Accelerating Scientific Algorithms in Array Databases with GPUs

Simon Marcin, André Csillaghy
Institute of 4D Technologies
University of Applied Sciences North Western Switzerland (FHNW)
5210 Windisch, Switzerland
Email: simon.marcin@fhnw.ch, andre.csillaghy@fhnw.ch

***Abstract*—In data science, interactive data analysis allows to very efficiently interpret what information is in the data. However, with increasing amounts of data available, the data crunching and data presentation becomes a more complex and resource-demanding task. Array databases allow to mitigate this difficulty by distributing the workload on a cluster. But there are two major challenges. Firstly, they rely on CPUs to process the data. Secondly, it is difficult to represent complex scientific algorithms in terms of native database operations. A way to integrate GPU accelerated algorithms is needed. In this paper, we study and test how to run scientific algorithms on GPUs entirely as native array database operators. We present the implementation of a specific scientific algorithm as exemplary use case. The computation of the Differential Emission Measure (DEM) calculates the distribution of plasma density in the solar corona at specific temperatures. DEM uses image series of the NASA spacecraft Solar Dynamic Observatory (SDO). We stack these images in a 3-dimensional array and process the data on GPUs, distributed in an array database cluster. For our algorithm, we measure a decrease of the overall runtime by a factor of 10700. We discuss different strategies used to minimize the overhead of GPUs and parameters used to scale the array database to a cluster.**

***Index Terms*—AIA; Array Database; CUDA; DEM; GPU; User Defined Operator; SciDB;**

I. Introduction

Scientific advances are driven by data inspection, with data processing being its motor. Fast data processing can change the way we progress. For instance, running a complex algorithm on a big data set in real-time can enable scientists to change parameters on the spot and gain a more intuitive understanding of the phenomena studied.

Scientific data can often be stored, managed, and visualized as multidimensional arrays - one obvious example being image series. Specialized array databases like SciDB [19] or Rasdaman [2] handle issues related to materializing the array. They store and process the data efficiently by distributing them to multiple servers which use hundreds to thousands of CPU cores. This enables scientists to process far more array data than ever before. They are not bound to the limits of their own infrastructure. The scientists can work directly on the data, hence making the process of downloading data and moving the calculations where the data resides obsolete.

Graphics processing units (GPUs) have emerged as a powerful computational platform. Various research and established projects show that the runtime of algorithms can be decreased significantly through the use of GPUs [6], [8], [14]. Array data fits especially well to the parallel processing architecture of GPUs. Therefore, GPUs are an obvious choice to accelerate scientific algorithms.

In our previous paper we showed that running scientific algorithms completely inside the array database can bring a considerable speedup [11]. Expensive data movements between the database and the (external) algorithm logic are avoided while taking advantage of the optimal data management offered by the database.

As a logical step forward, we now investigate how to implement and run GPU accelerated scientific algorithms as native operators in complement of the multi-CPU architecture of the array database. We focus on the implementation of the same algorithm used in our previous paper, which calculates the solar plasma density distribution as a function of temperature, called the Differential Emission Measure (DEM). The algorithm uses data from the NASA spacecraft Solar Dynamics Observatory (SDO). We explore different strategies to minimize the overhead of bringing GPUs into the system. In particular we investigate the use of GPU streams to efficiently transfer the data of the array database to the GPU memory.

For our DEM implementation, we get a speedup of four orders of magnitude compared to running the code on a single CPU core. In our experiment, we not only increase the scalability of the algorithm to work with much more data than was possible before, but we also decrease the runtime to such a point that users can see the effects of changed algorithmic parameters on the spot. This enables an interactive way of working with the data. Scientists can access the database in a Python session and interact with the data in a way that was not possible before. They can manipulate data in a similar fashion as in the standard Python library SunPy [20], used for solar data, as we provide a wrapper for it.

We discuss the design and implementation of the GPU integration and an analysis of the parameters used to scale up the array database to a cluster in the following sections.

II. Our work in context

The publications bellow have already addressed the acceleration and scalability of scientific algorithms in different directions.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

There are some projects which use an array database to accelerate scientific algorithms. Some only use the database as storage engine and for preprocessing steps. Authors of [12] for example run a Markov Chain Monte Carlo algorithm and use an array database in a pre-processing step. The main algorithm runs in an R program outside of the DB. The downside is: contrary to the database, the main algorithm does not run in parallel on multiple nodes. ArrayLoop [17] added an iterative component to the array database SciDB. Iterative algorithms, which can be represented with native array database queries, can run completely inside the DB. An example is the SigmaClip algorithm which moves through the array and calculates the minimum of a small sliding window until there is no change in an iteration. With this approach dependencies inside of the iterative process can be challenging to represent [13].

Accelerating array queries with GPUs is also done by [10]. The reached speedup factor is up to two orders of magnitude. Bringing GPUs into the system does not come for free, it introduces significant overheads. Our work minimizes these overheads with different strategies and can harness the processing power of both the CPU and the GPU efficiently.

The implemented GPU accelerated DEM algorithm falls into the problem scope of batched GPU problems. A GPU is not designed to solve very small problems, as not enough work is provided to hide data access behind other operations. The mapping of thousands up to millions of small problems to the GPU architecture is not obvious. There are several approaches to solve this kind of problem described in [1], [4], [21]. These approaches focus on one linear algebra operation.

The author of [3] implemented a GPU accelerated deconvolution routine based on existing GPU frameworks. Therefore, no GPU kernel code had to be written. This approach is well suited if the used algorithms, like FFT, are available in GPU frameworks. This work could be well used complementary to ours.

