Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Tomas Karnagel, Dirk Habich, Wolfgang Lehner

Limitations of Intra-operator Parallelism Using Heterogeneous Computing Resources

Erstveröffentlichung in / First published in:

Advances in Databases and Information Systems: 20th East European Conference. Prag, 28.-31.08.2016. Springer, S. 291-305. ISBN 978-3-319-44039-2.

DOI: http://dx.doi.org/10.1007/978-3-319-44039-2_20

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-839039

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

Limitations of Intra-operator Parallelism Using Heterogeneous Computing Resources

Tomas Karnagel(✉), Dirk Habich, and Wolfgang Lehner

Database Technology Group, Technische Universität Dresden, Dresden, Germany
{tomas.karnagel,dirk.habich,wolfgang.lehner}@tu-dresden.de

Abstract. The hardware landscape is changing from homogeneous multi-core systems towards wildly heterogeneous systems combining different computing units, like CPUs and GPUs. To utilize these heterogeneous environments, database query execution has to adapt to cope with different architectures and computing behaviors. In this paper, we investigate the simple idea of partitioning an operator's input data and processing all data partitions in parallel, one partition per computing unit. For heterogeneous systems, data has to be partitioned according to the performance of the computing units. We define a way to calculate the partition sizes, analyze the parallel execution exemplarily for two database operators, and present limitations that could hinder significant performance improvements. The findings in this paper can help system developers to assess the possibilities and limitations of intra-operator parallelism in heterogeneous environments, leading to more informed decisions if this approach is beneficial for a given workload and hardware environment.

Keywords: Intra-operator parallelism · Heterogeneous systems · Dataflow parallelism · Data partitioning · GPU

1 Introduction

In the recent years, hardware changes shaped the database system architecture by moving from sequential execution to parallel multi-core execution and from disk-centric systems to in-memory systems. At the moment, the hardware is changing again from homogeneous CPU systems towards heterogeneous systems with many different computing units (CUs), mainly to reduce the energy consumption to avoid Dark Silicon [5] or to increase the system's performance since homogeneous systems reached several physical limits in scaling [5].

The current challenge for the database community is to find ways to utilize these systems efficiently. Heterogeneous systems combine different CUs, like CPUs and GPUs, with different architectures, memory hierarchies, and interconnects, leading to different execution behaviors. Homogeneous systems can be utilized by using uniformly partitioned data for all available CUs. The original idea was presented in GAMMA [4] as dataflow parallelism, where data is split

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

and processed on multiple homogeneous processors. There, data partitioning is easy, while skew in the data values, data transfers, and result merging already complicate the approach.

We want to evaluate the same approach to heterogeneous systems in a fixed scenario. Different from homogeneous systems, CUs in heterogeneous systems have different execution performances depending on the operator and data sizes. Therefore, we first define a way to find the ideal data partitioning according to the different execution performances of the given CUs. Afterwards, the partitioned data is used to execute an operator, which computes a partial result. Finally, the partial results of all CUs have to be merged. In this paper, we analyze this approach for two operators, selection and sorting, on two different heterogeneous systems to evaluate the advantages and disadvantages. We present performance insides as well as occurring limitation to intra-operator parallelism in heterogeneous environments. As a result, we show that the actual potential of improvements is small, while the limitations and overheads can be significant, sometimes leading to an even worse performance than single-CU execution.

In Sect. 2, we present intra-operator parallelism in more detail, before presenting the operators and hardware environments for our analysis in Sect. 3. Afterwards, we analyze the selection operator in Sect. 4 and the sort operator in Sect. 5, before presenting learned lessons in Sect. 6.

2 Intra-operator Parallelism

As intra-operator parallelism in heterogeneous environments, we define the goal of minimizing an operator's execution by using all available heterogeneous compute resources. This means dividing input data into partitions, executing the operator on the given CUs, and merging the result in the end.

In the following, we discuss the general idea, an approach to find ideal partition sizes, and the possible limitations of intra-operator parallelism.

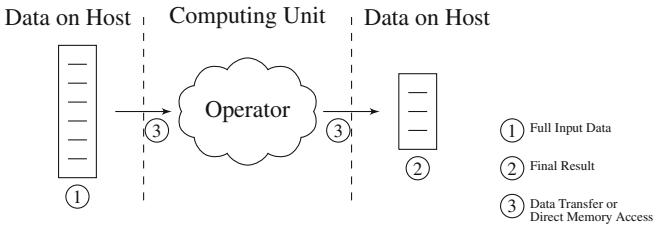


Fig. 1. Operator execution on a single computing unit.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

2.1 General Idea

Our starting point is the general operator execution on an arbitrary computing unit as shown in Fig. 1. We assume that the input data is initially stored in the system's main memory and that output data has to be stored in the same. Therefore, all our assumptions and tests include input and output transfer, if the CU is not accessing the main memory directly. We also assume that the operator implementation is inherently parallel and utilizes the complete CU, which should normally be the case when the operator is implemented with CUDA or OpenCL.

