Adaptive GPU Array Layout Auto-Tuning

Nicolas Weber
TU Darmstadt
Graduate School of Computational Engineering

Michael Goesele
TU Darmstadt
Graduate School of Computational Engineering

ABSTRACT

Optimal performance is an important goal in compute intensive applications. For GPU applications, this requires a lot of experience and knowledge about the algorithms and the underlying hardware, making them an ideal target for auto-tuning approaches. We present an auto-tuner which optimizes array layouts in CUDA applications. Depending on the data and program parameters, kernels can have varying optimal configurations. We thus adjust array layouts adaptively at runtime and achieve or even exceed performance of hand optimized code. We automatically detect data characteristics to identify different performance scenarios without user input or additional programming. We perform an empirical analysis of the application in order to construct our decision models. Our adaptive optimization requires in principle profiling data for an extremely high number of scenarios which cannot be exhaustively evaluated for complex applications. We solve this by extending a previously published method that is able to efficiently profile single kernel calls and enhance it to find application-wide optimal solutions. Our method is able to optimize applications in a few minutes, reaching speed ups of up to 20% compared to hand optimized code.

CCS Concepts

•**Software and its engineering** → *Massively parallel systems; Software performance;*

Keywords

GPU, Adaptive Auto-Tuning, Array Layouts

1. INTRODUCTION

In recent years, *Graphics Processing Units* (**GPU**) have become very popular in *High Performance Computing* (**HPC**) applications. Many HPC users are experts in their application domains such as chemistry, physics, mechanical or electrical engineering, but do not have much experience in hardware architectures. Systems that assist them in optimizing their applications to save time and money are therefore very important. But even expert computer scientists are often unable to find an optimal configuration or are forced to re-optimize their code for each and every new hardware architecture. This has brought forth a wide variety of different auto-tuners, targeting various kinds of performance problems [10, 16] or directly addressing application specific optimizations in various domains [14, 17].

We propose a new adaptive runtime optimization enabled by the MATOG auto-tuner [16]. MATOG optimizes array layouts in CUDA applications for arbitrary application domains. Our adaptive optimization can react to changing data characteristics and selects optimal configurations at runtime. It bases its decisions on automatically gathered meta data and requires no additional programming. We furthermore propose an application analysis method that extends the MATOG profiler. MATOG can so far predict per-kernel optimal configurations from a small number of benchmark results. Since per-kernel optimal solutions are not sufficient for our adaptive runtime optimization, we encapsulated the approach with a new method that finds application-wide optimal solutions. The new algorithm requires only a couple of minutes to analyze even complex applications while achieving performance comparable with or better than hand optimized code. Our contributions are as follows:

- An adaptive auto-tuning approach for array access in GPUs based on automatically gathered meta data,
- a very fast application analysis method that finds near optimal solutions and
- an evaluation of our system on a wide range of GPU applications from different application domains.

2. RELATED WORK

2.1 GPU Memory Access Auto-Tuning

Memory access is one of the most important performance aspects in many GPU applications and can be improved in various ways. Li et al. [9] optimize the usage of caches without changing memory access or array layouts. They force a specific number of the threads inside a thread group to use the cache, while the others bypass it. This relieves the cache and allows higher performance in certain applications. Park et al. [12] have presented an alternative scheduling method to improve the memory level parallelism in GPUs. Other approaches such as the MAPS framework [3] require adapting the application code. MAPS enables users to trans-

SEM4HPC'16, May 31 2016, Kyoto, Japan

ISBN 978-1-4503-4351-0/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2916026.2916031

parently write multi-GPU applications using STL-like data structures. It hides the entire memory management from the user and distributes the workload automatically to multiple devices. Koefler et al. [8] go one step further by improving the usage of *Array of Structs* (**AoS**) data structures in GPU kernels. They support storing these as AoS, *Structure of Arrays* (**SoA**) or as *Array of Structure of Arrays* (**AoSoA**) with variable tile sizes. We have presented the MATOG auto-tuner [16], which follows a similar approach but does not limit the optimizations to AoS data structures. It optimizes multi-dimensional arrays in addition to AoS, adjusts cache sizes and the usage of GPU specific memories for individual kernels. In this paper we do not add new optimization functionality. Instead, we improve the way they are applied by adding more flexibility through adaptive runtime adjustments according to the data characteristics.

2.2 GPU Application Analysis

One important part of auto-tuning is the method used to find optimal configurations. Koefler et al. [8] use an undirected graph to model memory access inside a kernel. This graph is then combined with a generic GPU model to determine the best working configuration. Such a static analysis can be performed relatively fast, but fails if the theoretical GPU model does not represent the architecture well enough. This could be especially problematic for future GPU architectures that may not conform to current models. Further, it does not consider any input data dependent effects that can have a significant performance impact. Other auto-tuners follow a different path and use empirical profiling. They execute the actual code on the target architecture with real data. This method does not suffer from the problems mentioned above, but is obviously much more resource intensive. Many different approaches have been used over time to handle this issue, e.g., greedy algorithms [10], genetic algorithms [1] or downhill-simplex based methods [6]. We proposed a prediction guided profiling method [16] specifically designed for optimizing array layouts on GPUs which significantly reduces the execution time. It profiles a specific predetermined set of configurations that is sufficient to estimate the performance of all non-profiled configurations. This method finds, however, only optimal solutions for single kernel calls and not for an entire application run. In this paper we add an additional analysis step, which uses profiling data in conjunction with a graph representing the array usage inside the application to analyze the profiled application runs and to determine the best layouts. We explicitly use the prediction capability of this method to estimate the performance of non-profiled configurations.