III. Use Case

The size of scientific data and the dimensionality of the experiments that produce them continue to grow steadily in all domains. We are about to enter the Exabyte era. The Astronomy domain is a fair generator of data. The Large Synoptic Survey Telescope (LSST), currently under construction, will produce half a Petabyte of data every month [7]. The radio telescope Square Kilometer Array (SKA), currently in design, is expected to store one Exabyte of scientific data per year [18].

The Atmospheric Imaging Assembly (AIA) on board of the NASA spacecraft Solar Dynamics Observatory is an already operating instrument. It provides full disk images of the Sun in seven different extreme ultraviolet wavelengths with a time resolution of twelve seconds. AIA creates these images with a spatial resolution of 4096 by 4096 pixels at 0.6 arcsec/pixel [9] . This results in over a Terabyte of raw data per day which is also challenging in terms of data handling.

One of the data products used to understand the physical processes of the Sun is called Differential Emission Measure (DEM). It creates temperature maps of the Sun out of multiple images with different wavelengths (e.g. AIA images). Current implementations run only on CPUs and thus need several hours to calculate a full disk DEM image. Especially if scientists want to analyze the DEM output over a longer observation time frame the calculation time gets unreasonable. This is where our work steps in. By using an array database our system can scale with the data and distribute the processing. Using GPUs to accelerate the DEM algorithm decreases the

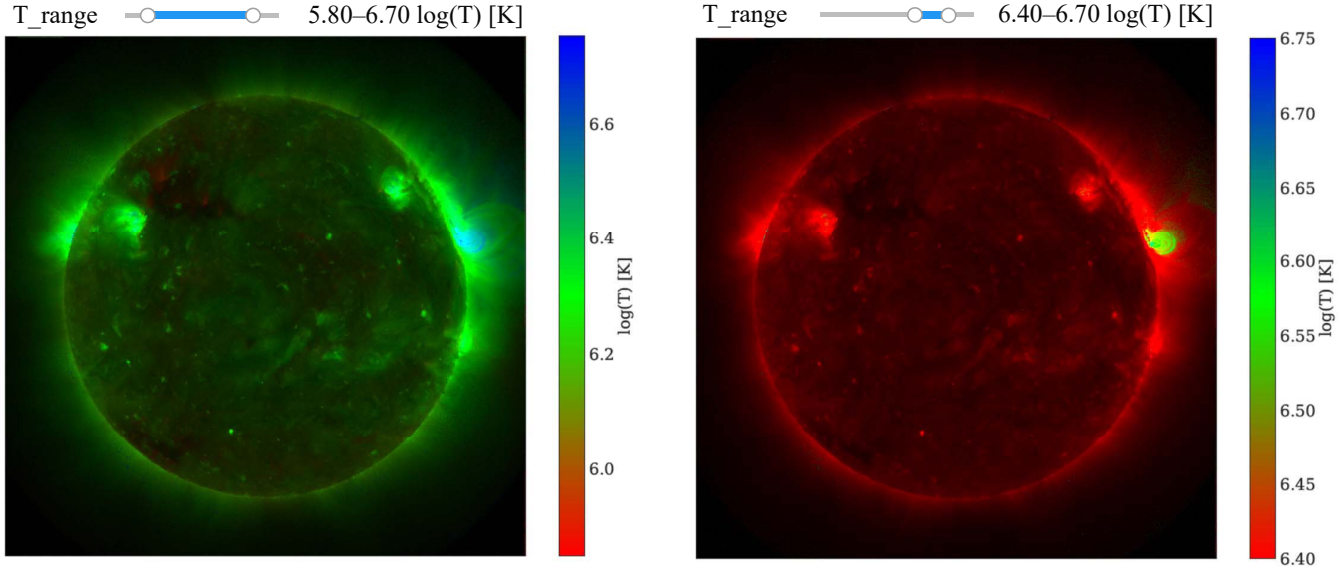

Fig. 1: The interactive DEM plot as shown in the Python interface. It allows users to change parameters (here the temperature range) on the spot. The plot shows the emission-weighted median temperature on a full disk temperature map of the Sun.

runtime drastically. This interactive way of working with big data is shown in Figure 1, where a full disk emission-weighted temperature map is calculated based on a parameter slider.

IV. System Overview

The array database SciDB splits the stored arrays into smaller sub-arrays, called chunks. The chunks are distributed to multiple SciDB instances and processed there one by one [19]. Our approach hooks into this process flow to shift work from the SciDB instance's CPU to the associated GPU. In addition, we also implement different strategies to increase the queued (pipelined) number of chunks on the GPUs. This increases the amount of parallel work that can be done.

A. SciDB Architecture

SciDB is a cluster system. The nodes of this cluster neither share memory, nor storage. This is called a shared nothing cluster architecture. By adding more nodes to the cluster, both the processing power and the storage increases. The nodes communicate by sending data via network to each other. SciDB is optimized to keep the data and processing as local as possible to minimize the data movement overhead. Each node runs multiple SciDB processes to use all available CPU cores of the system. Each process is called a SciDB instance. Normally, there are as many instances on a node as there are CPU cores. Figure 2 illustrates this architecture, as well as the mapping of multiple AIA wavelength images to a multi-dimensional array stored in SciDB.

SciDB distributes chunks over all SciDB instances with a hash-like distribution schema. A SciDB instance will always process the same chunks and store the results of these chunks locally. Therefore, an equal distribution of data to all SciDB instances is mandatory to utilize all CPU cores during an operation. As a result, the only possibility to control the distribution is by changing the size of chunks and thus the resulting total number of chunks.

SciDB uses its own query language called Array Functional Language. It is not easy for end users, in our case scientists working on solar data, to learn and use it. There are different interfaces to SciDB, e.g. Python and R, used to hide the query language. We use the Python interface SciDB-Py to communicate with the database.