Having a system with heterogeneous resources, parallel execution between multiple CUs becomes possible. At this point, we focus on single operator execution, therefore, we want to execute the same operator concurrently on multiple CUs, each CU working on its own data partition. During operator execution, we want to avoid communication overhead through multiple data exchanges, so we choose an approach, where we partition the input data, execute the operator atomically on each CU with the given partitions, and merge the result in the end. Figure 2 illustrates this approach for two CUs.

While this approach is well studied for many operators in homogeneous systems, where multiple CPU cores or multiple CPU sockets are used, there is not much information about heterogeneous systems. In a homogeneous setup, the input data can be divided uniformly, since every CU needs the same amount of execution time. In a heterogeneous system, different CUs perform differently, so data has to be divided differently and multiple limitations could hinder the execution. Mayr et al. [9] looked at intra-operator parallelism for heterogeneous CPU clusters with the goal to prevent underutilization of available resources. They also present a detailed overview of related work. We, however, look at heterogeneity within one node with CUs like CPUs and GPUs, leading to different approaches and limitations.

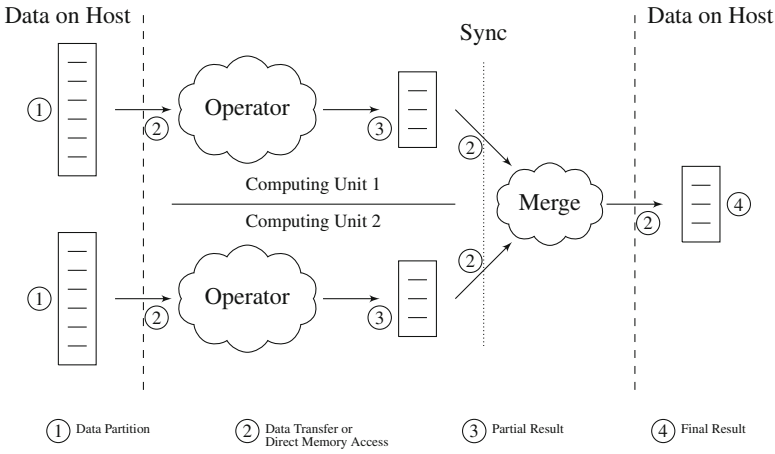


Fig. 2. Operator execution on two computing units.

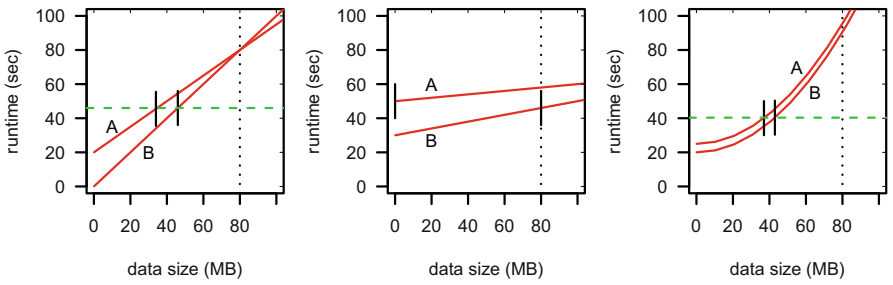
Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

2.2 Determining the Partition Size

In a first assessment, we want to look at the potential of intra-operator parallelism together with possible ways to determine the best data partition size.

The intuitive approach would be: when both CUs execute an operator with the same runtime, then the data is divided by two (50/50) and the potential speedup could be 2x. However, heterogeneous CUs usually show different execution behavior for an operator. There, even a slower CU can help improving the overall runtime by processing a small part of the work, however, different scenarios need to be considered. Figure 3 shows three scenarios of heterogeneous execution. The execution time for different data sizes is given for cuA and cuB. The goal for all three scenarios is to execute an operator with 80 MB of data and to partition the input data to achieve the best combined runtime.



(a) 1.74x speedup (42/58) (b) no speedup (0/100) (c) 2.25x speedup (46/54)

Fig. 3. Given two CUs (A, B) to simulate execution behavior in different setups. In this example, 80 MB need to be partitioned on cuA and cuB.

In Fig. 3a, both CUs show equal execution time at 80 MB, however, the best partition is not 50/50, but 42/58. This is caused by the slope of the execution behavior, resulting in different execution times for smaller data sizes. For example, when dividing 50/50, cuA runs for 50 s and cuB for 40 s, therefore, the concurrent execution would be 50 s (the maximum of both single-CU executions). This partitioning is not ideal. The goal is to achieve the same runtime on both CUs, which is 46 s when using 42/58 as partitioning. The speedup compared to a single-CU execution is 1.74x. Please note, for the remainder of the paper, speedups are always relative to the best single-CU execution.

