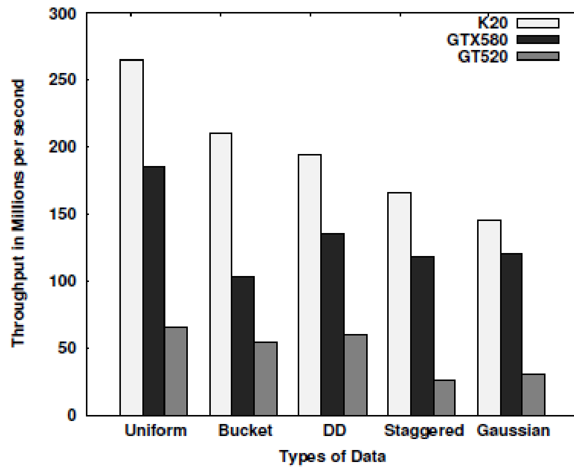


Figure 3.16 Performance of 32 bit sorting different platforms at input of 2^{21} uni formly random elements.

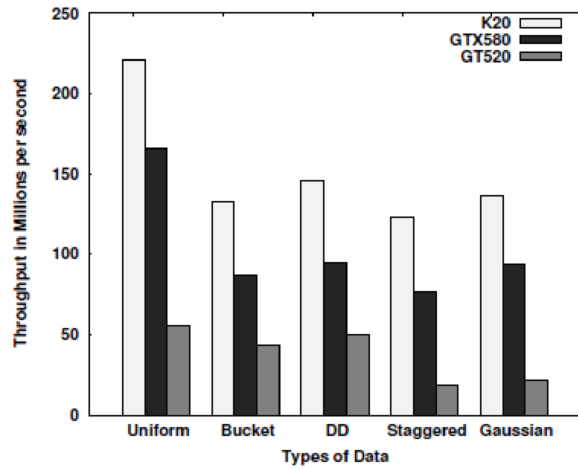


Figure 3.17 Performance of key value across sorting across different platforms at input of 2^{21} ele ments.

Conclusion

As is mentioned in several earlier works, the performance of the histogram on the GPU depends heavily on the utilization of the shared cache of the GPU and the minimization of atomic conflicts. Further, the GPU is not very amenable to such atomic operations. Hence, this computation has the potential to benefit from a hybrid computing model. This is verified in Figure 4.22, where we see that there is a gain of nearly 25% on an average over the pure GPU implementation. Such a gain is remarkable given that the peak FLOP rating of the CPU in our experimental platform is only a tenth of that of the GPU.

Our hybrid sorting clearly demonstrates the benefits that can be gained out of the use of heteroge neous processors that are most commonly available in today's commodity desktops and laptops. In this work we have implemented and verified our algorithm agains a wide array of inputs of fixed length as well as variable length keys. All the results definitively show the adavantage of heterogeneous imple mentations. In the near future we will be having completely heterogeneous processors such as the ones of AMD APUs and Intel Ivybridge. Hence, in hybrid programming and research in this area holds a lot of promise.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

In the near future we want to work on greater heterogeneous platforms such as the ones involving multiple GPUs and other multicore processors such as the Intel MIC. It will be interesting to see the performance of our sorting mechanism on such kind of platforms where there will be both tightly cou pled and loosely coupled processors. It will be our goal to arrive at a efficient performance model for such kind of platforms using various computing primitves like sorting, searching, ranking and graph algorithms.

REFERENCES

[1] OpenMP Application Program Interface - Version 4.0 - RC2 - March 2013 . http://www.openmp. org/mp-documents/OpenMP_4.0_RC2.pdf.

[2] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/ sparse/matrices/.

[3] V. Agarwal, F. P. D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of ACM SC*, page 111, 10.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Pro ceedings of the first annual ACM symposium on Parallel algorithms and architectures*, SPAA '89, pages 11–21, New York, NY, USA, 1989. ACM.

[5] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theory of Computer Science*, 71(1):3–28, Mar. 1990.

[6] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. Technical Report Computer Science Technical Report, ICL-UT-10-04, University of Tennessee, 2010.

[7] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processingunits for high-speed monte carlo simulation of photon migration. *Journal of Biomedical Optics*, 13(6), 2008.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

[8] R. J. Anderson and G. L. Miller. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[10] B. Awerbuch and Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, Oct. 1987.

[11] D. A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IEEE IPDPS*, pages 1–10, 2007.

[12] D. A. BADER and G. CONG. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994 – 1006, 2005.

[13] D. A. Bader and K. Madduri. GTgrpah: A suite of synthetic graph generators.

[14] S. Bandyopadhyay and S. Sahni. GRS-GPU radix sort for multifield records. In *International Conference on High Performance Computing (HiPC), 2010*, pages 1 –10, Dec. 2010.

[15] S. Bandyopadhyay and S. Sahni. Sorting large records on a Cell Broadband Engine. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 939 –944, June 2010.

[16] D. S. Banerjee and K. Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. of IEEE 18th Annual International Conference on High Performance Computing (HiPC)*, 2011.

 © 2012 IJFANS. All Rights Reserved

[17] D. S. Banerjee, S. Sharma, and K. Kothapalli. Work Efcient Parallel Algorithms for Large Graph Explo ration. In *Proc. of IEEE 20th Annual International Conference on High Performance Computing (HiPC)*, 2013.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[19] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[20] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[21] D. Boas, J. Culver, J. Stott, and A. Dunn. Three dimensional monte carlo code for photon migration through complex heterogeneous media including the adult human head. *Opt. Express*, 10:159–170, Feb 2002.

[22] B. Bollob´as. *Random graphs*. Cambridge University Press, 2001.

[23] D. Cederman and P. Tsigas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[24] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[25] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389, 2012.

[26] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, Aug. 1988.

e-ISSN 2320 –7876 www.ijfans.org
Vol.11, Iss.9, Dec 2022
© 2012 IJFANS. All Rights Reserved

[27] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix sums and List Ranking. *Information and Compu tation*, 81(3):334–352, 1989.

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

[29] N. Corporation. Cuda: Compute Unified Device Architecture programming guide. Technical report, 2007.