B. GPU Overview

The architecture of the used NVIDIA GPUs is described in detail in other work [14], [6]. We will give a short recap of the design and key features relevant to this work. In general, these are the different running engines, and in particular, the mapping of parallel GPU programs to the GPU architecture.

The GPU, also called "device", has three different engines that run independently of each other. There are two copy engines, one to copy data from host (RAM) to device, also referred as HtoD, and another to copy data back from device to host (DtoH). The third engine is the compute engine which executes the GPU program. To reach an 100% occupancy of the compute engine, the copy engines should move the requested data in advance, otherwise the compute engine would have to wait for input.

The programming API used to run parallel programs (kernels) is called CUDA [14]. A kernel is mapped to a two-level hierarchy grid. The outer level consists of multiple thread blocks. One should run thousands to millions of thread blocks as these are assigned to the hardware for processing. A GPU consists of multiple streaming multiprocessors (SM) which process such thread blocks. The inner level is a multidimensional layout of threads. These threads are assigned to the cores of an SM. 32 consecutive threads (e.g thread 0 to 31) are executed as a warp on the hardware. The GPU works with the single instruction multiple data (SIMD) model. This means that the same arithmetic operation is executed on different data for a given warp in parallel. In the case where an algorithm executes different arithmetic operations

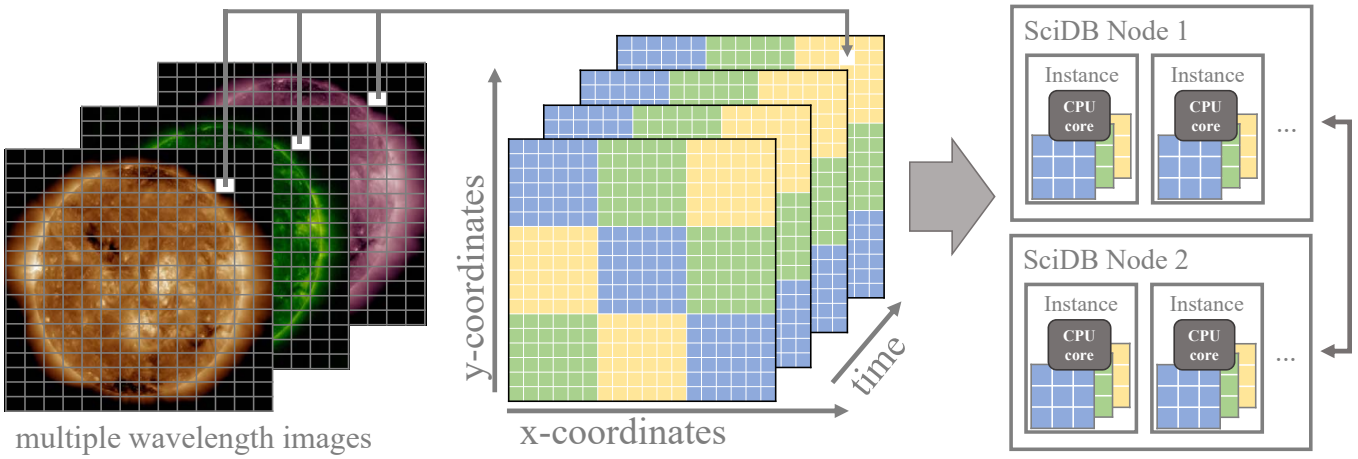

Fig. 2: Mapping of multiple AIA wavelength images to a multidimensional array. The array is split into smaller subarrays (chunks) which get distributed on multiple SciDB instances. There is a mapping function for chunks to SciDB instances, so an instance always stores and processes the same chunks.

on consecutive data (and therefore threads), the instruction unit blanks out some cores and runs the different operations sequentially. This thread divergence decreases the processing performance drastically. Therefore, GPUs reach their peak performance if the addressed problem can be expressed in the SIMD model [14].

C. GPU Integration

Introducing GPUs into the system also adds certain overhead. Data must be transferred from system memory to device memory and vice versa after the calculation is done. This overhead can reduce the performance dramatically if it is not managed carefully [10]. We use different strategies to overlay data movement with computing tasks to use all available resources in parallel.

There are two common approaches to move data from host memory to device memory: first, a regular copy which uses the CPU to transfer the data to the GPU. This approach blocks the CPU until the data is copied and is thus unavailable to perform other work. Second, an asynchronous, in respect to the CPU, copy which uses direct memory access (DMA). The copy engines of the GPU can move data independently from the CPU. However, there is a constraint related to the operating system (OS). The OS can move pages of memory to its swap partition which DMA cannot access. Therefore, the host memory area which should be copied must be page-locked (pinned). The pinning mechanism is, of course, an overhead but has to be done only once, as we can reuse it for each chunk. We use the asynchronous copy method to proceed with CPU work while the data is transferred. Especially with the chunking mechanism of SciDB, we can overlay the copy job with the next data load of a chunk.

To get the maximum acceleration out of the GPU, for any algorithm, it is essential that the compute engine of the GPU is occupied 100% of the time. To reach this goal we need to pipeline work on the GPU. Therefore, we use multiple GPU streams. A stream is a work queue on the GPU which sequentially processes GPU calls. Multiple streams are handled in parallel by the GPU. To process one chunk, we need three calls. First, copy the data from host to device (HtoD), second execute the kernel and third copy the result back (DtoH). As shown at top of Figure 3, the sequential calls cannot occupy all three engines of the GPU all the time. We use three concurrent chunks on each SciDB instance and map each chunk to a different stream. This enables us to use all three engines at the same time as shown at the bottom of Figure 3. There are no gaps between the calls in the kernel engine which leads to a high occupancy. But there can be gaps on the copy engines as a copy job is executed faster than the kernel itself. Using three chunks concurrent instead of only one on each SciDB instance, increases the memory footprint of the array database by a factor of three.