Figure 3b shows a similar scenario with a different outcome. Here no data partition size is beneficial to improve the best single-CU execution. Parallel execution has no potential to improve the runtime and should be avoided. On the other side, if the execution behavior is exponential (Fig. 3c) then larger improvements are possible.

The question is how to calculate the best data partition size for heterogeneous CUs. Assuming we have k different CUs and we know the execution time ($exec_k$)

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

of an operator for different data sizes ($partition_k$), we can calculate the total execution time ($exec_{total}$) for a given input data size ($data\_size$) by:

$$exec_{total} = \max_{1 \leq k \leq n} (exec_k(partition_k))$$

$$\text{with} \sum_{1 \leq k \leq n} partition_k = input\_data\_size$$

Finally, we have to minimize $exec_{total}$ by adjusting the partition sizes ($partition_k$) to achieve the best result. Essentially, this function finds the partition sizes, where the execution for multiple CUs takes the same time. If that is not possible, this function also allows single-CU execution if one partition size is equal to $input\_data\_size$. Execution times for different data sizes can be collected through previous test runs or can be estimated by using estimation models [8].

2.3 Possible Limitations

While the presented function calculates ideal data partition sizes for ideal parallel execution, there are many factors involved with parallel execution that could potentially increase the overall runtime:

1. **Under Utilization.** For small data sizes, an operator might not be parallel enough to fully utilize a CU, e.g., highly parallel CUs like GPU and Xeon Phi, leading to slow execution. In that case, executing the operator with less input data leads to only small runtime reductions (e.g., cuA in Fig. 3b).
2. **Synchronization Overhead.** Parallel executions have to be synchronized in order to merge their results (as shown in Fig. 2). This synchronization could lead to delays and communication overheads.
3. **Merge Overhead.** After synchronizing the executions, the intermediate results have to be merged to generate a final result. This merge step strongly depends on the operator. Some operators, like selection or projection, do not have a time consuming merge step, while others, like joins or sortings, rely on complex compute intensive merges, reducing the potential of intra-operator parallelism significantly.
4. **Shared HW Resources.** CUs within one system are most likely to use shared resources that could become a bottleneck when using all CUs simultaneously. This could be interconnects to the host memory, the memory or DMA controller, or computing resources. When a workload produces contentions on these resources, the performance might suffer.
5. **Thermal Budget.** Modern CUs reduce their frequency, and therefore their performance, when a certain temperature threshold is reached. This is usually caused by the CU itself, however, the temperature can also increase indirectly through other CUs. The best example are tightly-coupled systems, where it is possible through parallel execution, that both CUs produce enough heat to force each other to reduce the frequency.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

With the possible limitations in mind, we analyze the parallel intra-operator execution of two operators in two different heterogeneous systems.

3 Operator Implementation and Hardware Setup

To evaluate the potential and limitations of intra-operator parallelism in heterogeneous environments, we use two operators with different characteristics in execution time, result size, and merging overhead. In detail, we choose a selection operator and a sort operator, however, our findings can be applied to other operators by anticipating possible overheads, which are presented in this work. We want to analyze parallel execution relative to its single-CU execution, so the actual operator implementation is not the focus of our work, however, we briefly present the implementation for completeness. All operators are implemented in OpenCL, enabling them to be executed on all OpenCL-supporting CUs, including most CPUs and GPUs. The operators are implemented as an operator-at-a-time approach with column oriented data format.

Our **selection operator** scans an input column of 32 bit values and produces a bitmap indicating values that satisfy the search condition. The implementation is taken from Ocelot[1] [6], an OpenCL based extension to MonetDB [3]. During execution, each thread accesses 8 values from the input column, evaluates the given search condition, and writes a one byte value to the output bitmap. Since we are working with 32 bit values, the output is 1/32 of the size of the input. Merging results of multiple runs can be done simply by aligning the results contiguously in memory, which should introduce no additional merging overhead for parallel execution.

Our **sort operator** is based on the radix sort from Merrill and Grimshaw [10]. The actual OpenCL implementation is taken from the Clogs library[2], which has been implemented and evaluated by Merry [11]. In our evaluation, we only sort keys without payload, to avoid additional transfer costs. The operator execution is more compute-intensive than the selection operator and data transfers are also more significant, since the operator is not reducing the input values, leading to the same data size for input and output. To merge two sorted results, we implement a light-weight parallel merge for two CPU threads, where one thread starts merging from the beginning and another thread starts merging from the end. Both threads only merge the result until they processed half of the resulting values. We choose this way of merging, to avoid overheads of highly parallel approaches like significantly more comparisons (Bitonic Merge [2]) or defining equally sized corresponding blocks in both sorted results [12].