2.3 Adaptive Auto-Tuning

Optimal performance depends often on the actual workload, input data and parameters. Shen et al. [13] proposed an auto-tuner that profiles different input data parameters and tries to automatically identify occurring patterns. With this pattern recognition they feed a version selection wrapper to determine the optimal implementation prior to the application start. This does, however, not actively adapt the implementation during runtime, which can often be necessary. Some auto-tuners [6, 11] try to tackle this by actively reacting to changing effects. Both approaches base their decisions on user defined analysis methods, i.e., their success depends on the ability of the programmer to identify characteristic effects. The required data analysis can furthermore introduce an overhead depending on the data analysis actually performed. We combine the advantages of both solutions and automatically determine possible patterns without any user interaction. Based on these, we actively adapt array layouts to changing data properties during runtime.

Machine learning techniques are widely used in auto-tuning and provide an easy way for automatic decision making. Prominent examples are regression trees [4, 13] or *Support Vector Machines* (**SVM**) [11], which are used in conjunction with meta data and other characteristics to decide on optimal configurations. We use SVMs in our adaptive auto-tuning as an oracle for predicting optimal configurations.

3. OVERVIEW

Our adaptive array layout optimization is based on the MATOG auto-tuner [16], which we improve and extend in several ways: Whenever an array is allocated we decide on the optimal array layout, based on automatically gathered meta data such as the array sizes and kernel launch configurations. Further, on every kernel call we decide which configurations are optimal for the given data characteristics.

To establish our decision models we need a series of processing steps. First of all, we analyze the application with the MATOG profiling method [16]. We then use the resulting data to construct a specialized dependency graph that helps to find application wide optimal solutions. In the end, we use these solutions and meta data that was recorded during the profiling to establish our decision models. These are then used during runtime to predict the optimal configurations.

4. MATOG AUTO-TUNER

Optimizing memory layouts is a tedious task as the number of possible optimizations is extremely high. It is also a very time consuming process as nearly the entire code has to be adjusted to test whether one or the other layout performs better. If done manually, this can also be very error prone, if only a single memory access is missed. To solve this, we presented the MATOG auto-tuner that provides a multi-dimensional and/or AoS-like memory access (e.g., *array[x][y][z].field*) interface to abstract the underlying memory access from the user. MATOG integrates seamlessly into existing CUDA applications by interfacing with CUDA Driver API calls. It does not require any custom compiler but generates code for C++ data structures and uses a library to apply the necessary optimizations. This concept allows to intercept all CUDA calls without a special syntax and with minimal computational overhead. Further the user does not need to change anything in his compiler tool chain which makes the approach very portable and introduces hardly any limitations for the programmer. MATOG does not interfere with the usage of other CUDA libraries but does not automatically optimize their data structures if they do not use MATOG.

With this memory access interface, MATOG is able to optimize the memory access for multi-dimensional arrays by transposing the indexing, to store arrays with struct types as AoS, SoA or AoSoA, to adjust the L1 cache size on Kepler GPUs, placing arrays either in global or texture memory and to choose optimal configurations for user defined preprocessor optimizations inside the kernel code.

MATOG also allows to dynamically assign arrays with two different properties: First, arrays can be flagged as *read-only* which means that data from an array is only read. This implies that it is suitable to be stored in texture memory. Second, it can be marked as *write-only*, which means that the data will be entirely overwritten by the next kernel call. These properties enable additional optimizations or can be used to improve the profiling process.

4.1 Application Analysis

Like many other auto-tuners [6, 11, 13], MATOG relies on empirical profiling. It uses in-application profiling which only restarts kernels instead of restarting the entire application for every configuration that has to be profiled. I.e., there is no need to constantly repeat costly setup or I/O, which can be a quite significant effort, especially in HPC applications. MATOG automatically takes care of resetting the arrays prior to each configuration profiling as well as providing the data in the correct format to the GPU. For this, it stores copies of the kernel input data in the host system's memory, as well as converted versions, if necessary. The data conversion and the compilation of the kernels in the various configurations is performed on the CPU in parallel to the actual profiling on the GPU.

Due to the high complexity of some applications, it is necessary to perform a partial profiling, as an exhaustive search can require several days or more of computation. For this MATOG uses a prediction-guided profiling technique. This method profiles only a very limited set of configurations per kernel and then predicts the location of the optimum in the solution space to focus its search on promising configurations.

4.2 Prediction Algorithm

Our previously published prediction algorithm [16] can be used to find optimal configurations for single kernels. It is based on the assumption that configurations are independent of each other and the time difference ($\Delta(A, B)$) between configurations can be used to estimate the performance of others using a linear model. As this assumption does not hold for all optimization dimensions, we introduced two sets. The first contains dimensions which are *independent* (D_I) and do not have an influence on others. The second set is called *shared* (D_S) and contains all dimensions that influence others. By default, MATOG only uses the size of the L1 cache as a shared dimension but allows the user to add additional shared dimensions.