As SciDB runs multiple instances on one node, multiple processes send work to the GPU. But the GPU cannot handle this efficiently. Each process runs in an individual context on the GPU and multiple contexts cannot be processed in parallel. Hence, the GPU must perform a lot of expensive context switching. To remove this overhead we use an abstraction layer of NVIDIA called Multi Process Service (MPS). This service bundles all calls into one process for the GPU, and thus uses only one context. Figure 4 shows the difference between using the MPS layer and not using it. Of course, the MPS layer also has its downsides. It is no longer possible to use some CUDA features, like dynamic parallelism and callback functions. But using MPS is essential to get the best performance with the chunked workflow of multiple SciDB instances, as the otherwise needed context switching decreases the performance immensely. The gaps shown at the bottom of Figure 3 are filled with copy jobs of the different SciDB instances. Furthermore, the utilization of the kernel engine increases because the GPU

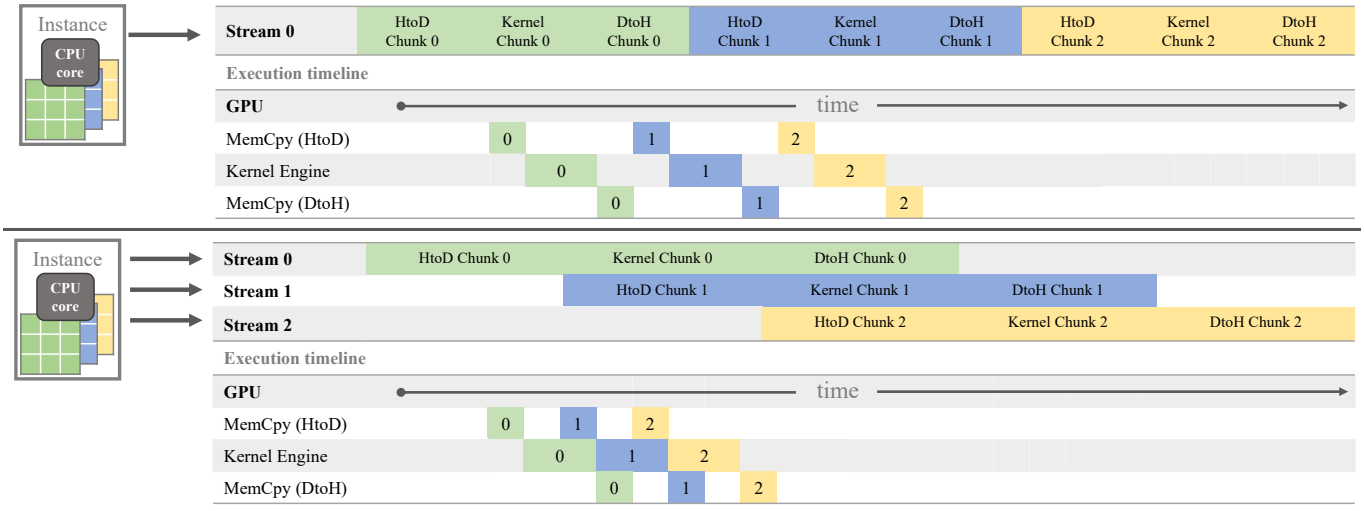

Fig. 3: Different timelines of using one (top) or multiple (bottom) GPU streams to process chunks from one SciDB instance on the GPU.

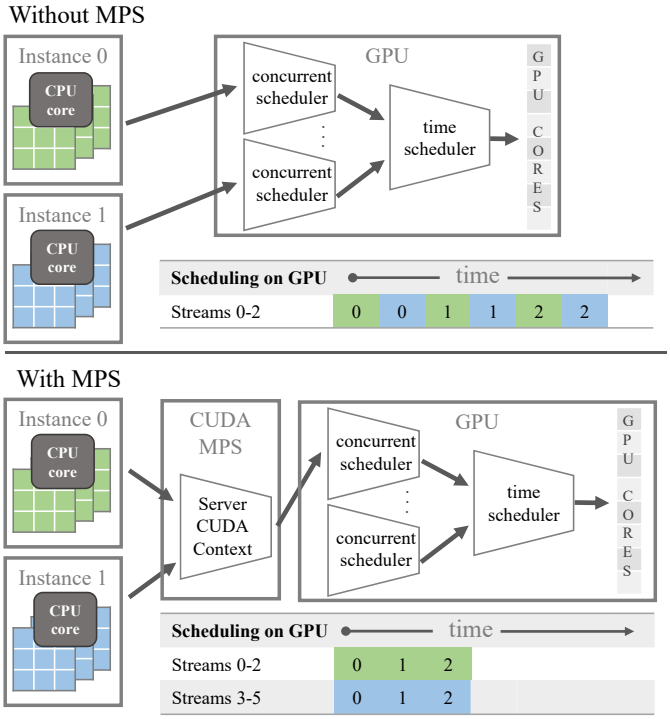

Fig. 4: Different timelines of using multiple SciDB instances communicate to the GPU with (bottom) and without (top) the MPS service.

can run multiple kernels concurrently.