For the analysis, we choose two different **heterogeneous systems**, to allow a broad evaluation: a tightly-coupled system using an AMD Accelerated Processing Unit (APU) that combines a CPU and an integrated GPU and a loosely-coupled system using an Intel CPU and Nvidia GPU. Both systems combine a CPU and a GPU, which is the most common setup for current heterogeneous

[1] https://bitbucket.org/msaecker/monetdb-opencl.
[2] http://clogs.sourceforge.net.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

Table 1. Tightly-coupled test system.

Type	Model	Frequency	Cores	Memory	Connection
CPU	AMD A10-7870K	3900 MHz	4	32 GB	host
GPU	integrated AMD Radeon R7	866 MHz	512	32 GB	host

Table 2. Loosely-coupled test system.

Type	Model	Frequency	Cores	Memory	Connection
CPU	Intel Xeon E5-2680 v3	3300 MHz	12 (24 with HT)	64 GB	host
GPU	Nvidia Tesla K80	875 MHz	2496	12 GB	PCIe3

systems. The tightly-coupled system consists of an APU combining two CUs on one die (Table 1). The GPU shares the main memory with the CPU, so it can actually access the CPU's data directly, however, for our tests we noticed that it is more beneficial to copy the data to the GPU region of the main memory before execution. This way, the GPU data can not be cached by the CPU, avoiding expensive cache snooping during GPU execution. The loosely-coupled system combines two CUs as shown in Table 2. The Tesla K80 actually has two instances of the same GPU on one GPU board, however, to isolate effects between heterogeneous CUs (CPU and GPU), we do not use the second GPU instance (Table 2 presents a single GPU instance).

4 Analysis of the Selection Operator

We begin with the analysis of the selection operator. In the following, we present the initial test results and discuss general performance issues before examining the executions on each CU separately in more detail.

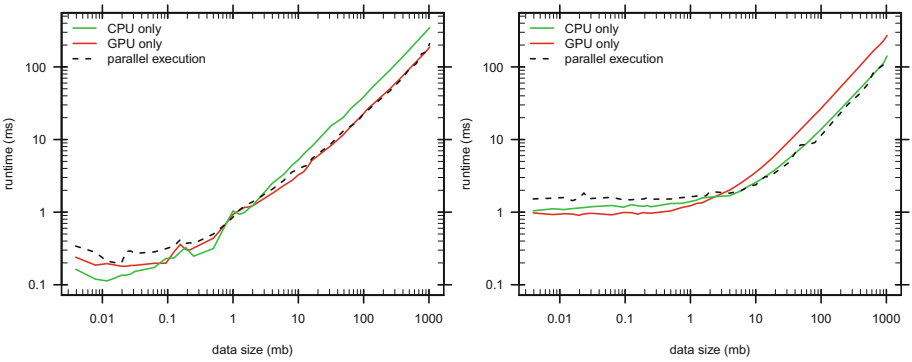
4.1 General Observations

For the initial experiment, we execute the operator on each CU with input sizes from 1024 values (4 KB) up to around 268 million values (1 GB). We capture the execution behavior and apply our calculations from Sect. 2.2 to determine the data partitioning. The calculated partitions are then used for the intra-operator execution. To see the effects of data partitioning, we force the execution to use at least a small part of data on each CU, not allowing single-CU execution, even if our calculations would suggest it.

The test results are shown in Fig. 4. Single-CU execution behavior is similar for both systems. For small data sizes, the execution time of a single CU does not differ much, because the CUs are underutilized and show a constant OpenCL initialization overhead. For larger data sizes, all CUs show linear scaling. Interestingly, for both systems the best choice CU changes between 1 and 4 MB of data.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20



(a) Tightly-coupled system. (b) Loosely-coupled system.

Fig. 4. Selection operator executed on both test systems with different data sizes.

In the tightly-coupled system, the GPU is better for large data, because of the limited computational power of the CPU. For the loosely-coupled system, the CPU is better because of the expensive data transfers to the GPU.

For both systems, the parallel version is generally not as good as expected. For small data sizes, we see the same setup as previously discussed in Fig. 3b. There is no potential for efficient parallel execution through the bad scaling of each single-CU execution. Since we force data partitioning to avoid single-CU execution, we observe at least the worst case performance of the two CUs caused by static overheads and, additionally, we see a constant overhead for data partitioning, CU synchronization, and final cleanup.

For large input data, these overheads should not be significant because of the longer execution time and the better single-CU scaling. However, we still do not achieve a significant performance improvement. In the following, executions with large data sizes are examined separately for both systems.