These two sets are used to determine the configurations, that need to be profiled in order to predict the performance of all others. Each permutation of values of the shared dimensions represents a *domain* (see Fig. 1). Only inside a domain, the linearity constraint is valid, allowing us to estimate the performance using a linear model. Cross-domain predictions are, however, not possible since the linearity assumption does not hold. Each domain has a so called *base configuration* (C_B) which serves as support point for the prediction. To predict the performance of any configuration in the domain, a so called *support configuration* (C_S) is required for each independent dimension. This support configuration is equal to the base configuration except for the value of the single independent dimension that it represents. For this dimension it matches the value of the *predicted configuration* (C_P). Fig. 1 shows an example of how the base

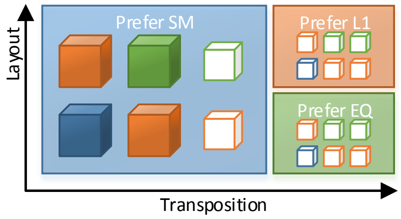


Figure 1: Example solution space for a kernel with three optimization dimensions (L1 cache size, Layout and Transposition). As L1 is shared, it is separated into three domains (Prefer SM, L1 and EQ). The border color indicates the type of the configuration. Base (blue), support (orange) and predictable (green) configurations. To estimate the performance of the filled green configuration, the correct domain, as well as the filled base and support configurations have to be chosen.

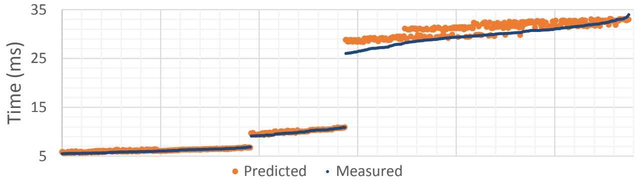


Figure 2: Measured versus predicted execution times for the main kernel of COMIC in all of its configurations, sorted for the measured values. The predicted values can suffer from noise and deviation, visible in the second half of the plot.

and support configurations are selected. Given the measured execution time $T(C)$ of the base and support configurations, it is possible to use the following equation to predict the execution time $P(C_P)$ of a specific configuration:

$$P(C_P) = T(C_B) + \sum_{d_I \in D_I} \underbrace{\left(T(C_{S,d_I}) - T(C_B)\right)}_{\Delta(C_{S,d_I}, C_B)}. \quad (1)$$

To estimate the performance of the entire solution space, all main axes of each domain have to be profiled. Fig. 2 shows an example for a prediction. As can be seen, the method can suffer from noise and deviation, caused by non-linear effects (e.g., varying occupancy). However, this has not shown any significant negative effects on the optimization. For more details on MATOG and the prediction algorithm, please refer to Weber et al. [16].

5. APPLICATION ANALYSIS

In order to apply our optimizations, we had to incorporate a series of changes to the MATOG auto-tuner: This includes full application profiling, an additional post-profiling step and finally adjustments to the runtime environment to apply our adaptive optimizations.

5.1 Automatic Meta Data Gathering

The ability to choose different layouts every time an array is allocated enables us to react to the effects caused by different input data, e.g., varying array sizes. This is neces-

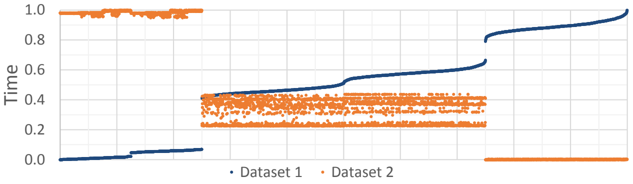


Figure 3: The execution time of a kernel run with two different data sets. The normalized results (0: best, 1: worst) are sorted for the first data set (blue). Configurations which have been optimal for the first data set are worst for the second.

sary for some kernels since they can have different optima for varying input data, as shown in Fig. 3.

This raises the question of how to select the optimal configuration for a given input data set. The most convenient way is to use some kind of metric which identifies the class of input data. Some auto-tuners [6, 11] use user defined callback functions. We do not follow this strategy as we do not want to put the burden of identifying data characteristics onto the user. Furthermore, such a user-driven identification could introduce a runtime overhead depending on the complexity of the analysis performed. We therefore use the meta data that we already have in our system, such as the launch configuration of kernels and size of arrays. We extended the profiling capabilities of the MATOG auto-tuner to automatically record this data during each profiling run.

5.2 Kernel Profiling

One major problem of the profiling and analysis is the number of available configurations, as it can be extremely high, even in small applications. This is caused by the fact that each kernel usually uses a variety of different arrays where each array can have up to three optimization dimensions: AoS layout, transposition and whether global or texture memory should be used. This alone suffices to reach several million of configurations for a single kernel. As we can choose different layouts every time an array is allocated we do not only have to consider all configurations of single kernels, but also all permutations that are possible by allocating arrays inside loops, as in every iteration the layout can be different. In the following we call these application-wide permutations *sequence* to differentiate from local kernel permutations. This leads to a nearly unlimited number of possible combinations to profile. However, this can be avoided by profiling every kernel separately and then calculating the total execution time of all sequences as the sum of the kernel calls. This is based on the assumption that there is no difference between executing a sequence entirely at once or to execute each kernel separately and sum up the execution times which should be fulfilled except for occurring noise.

To gather all necessary data required for the reconstruction, we have change some parts of the auto-tuner. MATOG now uses predefined compiler macros to uniquely identify an array allocation inside the code and use this information to assign a unique ID. For each of these IDs, we later build a decision model. Listing 1 shows an example code snippet for the ID assignment. We also trace the usage of data, when it is allocated, copied or used in a kernel.