D. User Interface

We use the Python interface to SciDB, SciDB-Py, which has a similar API as the NumPy package. SciDB arrays can even be downloaded as NumPy arrays to the local Python session. Nevertheless, further processing and visualization of the used solar data is still not convenient with this approach. The package SunPy provides a lot of useful visualization and processing tools for AIA data, which is a more natural way of working with this kind of data in Python. Therefore, we implemented a wrapper to combine SciDB-Py and SunPy. The wrapper tracks all changes within the AIA-array stored in SciDB. This information is needed to load the corresponding AIA metadata and recalculate some information according to the executed SciDB operations. Once all calculations are done on the SciDB cluster, the user can download the result as a SunPy map object instead of a NumPy array. This enables a convenient way to run calculations on a SciDB cluster with AIA data and do further processing and visualizations with the standard tools used in this scientific domain.

V. DEM ALGORITHM

We give a brief theoretical overview of the used DEM algorithm. The interested reader finds more information in the mentioned publications. Furthermore, we describe the adaptions made to run the DEM algorithm efficiently on GPUs and how we validate the results of the GPU implementation.

A. Algorithm

The DEM, $\varepsilon(T)$, characterizes coronal emission at a given temperature, T, and is defined as densities, n, along the line of sight. It is based on observed intensities, g_n, as by the temperature-dependent emissions of the different spectral lines of AIA images. The observed intensities are given by the instruments response function, $R_n(T)$ integrated against the DEM, as shown in equation (1) [5], [16].

$$g_n = \int R_n(T) \cdot \varepsilon(T)\ dT \tag{1}$$

The temperature response functions of AIA are broad, therefore the DEM calculation has to minimize the error between the measured data and the calculated emissions. The error is defined as χ^2 error, which is a standard technique in inverse problems [5]. We use the algorithm proposed by [16] which consists of a simple regularized inversion followed by an iterative algorithm used to remove negative emission measures. The constraint that the DEM can be expressed as linear combination of narrow temperature bins, $B_j(T)$, with coefficients e_j, where $\varepsilon(T) = \sum_j e_j \cdot B_j(T)$, the integral equation (1) becomes the matrix equation (2)

$$g_n = \sum_j \left[e_j \int R_n(T) \cdot B_j(T)\ dT \right] \tag{2}$$

The outputs of the used DEM algorithm are the coefficients e_j which can be multiplied by the temperature bins $B_j(T)$ to get the median temperature.

B. Adaption for GPU processing

To use the GPU resources efficiently we set the number of coefficients e_j to 32. This allows to use thread blocks of the size 32 by 7 (number of AIA wavelengths) for each pixel. The thread block is perfectly aligned with the warp size of 32. Therefore, memory access on device memory and GPU shared memory (a fast 49KB scratchpad memory on the level of a SM) can be accelerated by a perfectly aligned warp access (coalesced). Our implementation heavily reuses the shared memory and thread registers for intermediate results. The order of some calculation steps is changed within the algorithm to reuse intermediate results or to hide data loading behind other operations. There are no other adaptions made to the algorithm and therefore the results can be compared with the output of the origin code. Algorithm 1 shows the coarse structure of the calculation with the most occurring loops and most compute intense calls.

It is obvious that we have to accelerate the longest running calculations inside of the loops which are executed over and over again. Therefore, we focused on the matrix-vector multiplications (gemv function) on line 5 to 10 in Algorithm 1 and the matrix inverse call on line 4. We implemented different techniques like warp reductions for the matrix multiplications, unrolling small loops and atomic shared memory operations to accelerate these crucial operations. Line 1 is abstracted by

Algorithm 1 Pseudocode of the most intense calculation steps of the DEM algorithm as well as the coarse structure of the algorithm.

Input: g_n (7 AIA wavelength images)
Output: e_j (32 coefficients per pixel)
1: **for** each *pixel* **do**
2: **for** each χ^2*_minimum* **do**

regularize input_data
3: **while** $\chi^2 > \chi^2$*_min* **do**
4: inverse(regularized_data[7, 7])
5: gemv(inverse_mat[7,7], data_vector[7])

calculate DEM to check χ^2 error
6: e_j = gemv(instrument_basis[32,7], reg_data[7])
7: χ^2 = gemv(intensities_basis[7,32], e_j[32])

iterative removal of negative emission measure
8: **while** $\chi^2 > \chi^2$*_min* **do**
9: e_j = gemv(instrument_basis[32, 7], reg_data[7])
10: χ^2 = gemv(intensities_basis[7,32], e_j[32])

the parallel kernel launch and the parallel execution of thread blocks on the GPU.

C. Batched GPU problem

A batched problem is the calculation of a large number (e.g. thousands to millions) of small problems. This problem class is widely common in computer vision, anomaly detection or, among others, in magnetic resonance imaging (MRI) where billions of 32 by 32 eigenvalue problems need to be solved [9]. Small problems normally do not provide enough work to completely saturate the GPU and thus run faster on a CPU. The challenge of running batched problems in general and batched linear algebra routines in specific is studied by multiple groups [1], [4], [21].

All proposed approaches use the large amount of problems which can be solved independent in parallel. There are mainly three distinctive designs to run the GPU kernel. The difference is in the mapping of a single problem instance (in our case one pixel of the AIA image with multiple wavelengths) to the CUDA architecture. First, each single thread is responsible for one problem. This strategy is used by [1] among others and shows the best performance if the data fits into thread registers. An algorithm using this approach cannot have any thread divergence over all independent problems as it would decrease the performance drastically. Second, a single problem instance is mapped to one warp. Each problem is solved in parallel by 32 threads [21]. As the problem fits perfectly into one warp, there is no synchronization inside the code needed as it is done by the GPU automatically. This enables divergence in code and computation among the individual problems without increasing the runtime. The downside is the small number of threads (32) which will work concurrent on a single problem. Third, a problem is assigned to a complete thread block [4]. This enables to choose the number of threads which work concurrently on a single problem instance. We use this approach as it has the fastest runtime and was easiest to program among all. The first approach is not feasible at all because of the thread divergence of the DEM algorithm in its iterative steps. Table I shows the evaluation of the three approaches for the DEM algorithm.