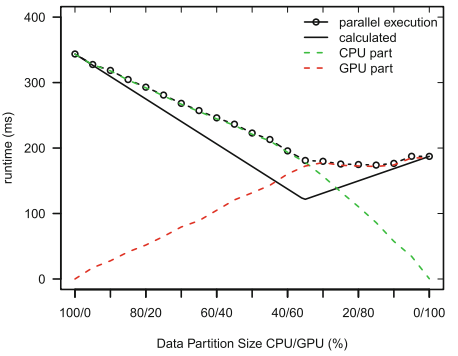
4.2 Selection Operator on the Tightly-Coupled System

For large data sizes, limitations like underutilization or missing potential do not apply, however, the parallel execution performance is worse than expected. Therefore, we choose one setting, specifically 1 GB of input data, and analyze the execution in more detail. We execute the operator with the fixed data size using different partition ratios (CPU/GPU) from 100/0 to 0/100, i.e. from 100 % of the data on the CPU to 100 % on the GPU. The result is shown in Fig. 5a. The parallel execution does not show the expected performance of our calculations and differs from the calculations especially for partition ratios where it should be beneficial.

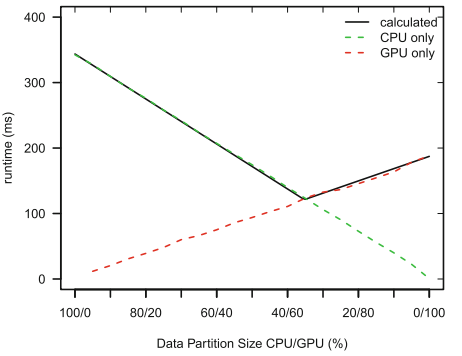
Is the Calculation Wrong? To evaluate if the problem lies in our calculations, we rerun the experiment without parallel execution. That means we use the data partitioning but execute the operators separately on each CU, not allowing

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

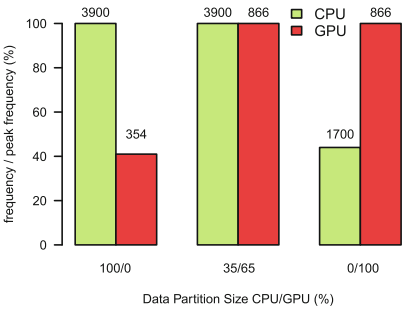
Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20



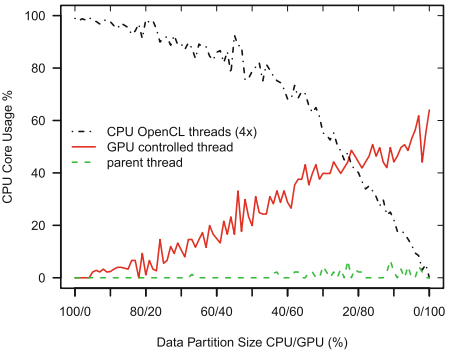
(a) Observation of parallel Execution.



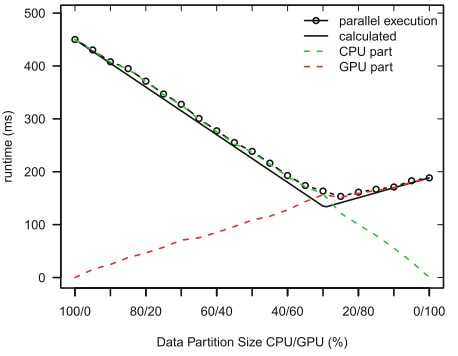
(b) Observation w/o parallel execution.



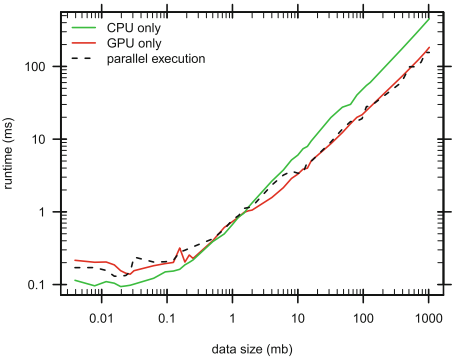
(c) CU frequency (MHz) during tests.



(d) CPU usage per thread.



(e) Execution with 3 out of 4 CPU cores.



(f) Repeating initial test with 3 cores.

Fig. 5. Extensive analysis of the parallel selection operator on the tightly-coupled system (fixed to 1 GB of data, except for (f)).

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

parallel execution. Figure 5b shows that the calculation and the actual execution are similar, confirming our calculation approach. Therefore, the performance difference has to be caused by parallel execution itself.

Is Heat a Problem? Since our first test system is a tightly-coupled system, we would expect the additional heat of parallel execution to be a problem, forcing both CUs to reduce their frequency and therefore decrease in performance. For evaluation, we rerun the three most interesting configurations multiple times while monitoring the frequencies of the CPU (using lscpu) and the GPU (using aticonfig). Figure 5c shows the result. For the CPU, the peak frequency is 3900 MHz, while it will reduce the frequency to 1700 MHz when idle. For the GPU, the peak frequency is 866 MHz and 354 Mhz when idle. The results show for each CU that peak frequencies are always used when a CU is executing the operator, not supporting our the theory of reduced frequencies caused by heat problems.