Listing 1: Assignment example for unique array IDs.

```
Array2D::Host   a(X, Y, _fl);  // code.cpp:1 --> ID: 1
Array2D::Device b(X, Y, _fl);  // code.cpp:2 --> ID: 2
Array::Device   c(X, _fl);     // code.cpp:3 --> ID: 3

cuMemcpyHtoD(a, b, ...);
c.setMode(WRITE_ONLY);

for(...) {
    Array::Device d(X, _fl);   // code.cpp:9 --> ID: 4
    cuLaunchKernel(...);
}
```

Further we modified the profiling controller, since we do not need to find the minimum of a single kernel but the minimum of the entire application. As we rely on the prediction capability, we do not need to explicitly sample the best kernel configurations as done in the original code.

With the gathered meta data and the adapted profiling, we are able to reconstruct all possible application sequences using a so called *Array Dependency Graph* (**ADG**).

5.3 Array Dependency Graph

The idea of the ADG is to build a graph that represents the entire application work flow, i.e., how and when data is used in the kernels. To construct the ADG, we create one node for each allocation, memcopy and kernel call which have been recorded during the application profiling. These nodes are then connected by directed edges, where each edge represents the usage of a specific array. This leads to a graph, which has one allocation node for each array in the application and one path per array modeled by edges. It represents the sequential processing order of all memcopy and kernel calls that operate on the particular array. This path ends when the array is deallocated. Depending on the kind of application and how data is used, the resulting ADG is not necessarily fully connected and can consist of multiple disjoint graphs. In total, the ADG is an abstract representation of the application where all arrays, memcopies and kernel calls are placed in relation to each other.

For our analysis of the application, we further require an additional node type. These nodes represent the point during the execution at which the array layout has to be decided. These nodes decide explicitly on the AoS layout and/or transposition of their specific array.

This is important as one of our limitations is that we do not support layout conversions once an array is allocated. Such a feature introduces a very complex problem which would exceed the scope of this paper, since it does not only require the handling of the actual conversion process but also an estimate if whether the additional time for converting is worth the improved performance. We will show in Sec. 7 that, at least in the benchmarks we have evaluated it did not show any mentionable deficit to not perform conversions.

Resulting from this, the placement of the decision nodes is defined by the following rules:

- Each host array directly decides its layout after it is allocated. This is necessary as usually the user directly inserts data into them.
- Device arrays are only initiated by their first usage in a kernel.
- Arrays inherit the layout when they are overwritten by a memcopy and therefore do not require any decision nodes.
- Device arrays, which are marked as write-only can be decided on every time they are used as this implies that no

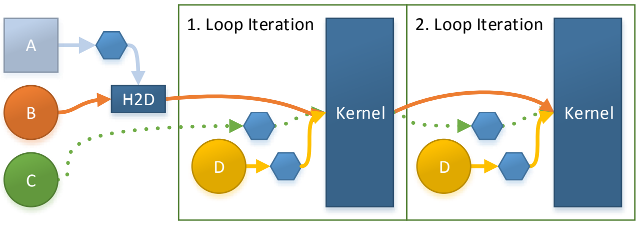


Figure 4: Directed ADG, with one host array (square), four device array allocation (circles), two kernel calls and one memcopy. Decision nodes are indicated by hexagons. The colored edges indicate the usage of the specific arrays. The array C is marked as write-only (dotted line) and therefore has a decision node, prior to each kernel call.

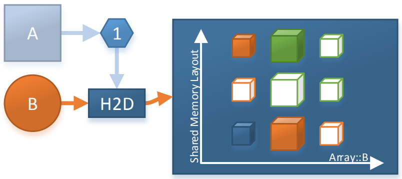


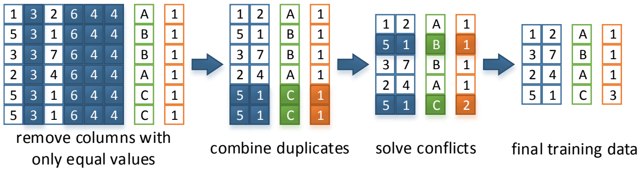
Figure 5: Finding the minimum in a prediction domain: The value for the *Array::B* dimension is fixed to the value "1", so only the big cubes can contain the minimum. To determine the correct option, the best support configuration (orange) of the *Shared Memory Layout* dimension is searched. The optimal configuration (green) is then located at the intersection of the best *Shared Memory Layout* and the fixed value of *Array::B*, indicated by the filled green cube. This is repeated for all kernel calls inside the graph.

data in the array will be used by the kernel call and therefore can be reassigned a more optimal layout without any conversion.

Fig. 4 shows an example for an ADG with all four cases, for the code given in Listing 1. To use the ADG for our optimization, we sum up all execution times of the kernel calls and get the total GPU execution time of the application. The graph itself is used to limit the search space for the kernel calls as arrays that are shared between calls cannot be converted in between and therefore configurations with different layouts are invalid.

5.4 Optimal Application Sequence

In order to find the optimal solution for the profiled application, we have to identify the best sequence. We do this with a brute force search. For each sequence we initialize the values of the decision nodes and then use our prediction formula (Eq. 1) to calculate the best time of all kernel calls, while optimizing dimensions that are not defined through a decision node. These times are then summed up and used to determine the sequence with the lowest execution time. Fig. 5 shows an example for this. Given the optimal solution that we have found through the ADGs and the meta data, we can train models that predict good working configurations.