Table I: Properties of three different approaches for batched GPU problems in the context of the used DEM algorithm.

Properties	1 Pixel / Thread	1 Pixel / Warp	1 Pixel / TB
Max concurrent pixels per SM	2048	64	10
Synchronization calls	No	No	Yes
Prone to thread divergence	Yes	No	No
Calls for biggest matrix multiplication	224	7	1
Full disk DEM runtime (GTX 1080Ti)	-	65s	32s

D. Validation

Running double precision calculations on GPUs can result in different output for each run. This is because arithmetic operations on doubles are not associative. The order of warps which are executed cannot be controlled and this leads to different sequences of arithmetic operations. We compared our results with the results of the origin code written in IDL with a precision of 10^{-15}. Furthermore, we validated if we reach the desired χ^2 error in order of unity for the 95th percentiles as done by the authors of the origin algorithm [16].

VI. Benchmarks

We test different SciDB database parameters such as chunk size or number of instances per node, and discuss their impact on the runtime. Furthermore, we analyze the usage of the GPU. All benchmarks are based on a full disk (4096 by 4096 pixel) DEM calculation with 7 input attributes (different wavelengths). First, we run the calculation on a PC with an Intel i7-7700 3.6GHz CPU and a Nvidia GTX 1080Ti GPU with 3584 cores distributed on 28 SMs. This gives us insights on how fast the GPU accelerated algorithm runs locally on a PC with the latest generation of GPUs. Second, we extend the benchmarks to a SciDB cluster in the Amazon cloud. There, we use instances which run Intel Xeon E5-2686 v4 CPUs with 32 cores and eight NVIDIA K80 GPUs with 5760 cores distributed on 30 SMs, called p2.8xlarge instance. The results are compared with our baseline of 4 hours for a full disk DEM calculation on the mentioned PC running a C++ implementation of the algorithm on one CPU core.

As the data loading, storing, algorithm processing and cluster management part run asynchronous, we cannot split the runtime into the distinct categories. We only measure the SciDB overhead which is defined as the work of SciDB while no asynchronous calculation is running at the same time.

A. Runtime on a PC

We run our implementation on a PC to check the behavior on commodity hardware. This way we learn about the runtime

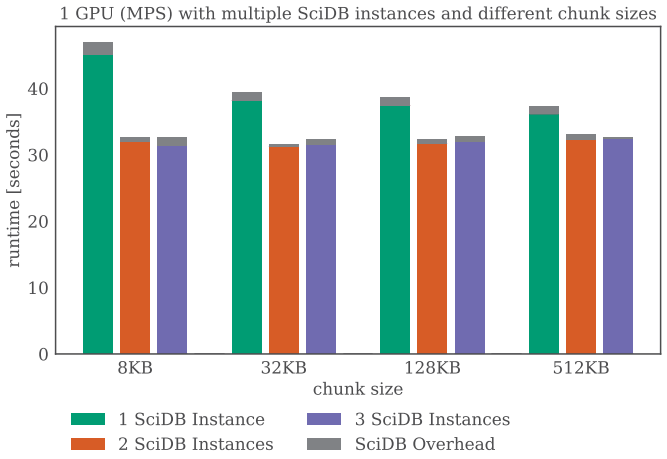

Fig. 5: Runtime (y-axis) of a full disk DEM calculation on a GTX 1080Ti GPU with multiple SciDB instances and different chunk sizes (x-axis).

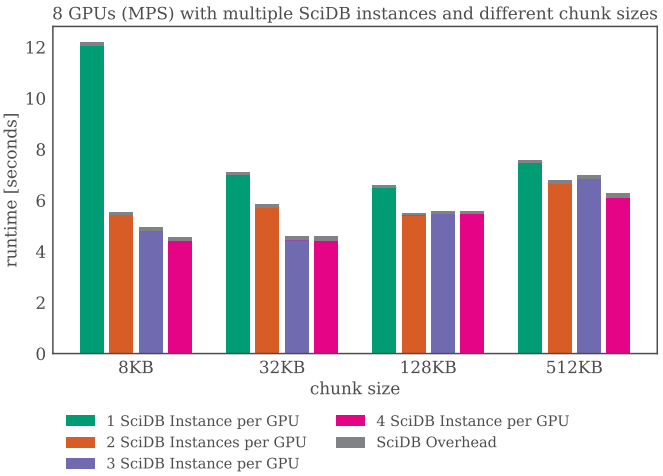

Fig. 6: Runtime (y-axis) of a full disk DEM calculation on 8 Tesla K80 GPUs with multiple SciDB instances for each GPU and different chunk sizes (x-axis).

that can be achieved on a standalone PC used by a scientist. Figure 5 shows the runtime of a full disk GPU accelerated DEM calculation for different chunk sizes and the usage of one to three SciDB instances. It shows that the GPU is fully occupied with 2 instances and pipelining more work to it by using more SciDB instances does not decrease the runtime. This in turn means that six out of eight CPU cores of this PC are not used. Further performance gain can be achieved by using multiple GPUs. A runtime of 32 seconds is the best we can achieve with the used hardware. This is a speedup of 450 times compared to our baseline.

B. Runtime on a GPU Server

Figure 6 shows the runtime of a full disk DEM calculation on a p2.8xlarge Amazon instance. There is again the chunk size on the x-axis for one to four SciDB instances. As in our implementation a SciDB instance always uses the same GPU, this corresponds to one to four CPU cores assigning work to a single GPU card. Compared to the PC run, the GPU is not fully occupied with only two SciDB instances assigning work for small chunk sizes. This is because the used GPU has more SMs and is faster in double precision calculations. Using a chunk size of 128 KB or 512 KB increases the runtime, as loading and storing data chunks takes too long to put work to the GPU without gaps in the compute engine. We achieve the best performance with 3 SciDB instances per GPU and 32KB chunks, which means the GPU is the bottleneck as we still have unused CPU cores. The SciDB overhead can be neglected if we use only one node.

C. Runtime on a GPU Cluster

Only by scaling to a cluster we really profit of the underlying array database. SciDB handles the complete management of the cluster and is responsible for the scaling of the system. Using small chunks increase the SciDB overhead as each chunk has some metadata to generate and some initialization effort. Figure 7 shows the runtime of a multi-node GPU cluster. We reach the best performance of 1.34 seconds with a chunk size of 32KB and the maximal number of nodes.

D. Bring it all together

The scaling is almost linear, the small difference is mainly due to two reasons. First, there is some cluster and array database overhead. Second, the work is unevenly distributed as some chunks contain input data which is harder to solve than other (like the pixels outside of the sun compared to pixels in an active region). In our previous paper the latter fact caused sublinear scaling on a SciDB CPU cluster [9]. We get much better performance and scaling on our GPU implementation as each pixel gets mapped to a thread block

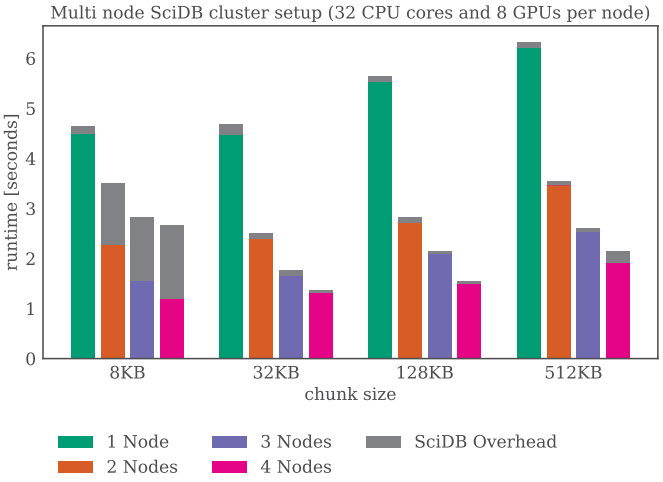

Fig. 7: Runtime (y-axis) of a full disk DEM calculation on a SciDB cluster with up to four nodes running each 8 Tesla K80 GPUs for different chunk sizes (x-axis).

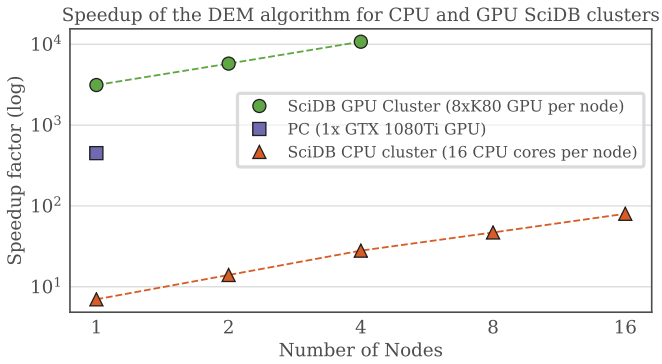

Fig. 8: Speedup of a full disk DEM calculation of different implementations and used computing nodes compared to the baseline of a C++ implementation using 1 CPU core. Triangle markers are from our previous paper on a CPU only SciDB cluster [9]. The square marker is the speedup of a PC with one GPU and the round markers are the GPU cluster results.

and therefore individually assigned to the GPU SMs. Long running calculations of difficult input pixels do not occupy the complete hardware anymore. Figure 8 compares the different speedup factors of the CPU only implementation with the GPU implementation of this paper. Even a standalone PC with a GPU outperforms the runtime of a 250 CPU cores SciDB cluster. We reach a speedup factor of over 10700 on our 4 node GPU SciDB cluster compared to our baseline. As the measured runtime contains loading, calculating and storing the data, it can be linearly extrapolated to a time series of AIA images.

Conclusion

We extended the array database SciDB with a scientific algorithm as custom database operator and accelerated it with GPUs [1]. The chunked workflow of an array database works well with GPUs if we do some adaptations. We adapt to allow concurrent asynchronous copying and computing operations on the GPU and CPU. Our implementation can interleave the work of the GPU with loading and storing of chunks done by the CPU. Therefore, we can utilize 100% of the computing engine of the GPU. However, we increase the memory footprint of SciDB by a factor of three in order to work concurrently on multiple chunks on each SciDB instance. The Python package normally used to work with the referred solar data is SunPy. By combining SunPy with SciDB, scientists get a convenient way of running GPU accelerated calculations on an array database cluster. This enables scientists to perform their research on a far bigger data set than before due to the scalability of SciDB. The scientists can run the DEM algorithm four orders of magnitude faster now. As the runtime of a full disk DEM calculation is between one and two seconds, scientists can change parameters of the algorithm on the spot as they do not have to wait hours for a result. Therefore, they can work interactively with the data and the DEM algorithm.

A GPU implementation of a scientific algorithm requires extra work which needs to be carried out for each algorithm. Nevertheless this is a valuable investment as our architecture separates the GPU code from SciDB specific implementations. This allows us to reuse the GPU implementation on other infrastructures or to easily integrate already existing GPU code. Furthermore, the implemented algorithm can be combined with existing database operators. With a GPU implementation, scientists lose the ability to change the internals of the algorithm, as it no longer runs locally on their machines and changing it is no longer as easy.