Are CU Synchronizations Interfering with Each Other? The OpenCL calls are submitted asynchronously, therefore the parent thread is not blocking for each function call, however, the parent thread has to synchronize in the end in order to wait for the execution to finish. This synchronization might interfere with execution, if multiple CUs are used. We profiled the CPU usage on thread level, for more insides. The result is shown in Fig. 5d. One thread can use up to 100 % of one core, and since the system has four CPU cores, the total core usage of all threads can not exceed 400 % (calculation similar to the Unix-tool top). The presented numbers are averages over many measuring points for each partition size, therefore, a low percentage can represent a thread running on 100 % for a short time, while being idle for the rest of the execution. In Fig. 5d, the black line represents CPU workers of OpenCL. There are four threads (one per core) with similar execution behavior, so only one line is plotted, showing the average of all 4 threads. For large data partitions on the CPU, the threads work constantly at 100 %. For small CPU partitions, the runtime is defined by GPU execution, and therefore the CPU runs at 100 % shortly, while being idle the rest of the time, hence, the smaller core usage. So far, this is as expected. Surprisingly, the parent thread has nearly no CPU usage, showing that the synchronization is not the problem because, apparently, it is implemented using suspend and resume instead of busy waiting.

In Fig. 5d, we see another thread which has not been created explicitly but, however, has a significant CPU usage. We tested the same setup with single-CU execution, noticing that this thread is only occurring when the GPU is used. We suspect this thread to be a GPU control thread, that manages the GPU queues and execution from the CPU side. With small data partitions on the GPU, this thread is only running shortly, while it has a constant 60 % core usage, when using the GPU for a longer time. This thread leads to contention on the CPU resources. The interference is not significant for the skewed execution times, e.g. for 90/10 the GPU runs only shortly, therefore the GPU thread interferes only shortly, while for 10/90 the CPU runs shortly leaving the resources to the GPU thread. However, for similar execution times of CPU and GPU, the interference

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

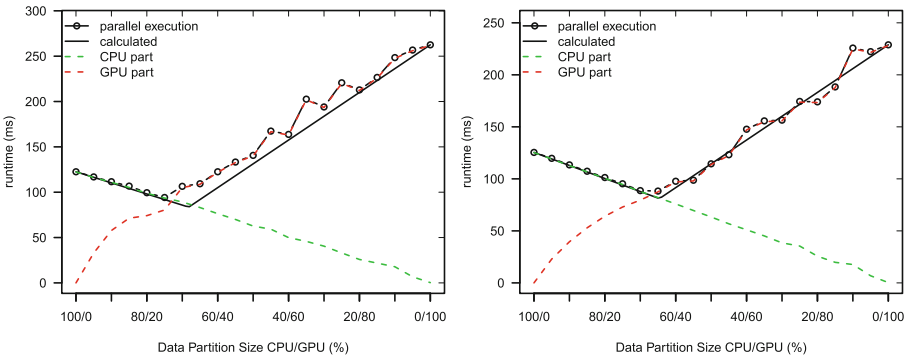
Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

is large, leading to a performance decrease of CPU *and* GPU. The CPU can not use all its resources, hence, the slow down. The GPU, has a queue consisting of input transfer, execution, and output transfer, where the queued commands are not executed on time if the GPU thread is interrupted.

How to Avoid the Interference? Since we can not avoid the GPU controlling thread, we could either accept the contention on the CPU resources and have the operating system handle the thread switching, or we could reduce the number of CPU cores used by OpenCL. This can be done with OpenCL device fission, where we reduce the number of used cores by one. Other papers also propose to leave one core idle for controlling CPU and GPU execution [7]. Figure 5e shows the execution with only three CPU cores. Here, parallel execution and calculation are similar. We can see that the CPU execution is about 25 % slower with three cores instead of four, as it is expected. However, this also influences the ideal data partition and the potential to achieve a speedup. With four cores, the calculated speedup would be 1.54x while with three cores it is only 1.41x. Adding the interference of CPU and GPU, parallel execution takes 181 ms with four cores (35/65) and 164 ms with three cores (30/70), leading only to a small difference. This effect can be seen when rerunning our initial experiment with three CPU cores in Fig. 5f, which, unfortunately, does not show a significant difference to the initial results.

4.3 Selection Operator on the Loosely-Coupled System

For the loosely-coupled system, we see different performance results as for the tightly-coupled system. When looking at 1 GB of data with different partition ratios, we see a nearly ideal performance according to our calculations (Fig. 6a). The GPU runtimes are slightly unstable because different data sizes result in a different degree of parallelism, leading to divergent GPU-internal scheduling,



(a) Observation of parallel execution.

(b) Execution with 23 out of 24 CPU cores.