1	3	2	6	4	4	A	1
5	3	1	6	4	4	B	1
3	3	7	6	4	4	B	1
2	3	4	6	4	4	A	1
5	3	1	6	4	4	C	1
5	3	1	6	4	4	C	1

remove columns with
only equal values

1	2	A	1
5	1	B	1
3	7	B	1
2	4	A	1
5	1	C	1
5	1	C	1

combine duplicates

1	2	A	1
5	1	B	1
3	7	B	1
2	4	A	1
5	1	C	2

solve conflicts

1	2	A	1
3	7	B	1
2	4	A	1
5	1	C	3

final training data

Figure 6: Example for our training data preparation. We require three steps to get our final training set. The meta data is indicated in blue, the result vector in green and the count in orange. Affected fields are highlighted. The size and the values of the training set are simplified for easier demonstration.

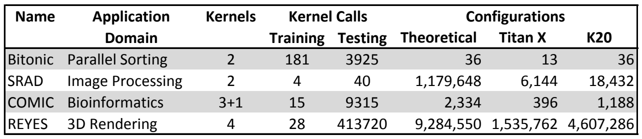
5.5 Model Construction

To establish our decision models, we have to perform some data consolidation. First of all, each decision node is assigned to a specific array. Further, each of these arrays has a unique ID assigned through its location in the source code. We group all decision nodes by the unique ID of their assigned array. Depending on how often an array was reallocated or how often the application was profiled, we end up with one or more decision nodes per unique ID. Each of these contains an optimal configuration (as a result of the ADG analysis).

For our training data set, we first create a result vector, where the optimal configuration of a decision node is stored in a row. Further we create a meta data matrix containing data gathered during the profiling. The matrix has one row per decision node and one column for each array size dimension per unique ID. For example, if two unique IDs have been recorded during the profiling, one is a 1D array and the second a 4D array, then the matrix would have 5 columns. We initialize the values of the matrix with zeros and then iterate all allocation nodes that precede the particular decision node and set the values with the allocation sizes.

Now, each row in the data set represents an optimal solution that has been found for a given meta data set, while each column in the matrix stands for a meta data field. As the data usually contains a lot of redundancies, we reduce it in multiple steps. First, we remove all columns that have only one value, as they are no help for our predictions. This occurs quite often, as certain array sizes or launch configurations are set fixed and do never change. Then, we combine equal rows while we increment a counter for each combined row. In the last step we check if there are multiple best solutions for the same meta data. If this is the case, we choose the row, with the higher count as this has proved to be optimal in most cases. Fig. 6 shows an example for our training data preparation.

Such a training data set is created for each unique ID, i.e., for each array that is allocated in the application. For each of the data sets we train a SVM model which is then used to decide on the optimal configurations during runtime. Further, we employ the same technique for each kernel. This way, we can dynamically decide on the optimal layout for shared memory arrays, L1 cache size, texture memory usage and user defined preprocessor optimizations. The only difference is that we also add the launch configuration of the kernel to the meta data set. If the final data set contains



Name	Application Domain	Kernels	Kernel Calls		Configurations		
			Training	Testing	Theoretical	Titan X	K20
Bitonic	Parallel Sorting	2	181	3925	36	13	36
SRAD	Image Processing	2	4	40	1,179,648	6,144	18,432
COMIC	Bioinformatics	3+1	15	9315	2,334	396	1,188
REYES	3D Rendering	4	28	413720	9,284,550	1,535,762	4,607,286

Table 1: Overview of all applications with the number of kernels, number of kernel calls during training and testing, theoretical configuration count and the exact count for the Titan X and K20. The theoretical count also contains configurations that cannot be executed due to application constraints.

only one row, we do not build a SVM and store instead the result directly as optimal solution for this particular array or kernel. For the SVM model we use the OpenCV Machine Learning library.

5.6 Adaptive Runtime Environment

The next step is to adjust the runtime environment of MATOG to gather the necessary meta data during an optimized application run and to use the decision models to determine the optimal configurations.

Tracing the meta data is quite simple as we use the same format used for our training data matrix. Thus, we store all allocation sizes for each unique ID. If an array with the same unique ID is allocated, it overwrites the values of a previous allocation.

To apply our optimal decisions, we differentiate between host and device arrays and follow the previously introduced rules for placing the decision nodes. When a host array is allocated, it evaluates the decision model that is assigned to its unique ID using all traced meta data. This direct initialization is necessary as host arrays can directly be accessed by the program.

For device arrays the direct initialization is not necessary. They are initialized by their first usage where we again differentiate two cases. If data is copied from another array, it inherits the layouts from the source array. Otherwise, if the array is directly used in a kernel call, we initialize it prior the call by evaluating its assigned decision model. As the MATOG framework allows to mark arrays as write-only, we reinitialize all write-only arrays prior each kernel call. As mentioned before, this indicates that the array will not be read from in the kernel and therefore the layout can be changed as no data has to be preserved.

Finally, every time a kernel is executed, we determine the optimal configuration for shared memory layouts, L1 cache size, usage of texture memory or user defined preprocessor optimization depending on the array meta data and the launch configuration.

6. EVALUATION

We evaluated our approach on four applications: BitonicSort and REYES from Weber et al. [16], COMIC [15] and SRAD from the Rodinia [5] benchmark suite (see Tab. 1). Note that the Maxwell architecture no longer supports to adjust the L1 cache size. This reduces the number of configurations to one third compared to the Kepler architecture.

BitonicSort is a widely used parallel sorting algorithm [2]. We sort a 1D array of structs with four integer fields (8, 4, 2 and 1 B), ensuring that the 8 B value is sorted first and only if this value is equal, the 4 B value is sorted and so on. This results in a sorted list, for all fields. To ensure conflicting rows, we limit all values to 0 to 1023 (255 for the 1 B field). The application consists of two kernels with a total of 36 configurations: One uses shared memory in loop iterations where it can be used efficiently, while the other directly operates on global memory. Our training data contains two sets with 64 Ki and 128 Ki entries. The testing data consists of five sets with 256 Ki, 512 Ki, 1 Mi, 2 Mi and 4 Mi entries. All data sets contain random values but also entirely or partially sorted sequences.

Speckle Reducing Anisotropic Diffusion (**SRAD**) is used in ultrasonic and radar imaging applications. For training we use two input data sets with 128^2 and 512^2 grid cells and test on two others with 256^2 and 2048^2 cells. This application has a high number of configurations (18.432) for two kernels but its total execution time is extremely low.

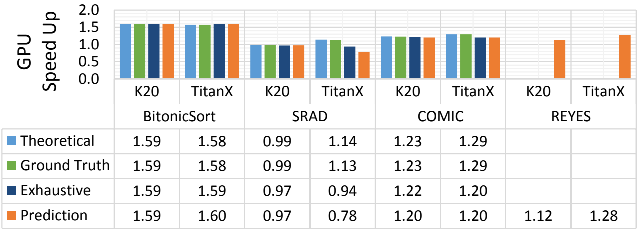
Coevolution via MI on CUDA (**COMIC**) [15] calculates the coevolutionary mutual information for protein and DNA sequences. It consists of three kernels: The first one initializes a randomization seed that is used in a second kernel to permute the sequences. The third kernel performs the main operation by creating a 3D histogram of occurrences in the permuted sequences. This kernel is templated and uses different compression schemes depending on the input data. MATOG handles each template variant as a different kernel. For training we use three data sets with varying complexity ($(\textit{number of sequences} \times \textit{sequence length})$ 140×72, 2261×238 and 16000×99) and run one iteration of the algorithm. For testing we evaluate nine other data sets (211×465, 753×264, 753×275, 211×570, 390×244, 616×336, 4204×120, 500×99 and 376×222) and perform 100 iterations. All kernels together have only 1.188 configurations but the overall execution time of the application is very high compared to the other applications.

Rendering Everything You Ever Saw (**REYES**) [7] is a technique used in movie productions. In contrast to classical 3D mesh rendering, it uses patches to model smooth surfaces. The patches are transformed into micro polygons and iteratively split into smaller polygons until they have subpixel size. The implementation uses four kernels: One of the kernels performs the splitting while the second compresses the resulting data after each iteration. A third kernel uses the final polygons and renders the resulting image into a depth buffer which contains depth and color information for each pixel. Finally a fourth kernel extracts the image information from this buffer into a 2D texture which then can be displayed. For training we render the Utah Teapot at FullHD resolution (1920×1080) and evaluate the same model at three different resolutions (1024×768, 1280×720 and 1920×1080). We render a sequence of 1000 frames varying model rotation in each frame. Although the execution time of a single frame is rather small, the extreme high number of over 4 M kernel configurations prohibits an exhaustive search of configurations which would take more than two weeks to render a single frame.

7. RESULTS

We first analyze the total GPU execution time (see Fig. 7). We distinguish between four cases:

- *Prediction* is our proposed method which uses the prediction based profiling to learn the decision models.



	BitonicSort K20	BitonicSort TitanX	SRAD K20	SRAD TitanX	COMIC K20	COMIC TitanX	REYES K20	REYES TitanX
Theoretical	1.59	1.58	0.99	1.14	1.23	1.29		
Ground Truth	1.59	1.58	0.99	1.13	1.23	1.29		
Exhaustive	1.59	1.59	0.97	0.94	1.22	1.20		
Prediction	1.59	1.60	0.97	0.78	1.20	1.20	1.12	1.28

Figure 7: Kernel Speed Up, normalized by the performance of the original benchmark code. The minimal higher speed up for the Exhaustive and Prediction measurements in the BitonicSort benchmark are caused by GPU clock boost effects.

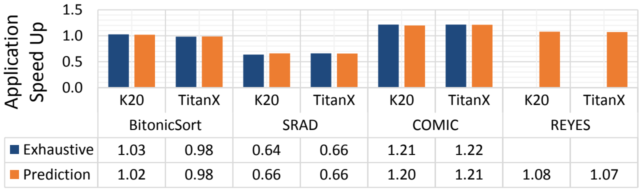
- *Exhaustive* uses our application analysis method but does not rely on the prediction. Instead, it uses exhaustive profiling data to learn the decision models.
- *Theoretical* would be the optimal performance if we allowed data conversions between kernel calls.
- *Ground Truth* is the optimal solution for the tested data sets, without data conversions.

All results are normalized by the execution time of the original implementation provided by the respective authors. Due to the high number of variants, Theoretical, Ground Truth and Exhaustive results are not available for REYES. A speed up of close to 1.0 means that the original implementation is fully optimized. We repeated all tests 5 times. Nevertheless, performance variations of 1-2% can still occur due to noise or the GPU clock boost.

Fig. 7 shows no significant difference between Theoretical and Ground Truth, indicating that data conversions between kernels would not provide any measurable improvement in all tested applications. Further, except for the SRAD and COMIC on the Titan X, our models chose configurations which are similar to the Ground Truth. Some performance drop is expectable as decisions made on unknown data sets are approximate as the models have been trained on different data. The graph also shows that Exhaustive and Prediction achieve the same performance in nearly all cases. This is a very good result as the predicted cases use significantly less input data and time but still achieve nearly the same results. The only case where we observe a significant performance drop is SRAD on the Titan X. The size of the grid does not allow conclusions on the optimal layout so that MATOG selects non-optimal configurations.

7.1 Application Execution Time

We now take a closer look at the speed up achieved, for the application execution. The results are shown in Fig. 8. The application speed up is much lower than the speed up of kernels themselves, which is obvious as the serial CPU time is not optimized and therefore lowers the overall speed up. For the BitonicSort, the difference between kernel and application speed up is very high. This is caused by two effects. First, the CPU only time of the application is significantly higher than the GPU time, as can be seen in Fig. 9. Second, the optimal GPU layout is SoA which is less efficient on the CPU. Also the input data is stored as binary AoS format and therefore has to be implicitly converted during the loading procedure. The SRAD uses close to 100% of the execution for the CPU so that any improvements of the GPU performance would not be visible in the results. The drop of CPU performance is caused by overhead of the auto-tuner and memory layouts that work less efficient on the CPU. COMIC and REYES, as the most complex of our applications, have a much higher GPU to CPU ratio and greatly benefit from the improved GPU performance. For the COMIC case, the memory layouts not only worked better on the GPU but also on the CPU which improved the CPU performance compared to the original implementation. REYES, however, has a slightly lower CPU performance, but this is compensated by the much higher GPU performance improvement.



	BitonicSort K20	BitonicSort TitanX	SRAD K20	SRAD TitanX	COMIC K20	COMIC TitanX	REYES K20	REYES TitanX
Exhaustive	1.03	0.98	0.64	0.66	1.21	1.22		
Prediction	1.02	0.98	0.66	0.66	1.20	1.21	1.08	1.07

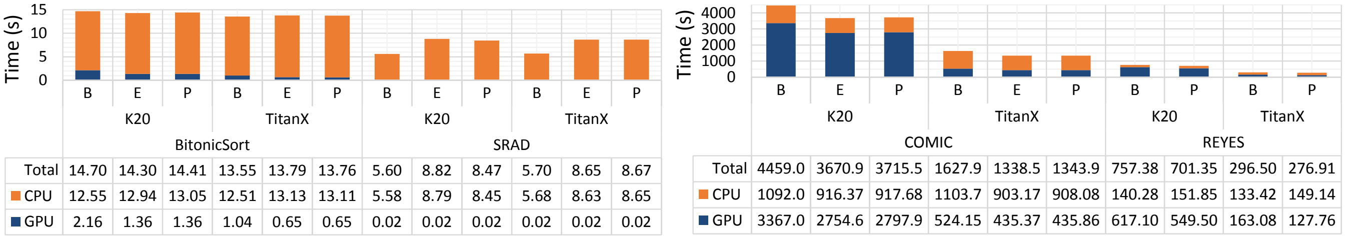
Figure 8: Speed up of the application execution time. The values are lower than for the kernel speed up, as this also contains the auto-tuning overhead and the serial CPU time.

7.2 Optimization Time

Finally we take a look at the time required for profiling and analysis of the application. The results for the predictive profiling compared to an exhaustive search are shown in Fig. 10. As can be seen, the time required for all applications varies significantly, depending on the complexity and execution time. For the prediction based profiling it is possible that after the learning an additional compiling step is executed. In this case, configurations have been chosen to be optimal which were not profiled and therefore not compiled before. As we do not want to compile these during an optimized application run, all configurations that are optimal and have not been compiled before are compiled at the end of the learning process. The slower learning time for some exhaustive results is caused by the fact that the application has to handle much more profiling data as in the prediction case. In this case it cannot calculate the minimum of a kernel call using the prediction formula but has to search for it explicitly in the profiling data, which can contain millions of entries. Further we can see that the prediction based method is 1 (BitonicSort) to 114 (SRAD) times faster than the exhaustive search. For COMIC on the K20 both methods required the most time: It took less than 6 minutes for the predictive method while the exhaustive method took about 100 minutes to achieve similar performance.

8. CONCLUSION

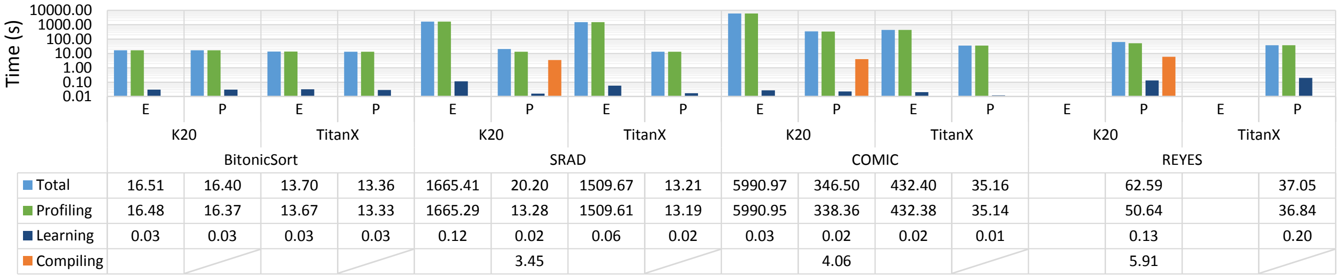
We presented an adaptive optimization for the MATOG auto-tuner that reacts to changing data characteristics. Our application analysis method is able to analyze and optimize complete GPU applications within minutes with very little user interaction and achieves or even exceeds performance of hand optimized code. We note that applications with an execution time of only a couple of seconds do usually not benefit much from our auto-tuner as the optimization potential is



	B	E	P	B	E	P	B	E	P	B	E	P
	K20			TitanX			K20			TitanX		
	BitonicSort						SRAD					
Total	14.70	14.30	14.41	13.55	13.79	13.76	5.60	8.82	8.47	5.70	8.65	8.67
🟧 CPU	12.55	12.94	13.05	12.51	13.13	13.11	5.58	8.79	8.45	5.68	8.63	8.65
🟦 GPU	2.16	1.36	1.36	1.04	0.65	0.65	0.02	0.02	0.02	0.02	0.02	0.02

	B	E	P	B	E	P	B	P	B	P
	K20			TitanX			K20		TitanX	
	COMIC						REYES			
Total	4459.0	3670.9	3715.5	1627.9	1338.5	1343.9	757.38	701.35	296.50	276.91
🟧 CPU	1092.0	916.37	917.68	1103.7	903.17	908.08	140.28	151.85	133.42	149.14
🟦 GPU	3367.0	2754.6	2797.9	524.15	435.37	435.86	617.10	549.50	163.08	127.76

Figure 9: Total execution time for the *Base* (B) implementation, *Exhaustive* (E) and *Predictive* (P) optimized application (sum of all testing data sets, five repeated runs per data set). See text for details.



	E	P	E	P	E	P	E	P	E	P	E	P	E	P	E	P
	K20		TitanX		K20		TitanX		K20		TitanX		K20		TitanX	
	BitonicSort				SRAD				COMIC				REYES			
🟦 Total	16.51	16.40	13.70	13.36	1665.41	20.20	1509.67	13.21	5990.97	346.50	432.40	35.16		62.59		37.05
🟩 Profiling	16.48	16.37	13.67	13.33	1665.29	13.28	1509.61	13.19	5990.95	338.36	432.38	35.14		50.64		36.84
🟦 Learning	0.03	0.03	0.03	0.03	0.12	0.02	0.06	0.02	0.03	0.02	0.02	0.01		0.13		0.20
🟧 Compiling						3.45				4.06				5.91		

Figure 10: Time to gather profiling data, analyze and build decision models, compilation compared between *Exhaustive* (E) and *Predictive* (P) profiling. Compiling times are only needed if an optimal configuration has not been explicitly profiled (and therefore has not been compiled) before. Note the logarithmic time scale.

too low to compensate for the overhead created by the autotuner. Further, solely optimizing the GPU execution time does not necessarily result in better overall performance, as the CPU performance can decrease equally depending on the array layout.

The source code of our MATOG system is available at http://tinyurl.com/matog under the New BSD License.

Acknowledgements The work of Nicolas Weber is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

9. REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *PACT*, 2014.

[2] K. E. Batcher. Sorting Networks and Their Applications. In *Proc. SJCC*, 1968.

[3] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin. Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle. In *Proc. SC*, 2015.

[4] J. Bergstra, N. Pinto, and D. Cox. Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees. In *Proc. INPAR*, 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[6] I.-H. Chung and J. K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proc. SC*, 2004.

[7] R. L. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. In *SIGGRAPH*, 1987.

[8] K. Kofler, B. Cosenza, and T. Fahringer. Automatic Data Layout Optimization for GPUs. In *Proc. Euro-Par*, 2015.

[9] A. Li, G.-J. v. d. Braak, A. Kumar, and H. Corporaal. Adaptive and Transparent Cache Bypassing for GPUs. In *Proc. SC*, 2015.

[10] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for GPU program optimizations. In *Proc. IPDPS*, 2008.

[11] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A Framework for Adaptive Code Variant Tuning. In *IPDPS*, 2014.

[12] J. J. K. Park, Y. Park, and S. Mahlke. ELF: Maximizing Memory-level Parallelism for GPUs with Coordinated Warp and Fetch Scheduling. In *SC*, 2015.

[13] X. Shen, Y. Liu, E. Z. Zhang, and P. Bhamidipati. An Infrastructure for Tackling Input-Sensitivity of GPU Program Optimizations. *International Journal of Parallel Programming*, 41(6):855–869, 2013.

[14] H. H. B. Sørensen. Auto-tuning Dense Vector and Matrix-vector Operations for Fermi GPUs. In *Proc. PPAM*, 2012.

[15] M. Waechter, K. Jaeger, S. Weissgraeber, S. Widmer, M. Goesele, and K. Hamacher. Information-theoretic Analysis of Molecular (Co)Evolution Using Graphics Processing Units. In *Proc. ECMLS*, 2012.

[16] N. Weber, S. C. Amend, and M. Goesele. Guided Profiling for Auto-Tuning Array Layouts on GPUs. In *Proc. PMBS*, 2015.

[17] Y. Zhang and F. Mueller. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proc. CGO*, 2012.