Using the proposed architecture for the whole SDO mission would not be feasible for financial reasons, as all nodes would have to be equipped with GPUs and the storage requirements of the mission (over two Petabytes) would dictate a large number of nodes. A new SciDB feature called Elastic Resource Groups [15] can divide storage and computation nodes which would alleviate this problem but would add some data movement overhead. Our proposed architecture can be used well in projects which analyze data from a limited time frame. A GPU accelerated DEM service could be implemented near the data archive which calculates the DEM output ad-hoc for the requested data. Scientists without access to a GPU cluster could profit from such a service. Therefore we will also publish a non SciDB version of the GPU accelerated DEM algorithm.

Further improvements to the GPU implementation of the algorithm and to the SciDB cluster could decrease the runtime again by an order of magnitude. The reported speedups are possible especially if an algorithm is designed to run on GPUs from the beginning and is not just a conversion of a CPU version. Furthermore, we have unused CPU cores in our architecture which should be better harnessed in the algorithm calculation complementary to the GPUs. To fully include GPUs to SciDB fundamental changes in the SciDB core are required. Two different GPU accelerated operators cannot be efficiently concatenated at the moment. Intermediate results would be copied from the GPU memory to system memory after each operation. Removing this hurdle would be an important step as a lot of already existing GPU code could be integrated. This would simplify the first steps for a broader use of GPUs in diverse research domains. Advanced GPU acceleration strategies which already exist for regular databases could be included, like transferring data directly between SSD drives and GPUs [22]. Such improvements could lead to real-time implementations.

Accelerating scientific algorithms in array databases with the processing power of GPUs can change the way scientists work. Bringing processing power closer to the data and harnessing it efficiently enables a more explorative and interactive way of handling big data. Calculations which needed careful planning of resources can be done ad hoc with the proposed architecture.

[1]Code available on GitHub: https://github.com/simonmarcin/SciDB_GPUs

REFERENCES

[1] M. J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 2–13. IEEE, 2012.

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Acm Sigmod Record*, volume 27, pages 575–577. ACM, 1998.

[3] M. C. Cheung. Gpu-accelerated imaging processing for nasa's solar dynamics observatory. *Presentation, GPU Technology Conference.*, 2015.

[4] T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched cholesky factorization on a gpu. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 432–440. IEEE, 2014.

[5] I. G. Hannah and E. P. Kontar. Differential emission measures from the regularized inversion of hinode and sdo data. *Astronomy & Astrophysics*, 539:A146, 2012.

[6] M. Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[7] J. Kantor, T. Axelrod, J. Becla, K. Cook, S. Nikolaev, J. Gray, R. Plante, M. Nieto-Santisteban, A. Szalay, and A. Thakar. Designing for petascale in the lsst database. In *Astronomical Data Analysis Software and Systems XVI*, volume 376, page 3, 2007.

[8] M. Korytkowski, P. Staszewski, P. Woldan, and R. Scherer. Fast computing framework for convolutional neural networks. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*, pages 118–123. IEEE, 2016.

[9] J. R. Lemen, D. J. Akin, P. F. Boerner, C. Chou, J. F. Drake, D. W. Duncan, C. G. Edwards, F. M. Friedlaender, G. F. Heyman, N. E. Hurlburt, et al. The atmospheric imaging assembly (aia) on the solar dynamics observatory (sdo). In *The Solar Dynamics Observatory*, pages 17–40. Springer, 2011.

[10] F. Liu, K. Lee, I. Roy, V. Talwar, S. Chen, J. Chang, and P. Ranganathan. Gpu accelerated array queries: The good, the bad, and the promising. *HP Laboratories, Palo Alto, CA, USA, Tech. Rep. HPL-2014-50*, 2014.

[11] S. Marcin and A. Csillaghy. Running scientific algorithms as array database operators: Bringing the processing power to the data. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 3187–3193. IEEE, 2016.

[12] D. S. Matusevich, W. Cabrera, and C. Ordonez. Accelerating a gibbs sampler for variable selection on genomics data with summarization and variable pre-selection combining an array dbms and r. *Machine Learning*, 102(3):483–504, 2016.

[13] M. Moyers, E. Soroush, S. C. Wallace, S. Krughoff, J. Vanderplas, M. Balazinska, and A. Connolly. A demonstration of iterative parallel array processing in support of telescope image analysis. *Proceedings of the VLDB Endowment*, 6(12):1322–1325, 2013.

[14] NVIDIA. Cuda c programming guide. *Docs.nvidia.com*, 2017.

[15] Paradigm4. elastic resource groups. *GitHub*, 2017.

[16] J. Plowman, C. Kankelborg, and P. Martens. Fast differential emission measure inversion of solar coronal data. *The Astrophysical Journal*, 771(1):2, 2013.

[17] E. Soroush, M. Balazinska, and D. Wang. Arraystore: a storage manager for complex parallel array processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 253–264. ACM, 2011.

[18] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[19] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[20] SunPyCommunity. Sunpy guide. *http://docs.sunpy.org*, 2017.

[21] H. Tokura, T. Honda, Y. Ito, K. Nakano, M. Nishino, Y. Hirota, and M. Saeki. Gpu-accelerated bulk computation of the eigenvalue problem for many small real non-symmetric matrices. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 490–496. IEEE, 2016.

[22] K. Zhang, F. Chen, X. Ding, Y. Huai, R. Lee, T. Luo, K. Wang, Y. Yuan, and X. Zhang. Hetero-db: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.