Fig. 6. Selection operator executed on the loosely-coupled test system with 1 GB of data and different partitions.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

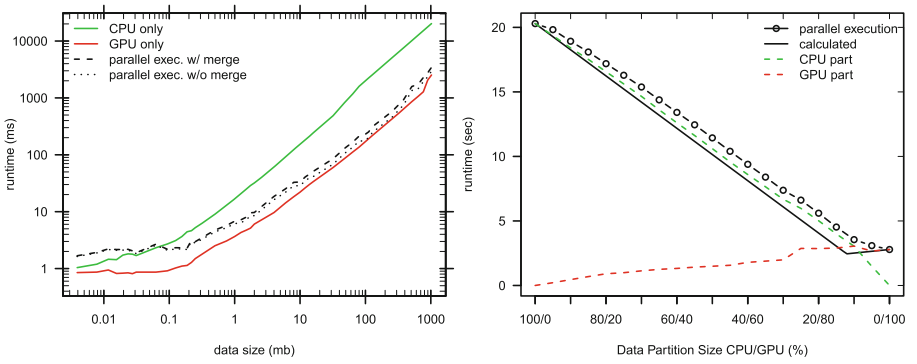
which, in this case, is more visible on the Nvidia GPU than on the AMD GPU. Additionally, the GPU runtime is slightly higher than expected. To solve this, we did the same sequence of tests as for the previous test system. Our calculations are correct according to the single-CU execution and power or heat issues are unlikely, because the system is loosely-coupled, therefore, does not share a direct power budget. When looking at the CPU utilization of each thread, we see the same effect as before: one additional thread is controlling the GPU, and therefore fighting for CPU resources. On the CPU side, there is no effect visible because one additional thread does not interfere significantly in a 24 core system (12 cores with Hyper-Threading). For the GPU, a delayed control thread leads to delays in the queuing and longer execution times. We apply the same solution as before: reducing the number of OpenCL CPU cores by one to 23 cores (Fig. 6b). This improves the GPU performance while the CPU slowdown is not significant (theoretically about 4 %). However, the GPU improvements are only marginal, leading to no substantial improvements for the overall execution.

5 Analysis of the Sort Operator

The sort operator differs from the selection operator in many ways. In general, the execution takes longer since there is more computation and multiple data accesses. Therefore computational power and data bandwidths to the CUs dedicated memories become important. On the other side, when executing in parallel, the merge step can be significant for the performance.

5.1 Sort Operator on the Tightly-Coupled Systems

Figure 7a shows the evaluation result for tightly-coupled systems. The GPU is always better than the CPU because the computational workload is more suited for the GPUs parallelism. For small data, the CUs are bound by underutilization



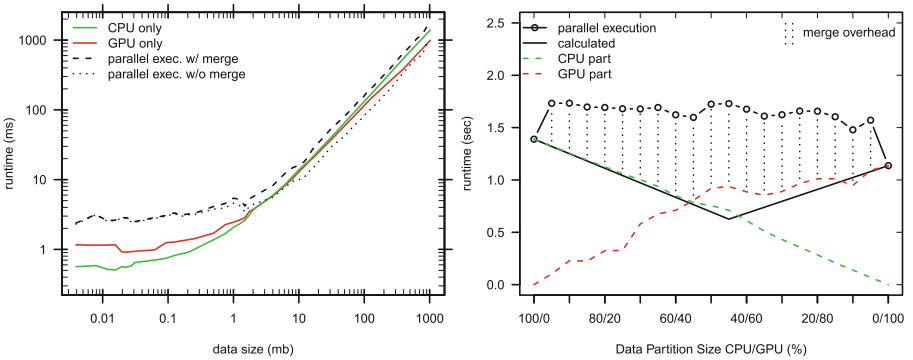
(a) Different data sizes.

(b) Different partitions for 1 GB of data.

Fig. 7. Sort operator executed on the tightly-coupled test system.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20



(a) Different data sizes.

(b) Different partitions for 1 GB of data.

Fig. 8. Sort operator executed on the loosely-coupled test system.

leading to no potential for parallel execution. For larger data, the parallel execution lies between the two single-CU executions, with the merge step seeming not significant. In a closer analysis using 1 GB of data (Fig. 7b), the reason for the parallel execution performance becomes obvious: the runtime between the CUs differs by one order of magnitude, so that parallel execution does not rectify the means of synchronization and merging. For this system, it would be best to use only the GPU, without executing the operator in parallel.

5.2 Sort Operator on the Loosely-Coupled System

For the loosely-coupled system, the results are different since both CUs seem to be equally good in executing the sort operator (Fig. 8a), which is ideal for parallel execution. However, we see a significant overhead through merging for larger data sizes. The close analysis in Fig. 8b illustrates the extent of the merge step through the dotted lines above the actual CU executions. Please note, the merge step is more significant for the overall runtime than with the tightly-coupled system because, here, the execution is faster on each CU, while the runtime of the merge is comparable for both systems. The calculated runtime bases on single-CU execution without any merging overhead. The execution on the GPU varies from the calculation, because of the additional GPU controlling thread. However, optimizing the GPU execution would lead to only minor improvements because the main difference between the single-CU parts and the actual execution is caused by the merge step. It might be possible to optimize the merge further by, e.g., adding range partitioning [1], however, the merge itself is unavoidable.

6 Lessons Learned

Concluding our analysis of two operators on two different evaluation systems, we have encountered most limitations explained in Sect. 2.3. Underutilization and

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

shared HW resources could be seen for every test. For the latter, only contention on CPU cores was noticeable and especially for the tightly-coupled system the impact was significant. Reserving one CPU core for controlling is a possible solution, however, CPU performance suffers if there is only a small amount of cores. Additionally, we have seen no potential if the single-CU differs too much or if the merge step is too large compared to the actual execution. These findings can be applied to many database operators or heterogeneous system, by quantifying the merge overhead or CU performance.

Ideally for parallel execution, we need to have (1) CUs that perform an operator equally fast, (2) one CPU core reserved for controlling, and (3) a merge step with no significant impact on the total execution time. If a merge step is needed, however, it will always be an additional overhead compared to single-CU execution. To avoid this overhead, we thought about partitioning input data once and run multiple operators in parallel on each others partial results without merging in between. While it is possible in homogeneous systems with uniform data partitions, in heterogeneous systems, each operator needs differently sized data partitions because different CUs execute an operator differently. For example, the tightly-coupled system with 1 GB of data needs a 35/65 partition for the selection and a 18/92 partition for the sort operator. Executing both operators after each other using one global partitioning would lead to a skewed execution time for CPU and GPU. It might be possible to find a partitioning for a chain of operators, so that all CUs finish this chain at the same time, however, this would only be possible if intermediate results do not need to be merged and it is unclear if a the final execution time, using suboptimal partition sizes for the single operators, is worth the effort.

All in all, we learned two major lessons from our experiments. (1) Given the limited potential and possible limitations, it is hard to achieve any speedup through intra-operator parallelism in heterogeneous environments and even for ideal cases we only achieved a speedup of 1.52x (Selection on the loosely-coupled system). It should always be considered if intra-operator parallelism is beneficial or should be avoided. (2) During our analysis, we have seen different single-CU execution behavior like different ideal CUs for the selection or always better CUs for sorting on tightly-coupled systems. If parallel execution is not beneficial, at least the placement of the execution should be considered, e.g., for the selection on the tightly-coupled system changing from CPU execution on small data sizes to GPU execution for large data sizes.

7 Conclusion

In this paper, we analyzed intra-operator parallelism for heterogeneous computing resources. We proposed an initial way to calculate good partition sizes and presented possible limitations that could hinder parallel execution. In our analysis, we used two operators with two different hardware setups and showed that especially underutilization, shared resources, different execution performance, and the merging step limit parallel execution. Therefore, it should be carefully considered if intra-operator parallelism between heterogeneous resources

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Advances in Databases and Information Systems: 20th East European Conference. Prag 2016", S. 291-305, ISBN 978-3-319-44039-2
http://dx.doi.org/10.1007/978-3-319-44039-2_20

can achieve a performance improvement, which is worth the effort, or if the resulting performance is worse and partitioning it should be avoided.

Acknowledgments. This work is funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden". Parts of the hardware were generously provided by Dresden GPU Center of Excellence.

References

1. Albutiu, M.-C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proc. VLDB Endow. **5**, 1064–1075 (2012)
2. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS 1968 (Spring), New York, USA (1968)
3. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM **51**(12), 77–85 (2008)
4. DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B., Muralikrishna, M.: GAMMA - a high performance dataflow database machine. In: Proceedings of VLDB (1986)
5. Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: ISCA, New York, USA. ACM (2011)
6. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. PVLDB **6**, 709–720 (2013)
7. Huismann, I., Stiller, J., Froehlich, J.: Two-level parallelization of a fluid mechanics algorithm exploiting hardware heterogeneity. Comput. Fluids **117**, 114–124 (2015)
8. Karnagel, T., Habich, D., Schlegel, B., Lehner, W.: Heterogeneity-aware operator placement in column-store DBMS. Datenbank-Spektrum **14**, 211–221 (2014)
9. Mayr, T., Bonnet, P., Gehrke, J., Seshadri, P.: Query processing with heterogeneous resources. Technical Report, Cornell University, March 2000
10. Merrill, D.G., Grimshaw, A.S.: Revisiting sorting for GPGPU stream architectures. In: Proceedings of PACT 2010, New York, USA. ACM (2010)
11. Merry, B.: A performance comparison of sort and scan libraries for GPUs. Parallel Process. Lett. **25**(04), 1550007 (2015)
12. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of IPDPS 2009, Washington, DC, USA. IEEE Computer Society (2009)

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden