MATOG: Array Layout Auto-Tuning for CUDA

NICOLAS WEBER, Graduate School of Computational Engineering, TU Darmstadt

MICHAEL GOESELE, TU Darmstadt

Optimal code performance is (besides correctness and accuracy) the most important objective in compute intensive applications. In many of these applications, Graphic Processing Units (GPUs) are used because of their high amount of compute power. However, caused by their massively parallel architecture, the code has to be specifically adjusted to the underlying hardware to achieve optimal performance and therefore has to be reoptimized for each new generation. In reality, this is usually not the case as productive code is normally at least several years old and nobody has the time to continuously adjust existing code to new hardware. In recent years more and more approaches have emerged that automatically tune the performance of applications toward the underlying hardware. In this article, we present the MATOG auto-tuner and its concepts. It abstracts the array memory access in CUDA applications and automatically optimizes the code according to the used GPUs. MATOG only requires few profiling runs to analyze even complex applications, while achieving significant speedups over non-optimized code, independent of the used GPU generation and without the need to manually tune the code.

CCS Concepts: • **Computing methodologies** → *Parallel programming languages*; • **Software and its engineering** → *Software performance*; *Source code generation*; *Runtime environments*;

Additional Key Words and Phrases: CUDA, MATOG, auto-tuning, array layouts, GPGPU

ACM Reference format:

Nicolas Weber and Michael Goesele. 2017. MATOG: Array Layout Auto-Tuning for CUDA. *ACM Trans. Archit. Code Optim.* 14, 3, Article 28 (August 2017), 26 pages.

<https://doi.org/10.1145/3106341>

1 INTRODUCTION

With the demise of Moore's Law, one key direction in the development of processors has changed from shrinking the architectures to establishing specialized multi- or many-core processors. This allows one to further improve the compute power of today's computers using parallel computation without the physical limitations imposed by further shrinking the architectures. As the word

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. We want to thank Stefan Guthe for his help analyzing the optimized code.

Extension of conference papers. The additional contributions of this article over the previously published work of Weber et al. at EGPGV'14, PMBS '15, and SEM4HPC '16 are described in the Introduction.

Authors' addresses: N. Weber and M. Goesele, Department of Computer Science, TU Darmstadt, Rundeturmstr. 12, 64283 Darmstadt, Germany; emails: {nicolas.weber, goesele}@gcc.tu-darmstadt.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/08-ART28 \$15.00

<https://doi.org/10.1145/3106341>

specialized implies, code for these processors cannot be naïvely implemented. Instead, it has to be specifically adjusted to the underlying hardware to leverage its full capabilities. Graphics Processing Units (*GPUs*) are one of these specialized hardware architectures. Their massively parallel many-core architecture in combination with a very high memory bandwidth make them ideal for solving many problems in a broad range of application domains. Unfortunately, the underlying architecture requires that the programmer has a very good understanding of the hardware, as he has to explicitly use multiple different kinds of memory, utilize multiple cache hierarchies with varying capabilities, and even use special hardware functions, whereas all of these techniques somehow come with restrictions, limitations, or usage requirements. The fact that vendors release a new GPU architecture every 1 to 2 years makes it more tedious as these usually come with significant changes to the architecture including new, changed, or even removed features and functionality. So far, this required one to specifically tune the code for each new GPU architecture, otherwise it is possible that code optimized for an older GPU runs much less optimally on newer hardware. As most users in scientific or High Performance Computing (*HPC*) are not computer scientists but come from fields such as physics, chemistry, biology, mechanical, or electrical engineering it is essential that these can focus on their actual work and do not have to worry about changing hardware architectures. That is why many approaches have emerged to assist these scientists by abstracting the code and automatically determining optimal implementations.

So far, in every new NVIDIA GPU architecture the memory hierarchy has been significantly changed, while the programming model stayed the same, making this an interesting target for auto-tuning. Changes include how the L1 cache works (Fermi: serves L2; Kepler: serves only local memory; Maxwell: unified with non-coherent cache) and how texture memory can be used (texture references, unbound textures (since Compute Capability (*CC*) 3.0), non-coherent cache without textures (since *CC* 3.5)) (NVIDIA 2014). Further, even within the same GPU generation, there is a huge difference in used memory technologies, e.g., the memory type (DDR3, GDDR5, GDDR5X, and HBM), the memory bus width ranging from 40 to 4,096 Bit, or the number of multi-processors sharing the same memory. The biggest problem of optimizing the memory access in GPU applications is that it not only affects a small piece of code, but involves every GPU function (in the following, called *kernels*) where the data is used and the host code, as all parts of the application have to use the same data layouts. Depending on the size of an application and the number of kernels, this can be a very tedious, time-consuming and error-prone task to optimize manually. Furthermore, such applications usually use numerous different algorithms, so it can happen that different data layouts are optimal in different kernels that share the same data. In this case, either the data has to be converted in between the executions or a compromise has to be found to achieve optimal overall performance. Besides, depending on the input data, the optimal layout can vary significantly, so that purely static implementations do not necessarily leverage the full potential.

To ease this process, we developed the MATOG auto-tuner, which aims to optimize the array access in CUDA applications fully automatically without user interactions. It is capable of dynamically adjusting data layouts during runtime to optimize the performance for the given input data. The intention behind MATOG is to ease the development process and to enable performance portability by providing a tool that can automatically optimize a given piece of software. If this software is used on different (including future) hardware it was not developed for, the software may not run optimally without modifications. In this case, the automatic optimization procedure can be executed, which requires no code adjustments, and therefore can be performed by any user even without any programming experience. We introduced the concept of MATOG in Weber and Goesele (2014) and have significantly improved it since then. In Weber et al. (2015), we proposed a prediction-based profiling method to significantly reduce the time required to find optimal

configurations. Further, we introduced an adaptive runtime component in Weber and Goesele (2016) that dynamically selects optimal configurations according to automatically gathered meta-data. The contributions of this article are as follows:

- A unified and integrated description of the MATOG components published in the previous conference publications.
- An improved application analysis algorithm that further simplifies the application analysis, enabling one to analyze even extremely complex applications within seconds. It limits the available search space while achieving equal results to an exhaustive search.
- A new runtime decision-making system that takes not only the most recent but also previously collected metadata into account.
- A nearest-neighbor search based decision model that implicitly normalizes metadata, achieving higher performance in some benchmarks compared to a Support Vector Machine (SVM).
- An extensive evaluation on a broad range of six applications on eight GPUs, ranging from the Fermi up to the Pascal architecture, i.e., four architecture generations.

This article is structured as follows: Section 2 gives an overview of other auto-tuners and where MATOG is positioned in relation to these; Section 3 introduces the concept behind MATOG and its programming interface; Section 4 explains how we analyze CUDA applications and how to determine optimal configurations; Section 5 explains how our decision making is performed during runtime; Section 6 evaluates the overall system on a series of different CUDA applications; and in Section 7 we discuss our results and conclude with an overview on open issues and possible future work.

2 RELATED WORK

This section gives a brief overview of prominent auto-tuners, their optimizations, and used techniques. First, an overview of what can be optimized in GPU applications is given, followed by how these optimization approaches are integrated into the applications, how optimal configurations are found, how dynamic auto-tuners identify different data properties, and how this information is used during runtime to select optimal configurations. We mainly concentrate on GPU auto-tuning but also include generic auto-tuning approaches.

2.1 “What to Optimize?”

The variety of available optimizations in applications is nearly endless. It starts with simple and easily applicable optimizations such as compiler flags (Ansel et al. 2014). As compute-intensive applications usually iterate over a lot of data, loop transformations (Hall et al. 2009) and optimized launch configurations (Bergstra et al. 2012; Liu et al. 2008) are very important. Lutz et al. (2015) optimize the usage of Application Programming Interface (API) calls to optimally overlap memory copy and execution. Magni et al. (2014) apply thread coarsening by fusing multiple work items. As GPUs are usually used due to their high compute power and memory bandwidths, an optimal data access is necessary. This can either be done by specifically targeting particular operations such as in 3D stencil codes (Lutz et al. 2013), dense (Sørensen 2012) or sparse (Choi et al. 2010; Monakov et al. 2010) matrix vector multiplications, or by applying application-independent optimizations, e.g., tuning the cache utilization (Li et al. 2015), data mapping and partitioning (Ben-Nun et al. 2015), data placement (Li et al. 2015b), or data layouts (Kofler et al. 2015; Peng et al. 2016; Edwards and Trott 2013) (while the latter only provides an optimization API and no auto-tuning). Sung et al. (2012) propose a framework that transforms access to arrays of struct type based on static rules and explicit data transformations prior to the kernel execution. Hsu et al. (2014) go one step further

and propose a hardware controller that performs the necessary layout transformations during the memcopy, eliminating the conversion overhead. More general are approaches such as Muralidharan et al. (2014), Ansel et al. (2014), and Nugteren and Codreanu (2015) that allow user-specified optimizations, e.g., alternative implementations and algorithms. MATOG mainly optimizes data layouts, placement, and utilization of caches, but also allows user-specified optimizations.

2.2 “How to Integrate Optimizations into the Application?”

There are numerous ways to integrate optimizations into an application. Most prominent is the usage of compilers. Some approaches use code annotations (Lee and Vetter 2014; Han and Abdelrahman 2009; Liu et al. 2008) to generate parallel code for given serial code, or use them to apply optimizations such as loop transformations. Li et al. (2015a) apply their optimizations directly to the PTX intermediate format. Auto-tuners that allow user-defined optimizations (as mentioned before), usually require using a specific API to instruct the auto-tuner what to optimize and annotations or macros inside the code, which enable/disable features or adjust parameters. For MATOG we decided to use code generation, which has to be manually integrated into the application and replaces array data types with a MATOG-specific implementation. This is less comfortable to use, as it requires some manual work, but makes the code easily portable between different operating systems and compilers. Our API mimics the CUDA Driver API and therefore requires only a few changes to existing CUDA applications.

2.3 “How to Find Optimal Configurations in Short Time?”

Finding optimal configurations or values for parameters is essential for auto-tuning. Mainly there are two methods: to analyze the code (static analysis) and to measure the execution time (profiling). Peng et al. (2016) recently proposed a metric to estimate if either Array of Structs (AoS), Structure of Arrays (SoA), or SoAoS, a hybrid format between AoS and SoA, yields in better performance. The advantage of such static analysis methods over profiling is that it can be performed quite fast, but ignores data-dependent effects that can occur, as it operates on heuristics which make certain assumptions of the data and workload of the application. Further, as the interior of GPUs is proprietary, it is difficult to establish suitable models for static analysis methods, especially if a new architecture with lots of internal changes is released. On the other hand, empirical profiling is quite time intensive, so that numerous different approaches have emerged to reduce the necessary time. Kofler et al. (2015) use a hybrid approach, as they utilize a generic GPU benchmark to capture GPU-specific properties and then project them onto the code to estimate the performance using a directed execution graph. Magni et al. (2014) train a neural network on a dataset consisting of static code features and best candidates for some reference benchmarks. With this trained model they are able to predict optimal thread coarsening factors based on the static code features. Other approaches use methods such as an exhaustive search (Muralidharan et al. 2014), greedy search (Liu et al. 2008), swarm search, simulated annealing (Nugteren and Codreanu 2015), Nelder-Mead downhill simplex method (Chung and Hollingsworth 2004), or a genetic algorithm (Ansel et al. 2014) to guide the search into the direction of the optimal configuration. In Weber et al. (2015), we proposed a prediction-based empirical profiling method, that requires only a very small subset of configurations to estimate the execution time of non-profiled configurations. This is based on the observation that changes induced by one optimization have little influence on others. With this method, it is possible to easily estimate the optimal configuration with only a very limited number of executed configurations, making this method very fast while it achieves nearly the same results as an exhaustive search.

Fig. 1. Execution time for two kernel runs with different data. All results have been normalized between 0.0 (best) and 1.0 (worst), and are sorted ascending for the first call (black). Configurations which were optimal for the first call, became the worst in the second (blue) execution. The shown kernel is the first main kernel of our KD-Tree benchmark. The big jumps come from using local memory as a buffer for shared memory. This works well for the black iteration, but not for the blue one.

2.4 “How to Identify Different Data Properties?”

To dynamically select optimal configurations during runtime requires that the auto-tuner is able to identify different data properties. Approaches such as Muralidharan et al. (2014) and Chung and Hollingsworth (2004) use user-defined callback functions. This requires that the user knows of the different properties of his data and how to characterize these. In contrast, we use a fully automatic process that relies on metadata that is already in our system (i.e., the launch configuration of a kernel or the size of arrays). However, MATOG also supports monitoring of user-defined variables, which can contain additional information.

2.5 “How to Select Optimal Configurations During Runtime?”

With the ability to identify different data properties, it is possible to create decision models based on machine-learning techniques, e.g., regression trees (Bergstra et al. 2012) or SVMs (Muralidharan et al. 2014) that adjust the application accordingly. In Weber and Goesle (2016) we relied on SVMs, but encountered cases where these performed badly. We now realized that, as the value range for our metadata is unknown, the SVM prioritizes metadata dimensions whose values differ significantly from the training data, causing bad decisions. So we experimented with several other approaches and came up with a nearest-neighbor search that employs an implicit data normalization, yielding equal or better results compared to a SVM.

3 MATOG AUTO-TUNER

MATOG is an application domain independent array layout auto-tuner for CUDA applications. It has been introduced first in Weber and Goesle (2014) and has been constantly improved since then. MATOG is open source and can be downloaded from our project page.¹ The main concept of MATOG is that it abstracts the memory access from the application and dynamically chooses optimal array layouts during runtime. MATOG is meant to work for arbitrary NVIDIA GPUs. In order to achieve its goal, MATOG requires one to run and analyze the application with real data. During this empirical profiling, it measures the execution time of different data layouts, tracks metadata, and stores the results in a database (Section 4.2). These profiling results are then analyzed (Section 4.3) and used to build decision models that are utilized during runtime to select optimal configurations according to changing data properties (Section 4.4). As shown in Figure 1, there

¹matog.org.

```

1  // Define Example: values: 15, 42
2  #if MY_DEFINE == 15
3  // algorithm 1 ...
4  #elif MY_DEFINE == 42
5  // algorithm 2 ...
6  #else
7  #error NOT_IMPLEMENTED
8  #endif
9
10 // Range Example: min: 32, step: 32, max: 512
11 __shared__ int bins[MY_BIN_COUNT];
12 for(...) {
13     int binIndex = (int)(value[i] / MAX_VALUE * MY_BIN_COUNT);
14     //...
15 }

```

Listing 1. Example code to show the define/range feature. The first code snippet shows how different implementations can be differentiated using a preprocessor definition. The second shows how a range of values can be evaluated.

can be different optimal configurations of a kernel, depending on the input data. We automatically identify different data properties and adjust the layouts accordingly.

MATOG optimizes array access for one- or multi-dimensional arrays of primitive, struct, and hierarchical types. Pointers can currently not be used as data type. It arranges these arrays in different struct layouts, i.e., AoS, SoA, or AoSoA (often referenced as tiled-AoS (Kofler et al. 2015) or ASTA (Sung et al. 2012), with a tile size of 32, the size of a CUDA thread group). Further, multi-dimensional arrays can be stored in a transposed way. The array layouts can be separately optimized for global, shared, and local memory. For read-only arrays residing in the global memory, MATOG determines which of these arrays should use the default and which the non-coherent cache (also known as texture memory) to optimize the cache utilization. Arrays with constant size (known at compile time) and content can further be placed in constant memory. On Fermi and Kepler GPUs, the optimal ratio between L1 cache and shared memory size is automatically determined. Additionally, the programmer can define preprocessor optimizations, to use discrete values and value ranges in his code. This can be used to implement different algorithms for a task, or to evaluate a series of values as parameters of an algorithm as shown in Listing 1. Experienced users can further customize MATOG by defining their own indexing schemes, e.g., to implement a special triangular matrix, or to provide their own allocation/deallocation mechanism, in order to combine MATOG with other frameworks.

3.1 Programming Interface

The MATOG programming interface was designed to reduce programming overhead and to be as compatible to CUDA as possible. To support platform independence and easy maintainability, we decided to use code generation instead of an own source-to-source compiler, so no changes to the compile chain have to be applied. MATOG requires only small changes to the source code as it mimics the CUDA Driver API (NVIDIA 2016) interface. It is necessary to use the Driver API instead of the Runtime API as the latter does not allow one to dynamically load and exchange kernel implementations during runtime. To access data MATOG supports a multi-dimensional memory access (e.g., `array[x][y].subarray[z].field`) similar to the code skeletons used by GROPHECY (Meng et al. 2011). To apply its optimizations, it intercepts all communication to CUDA.

Fig. 2. An example struct (left) and how it would usually be stored in CUDA (center). MATOG uses an optimized layout (right) since GPUs require n-Byte variables to be n-Byte aligned.

```

1 typedef struct {
2   long rand;
3   float result;
4 } MyStruct[3][7];

```

Listing 2. Example data structure in C++.

```

1 { "name": "MyStruct",
2   "counts": [0, 3, 7],
3   "fields": {
4     ["name": "rand", "type": "long"],
5     ["name": "result", "type": "float"]
6   }}

```

Listing 3. Data structure from Listing 2 in MATOG JSON notation.

3.2 Programming Example

In order to use MATOG arrays, the programmer has to provide a JSON description (example shown in Listings 2 and 3) of the arrays, which is then used to generate the optimization code. This code can then be included into the application. The main contribution of MATOG is a dynamic selection of optimal parameters at runtime. However, MATOG automatically performs some trivial static optimizations, e.g., it ensures an optimized alignment inside the arrays to prevent alignment spacings (example shown in Figure 2). Further, the user can specify for each kernel how data is used (only read, only written, or read and write), which enables certain optimizations. This is explained in Section 4.3 in more detail. The programmer can further specify whether multiple arrays always have the same size. This allows one to reduce the number of used registers, as normally every MATOG data structure maintains a separate copy for its sizes. Finally, compiler flags can be specified to enable GPU debugging or specific features (e.g., `--use_fast_math`). There are three types of MATOG data structures that can be used in the host code: one for arrays located in host memory (`Array::Host`), one representing dynamic shared memory (`Array::Dyn`), and one for device memory (`Array::Device`). Data inside host arrays can be directly accessed. Data transfers are done by the CUDA `memcpy` functions. For kernel modules, the source code has to be provided instead of a pre-compiled CUDA module. MATOG takes care of the actual compile process in a Just-In-Time (JIT) manner. Nothing else has to be changed for loading and executing GPU code, as MATOG array references can simply be put into the kernel argument list. Listing 4 shows a host code example. Changes are indicated compared to CUDA Driver API. Inside the kernel code, each memory type uses separate implementations. As multiple instances of global or dynamic shared memory arrays can have different layouts, a template is used to distinguish the different instances. Only for static shared memory types, the same layout is used for all instances of the same type. Listing 5 shows an example for kernel code. Figure 3 shows a schematic workflow of MATOG. In the development phase the user specifies the optimizable data structures, which are then generated by the code generator. The user then incorporates these into the application and compiles it using a standard host compiler (e.g., GCC or Visual Studio). Listing 6 shows an example for using a

Fig. 3. Schematic workflow of MATOG. Orange indicates steps that are performed by the user, green is automated by MATOG, and blue is done by the host compiler.

```

1  /* allocate data */
2  int X = ..., Y = 3, Z = 7;
3  MyStruct* host = new MyStruct[X * Y * Z];
4  MyStruct::Host host(X, _fl);
5  CUdeviceptr device = cuMemAlloc(sizeof(MyStruct) * X * Y * Z);
6  MyStruct::Device device(X, _fl);
7  /* load data */
8  for(...) {
9      host[x + y * X + z * X * Y][x][y][z].rand = rand();
10     host[x + y * X + z * X * Y][x][y][z].result = 0.0f;
11 }
12 /* load module */
13 CUmodule module;
14 cuModuleLoad(&module, "preCompiledCode.ptx""sourceCode.cu");
15 /* load function */
16 CUfunction function;
17 cuModuleGetFunction(&function, module, "kernel");
18 /* exec */
19 void* args[] = {&paramA, &paramB, &device, 0};
20 cuLaunchKernel(function, ...);
  
```

Listing 4. Host Code Example: CUDA Driver API (stroked through) compared to MATOG (underlined).

```

1  __global__ void kernel(const float paramA, const int paramB, MyStruct*<u> data) {
2      const int x = threadIdx.x, y = threadIdx.y, z = threadIdx.z;
3      __shared__ MyStructShared<128> shared[128*Y*Z];
4      shared[x + y * X + z * X * Y][x][y][z] = data[x + y * X + z * X * Y][x][y][z];
5      /* some calculations */
6      data[x + y * X + z * X * Y][x][y][z].result = ...;
7  }
  
```

Listing 5. Kernel code example: CUDA (stroked through) compared to MATOG (underlined).

MATOG-based application and how to execute the optimization procedure. By default, MATOG runs in an unoptimized mode. To optimize it, it has to be switched into a profiling mode, which measures the execution time of the kernels. Every time a kernel is executed during optimization, MATOG runs multiple implementations of the same kernel to determine which layouts work best.


```

1  ## execute optimization ##
2  export MATOG_PROFILING=1 # enable profiling
3  ./myApp training_0 ...   # run application one or multiple times
4  ./myApp training_1 ...
5  ...
6  unset MATOG_PROFILING   # disable profiling
7  matog-analyze           # run analyzer
8
9  ## run application ##
10 ./myApp param_0 param_1 param_2 ...

```

Listing 6. Example for running a MATOG application, including the optimization process. If steps 1–7 are omitted, the application will run unoptimized using default implementations for the data structures.

After the profiling, MATOG analyzes the results and builds decision models. These are used to determine optimal layouts during runtime.

4 APPLICATION ANALYSIS

To optimize an application using MATOG, it is necessary to determine which configurations achieve optimal performance. As MATOG is meant to work for arbitrary NVIDIA GPU architectures and applications from all kinds of application domains, we treat GPUs, kernels, and optimizations as black boxes. In particular, we assume that we do not know anything about their specific properties, except for which optimizations can be applied for a given GPU, a particular kernel, and the hints given by the programmer. This potential lack of information might complicate the analysis but guarantees that our concept can be applied to any application on any past, current, and future hardware, as long as the optimization to hardware relation is modeled and as long as the code can actually be compiled on the specific hardware. In the following section, we formalize our optimization problem. Then we explain our algorithm to find optimal solutions for a particular application on a given GPU. We use a three-step application analysis consisting of empirical profiling (Section 4.2) to determine which optimizations are optimal, offline analysis (Section 4.3) to determine the application-wide optimal solution, and decision models (Section 4.4) that select data-dependent optimal solutions during runtime.

4.1 Optimization Problem

Our optimization problem has multiple optimization dimensions: If an array uses struct types, we can optimize the struct layout ($d_L \in [\text{AoS}, \text{SoA}, \text{AoSoA}]$). For multi-dimensional arrays, we can optimize the transposition (d_T), where the number of possible values is the factorial of the number of array dimensions. If data is only read from arrays residing in global memory, we can determine if we use the default or non-coherent cache hierarchy, or place the data in constant memory ($d_M \in [\text{Default}, \text{Non-Coherent}, \text{Constant}]$ *if size is known at compile time). On Fermi and Kepler cards, we can further determine the size of the L1 cache ($d_{L1} \in [\text{Prefer SM}, \text{Prefer L1}, \text{Prefer EQ}]$ *only for Kepler) by trading with shared memory. Additionally, we can select different implementations using user-defined preprocessor instructions ($d_D \in [\dots]$) and value ranges ($d_R \in [\min, \min + \text{step}, \dots, \max]$). As can easily be seen, the number of dimensions for a single kernel can be very high as a single array already can have up to three optimization dimensions. Further, we can see that the number of values for a dimension are very low. This causes a very high-dimensional optimization problem with a very limited extend but a very high number of configurations (C) for a given kernel ($k \in K$), as this is defined by $|C_k| = \prod_{d \in D} |d|$. This gets even more complex when multiple kernels are used as this results in $|C| = \prod_{k \in K} |C_k|$ configurations.

4.2 Step 1: Application Profiling

As processor architectures can significantly change from one to another generation, purely analytical models can break, as new or changed features no longer perform as modeled. Therefore, we solely rely on empirical profiling without any code parsing or specific GPU model. This removes the necessity of constantly updating the used analytical GPU model with each new generation. However, the mentioned high number of configurations is a big problem for empirical profiling. These configurations not only have to be executed but must also be compiled, as the optimizations for the GPU code have to be explicitly compiled to achieve maximal performance. Further, to gather the execution time of the application it has to be restarted in each configuration, which leads to massive overhead especially if a huge amount of I/O is taking place. To tackle these problems, we used a specialized profiling technique in Weber et al. (2015) which relies on an in-application profiling (similar to the NVIDIA CUDA profiler) and a prediction algorithm that performs a specialized search space pruning. For time measurements we rely on the CUDA Profiling Tools Interface (CUPTI). To validate correctness, MATOG not only checks whether configurations can actually be executed (e.g., do not exceed constant memory limitations or try to use texture memory on non-read-only arrays) but also has a verification mode that can be activated during profiling to verify that all configurations produce the same results.

4.2.1 In-Application Profiling. An in-application profiling has the advantage (compared to restarting an application) that it does not require one to repeat any application setup and finalization procedures, which can be very time-consuming depending on how much and from where data has to be read or written to. In addition, the in-application profiling reduces the number of configurations to test from $\prod_{k \in K} |C_k|$ to $\sum_{k \in K} |C_k|$ as we can calculate the execution time for all permutations of kernel configurations and do not need to explicitly measure them. One drawback of this method is, however, that it requires much more memory, as data has to be duplicated so it can be restored prior to (re-)executing the same kernel, in another configuration. Further, it has to be converted into other data layouts. In MATOG, we copy the data to the host system prior to each kernel execution and start parallel CPU threads to convert data if necessary. We compile all necessary kernel configurations in parallel to the GPU profiling, overlapping compilation and execution. This is realized using multiple threads so it does not interfere with the profiling of the kernel execution.

4.2.2 Prediction-Based Profiling. In Weber et al. (2015), we have introduced a prediction-based algorithm that requires only very few samples to estimate the performance of the entire optimization space for a single kernel. This is based on the observation that many optimizations do not influence others. Formally, this means that many dimensions are linear independent of others and can be optimized independently of these. This allows us to use a difference model to predict the execution time of a kernel in a specific configuration ($p(k, c_p)$). For this we require a reference point that we call *base configuration* (c_b) (that can be arbitrarily selected). Further, several additional data points are required, called *support configuration* ($c_{s,d}$). For these, all values are equal to the base configuration, except for the value of one dimension, d . To predict the performance of a specific configuration, we then use the difference ($\Delta(t_1, t_2) = t_1 - t_2$) of the execution time between support and base configurations, sum them up, and add the time of the base configuration.

$$p(k, c_p) = t(c_b) + \sum_{d \in D} \underbrace{(t(c_{s,d}) - t(c_b))}_{\Delta(c_{s,d}, c_b)}. \quad (1)$$

However, in reality this does not always work, as not all dimensions are independent of the other ones. This is caused by the fact that some optimizations, e.g., the L1 cache size (d_{L1}), significantly

Fig. 4. **Left:** all c_b (gray) and c_s (blue) that have to be profiled to estimate the performance for all non-profiled configurations (white) for three optimization dimensions. **Right:** Selection of the c_s and c_b to predict the execution time of c_p (green).

Fig. 5. Example for storing a 5×5 AoS. Each line represents a data line with four items. Blue boxes show accessed items in the first iteration of the code. Red bars indicate where the data is scattered over the memory. When AoS is used, each iteration of the inner loop (although executed in parallel) requires one to read a separate non-connected data line. In contrast, for AoS^T only three neighboring lines have to be read. SoA^T is the best layout that only needs to read two consecutive lines.

influence the availability of resources and can thus lead to strongly varying hardware utilizations. Other dimension types (e.g., layouts or transpositions) are mostly independent and hardly change the amount of used hardware resources. Given this influence, we have to modify the prediction formula. First, we divide all dimensions into two sets. The first contains *independent dimensions* (D_I) and the second contains all that have an influence on others, which we further call *shared dimension* (D_S). For each value combination of the D_S , we create a separate prediction domain. Only configurations that are located inside this domain can be used to predict configurations in this domain. Simply spoken, this means that each domain requires its own base configuration and consequently, also corresponding support configurations. For the profiling, this means that we have to execute all $D_S \otimes D_I$ combinations to gather all required measurements to estimate the performance of all other configurations (shown in Figure 4). The number of these configurations is significantly smaller than for an exhaustive search, as can be seen by

$$\prod_{d_s \in D_S} |d_s| \cdot \sum_{d_i \in D_I} |d_i| \ll \prod_{d \in D} |d|, \text{ with } D = D_S \cup D_I \text{ and } D_S \cap D_I = \emptyset. \quad (2)$$

Figure 4 shows an example on how the support configurations are selected for the prediction. Note that we empirically showed that this works on NVIDIA GPUs but cannot provide any proof of why this works, as the GPUs and the compilers are proprietary and their specification not publicly available. We can, however, provide the following argument: Let us assume a very simple kernel with a 5×5 AoS consisting of two fields, a memory access as shown in Figure 5, a device with four hardware threads, and a memory controller that can fetch one data line with four adjoining items

Fig. 6. **Left:** The measured performance (black) of 5,184 configurations sorted from best to worst, our predicted performance (green) based on measured support (blue cross), and base (red cross) configurations. **Right:** A closer look at the section of the best configurations (indicated by the black box). It can be seen that the prediction can be noisy and can slightly deviate from the exact results.

at once. For our theoretical system, each data line load requires 10 clock cycles and one additional clock cycle for reading data lines that are not adjointed. Further, let us assume that the data can only be represented as AoS or SoA, and can be stored untransposed or transposed. This results in a total of four configurations. Figure 5 shows the memory access for all configurations in the first iteration of the inner loop. The major goal for us is to minimize the required clock cycles for all memory loads. As can be seen, SoA^T is the best layout with only two lines to be read. To find the optimal layout, we have to sample AoS as base configuration. We also require two support configurations, which are AoS^T and SoA. Their measured execution time is $t(\text{AoS}) = 54$, $t(\text{AoS}^T) = 30$, and $t(\text{SoA}) = 51$ clock cycles for the inner loop. When we apply our predictor (Equation (1)) we get the predicted execution time $p(\text{SoA}^T) = 27$, which is the best result. Although this value differs from the exact value $t(\text{SoA}^T) = 20$, it is still a useful prediction. In fact, the difference is caused by the choice of parameters for our artificial example. However, in a real system we would expect a deviation anyway, caused by noise of the measurements. Figure 6 shows a real example for the prediction of a kernel, where we require three base and 36 support configurations to estimate the performance of 5,145 non-profiled configurations, which is less than 1% of the total configuration count. As can be seen, the prediction is not perfect but it is good enough to select a configuration, which is pretty close to the optimal solution.

4.3 Step 2: Determine Optimal Configurations

Given the prediction algorithm, we can calculate the optimal configuration for each kernel execution. Now we have to determine an application-wide optimal solution. This is difficult, as arrays that are used in multiple kernels introduce dependency constraints that need to be resolved. In general, there are two ways to handle these. Either data can be converted between two kernel executions, or not. Without knowing how long a kernel will execute using unknown data, we have no means to predict at runtime whether the conversion will yield enough improvement to compensate the conversion, so we decided not to allow any conversions once a layout has been determined. However, we plan to investigate this further in the future.

4.3.1 Decisions. As data is allocated at various points during the execution, we may have to apply some data layouts already prior to any kernel executions, e.g., host arrays are usually filled with data before they are copied to the device. We differentiate between three *decision events* that require action:

- (1) whenever a host array is allocated,
- (2) whenever a device array is used in a kernel (if it has not inherited a layout through a prior memcopy), and

Fig. 7. DDG example with global decisions of Type 1 (gray), Type 2 (green), and kernel decisions of (blue, Type 3). Global decisions are shown by their unique decision identifier and their unique allocation id. The array corresponding to $C\#X$ is reallocated inside the loop in every iteration.

- (3) whenever a kernel is executed, non-device memory decisions (e.g., shared/local memory, defines, ranges, ...) can be determined without any constraints.

There is one special case: If a device array was marked as *write-only*, the kernel will overwrite the entire content of the array and therefore is not constrained by the existing layout. In this case, MATOG can assign a new layout resulting in a new decision (Type 2). As an allocation can occur multiple times during the execution (e.g., in a loop), every array gets assigned a unique id. This allows one also to use the size of previously allocated arrays as metadata during the decision-making process.

4.3.2 Array Dependencies. To resolve the array dependencies, we previously used an Array Dependency Graph (ADG) (Weber and Goesele 2016). An ADG is a directed graph that contains nodes for each array allocation, kernel execution, and memcpy between host and device. This graph maps onto the lifecycle of arrays during the application execution. While the ADG representation fits exactly the flow of the application, it is difficult to parse. We propose here a simplified version, called Decision Dependency Graph (DDG). This is also a directed graph, but instead of mapping to the lifecycle of arrays, it maps onto the decisions made during the execution. There are only two types of nodes in this tree: *global* and *kernel* decisions. Global decisions occur whenever the layout of a global memory array needs to be determined (Types 1 and 2). These nodes are connected to kernel decisions (Type 3) by edges. Each edge is labeled by the global decision it originates from and connects all kernels in sequence it is used in. Figure 7 shows an artificial example of a DDG with all possible cases. Depending on the structure of the application, the graph does not need to be fully connected and can be a forest of multiple graphs. This also happens when the application has been profiled with multiple different datasets, where each of these profiling results is a disjunct graph. In the following, we describe our method on one graph, but this can be easily extended to a forest of graphs.

4.3.3 Exhaustive Search. In Weber and Goesele (2016), we proposed an exhaustive analysis of the DDG. This method guarantees one to find the optimum of the application, but can be very time-consuming for complex applications with a high number of reallocations. The exhaustive search varies all possible combinations of the DDG and calculates the total execution time for all kernel executions in it using exhaustive profiling data or our predicted execution times. If predicted data is used, the execution can be sped up by splitting all decisions into two categories: *local* and *global*. Local decisions influence only one kernel execution, e.g., shared/local memory layouts, usage of non-coherent cache, constant memory, ranges, or defines. Global decisions are all global memory struct layouts and transpositions. This separation allows one to precompute optimal values for the local decisions, for each kernel, as we can store one optimal local value for each possible global decision permutation.

4.3.4 Predictive Search. The problem with an exhaustive search is that if an application allocates or reallocates many arrays (e.g., in every iteration of a loop), the number of global decision nodes grows rapidly. As the number of combinations that have to be evaluated thus increases exponentially, this method becomes quickly unfeasible. Unfortunately, this often occurs in real applications. We therefore extended our predictive profiling onto the search for an application-wide optimal solution. The assumption of the prediction algorithm implies that we can select optimal values without respecting any interactions between the optimization dimensions. In the first step, we iterate over all global decisions separately and select optimal values for these. We ignore the optimization domains in this step and only search inside one domain of the shared degrees. This is possible, as we have previously mentioned, layouts and transpositions (as employed by global decisions) hardly change the amount of used resources. In a second step, we iterate over all local decisions and search for the best values, but now we obey the optimization domains. We will show in our experiments that this method provides comparable results to an exhaustive search.

4.4 Step 3: Decision Models

At this point, we know the optimal solutions for all profiled application runs. Now we have to use this information to build decision models that can select optimal configurations during runtime. For these decisions, we require some kind of metadata to distinguish between different input data scenarios. Auto-tuners such as Chung and Hollingsworth (2004) and Muralidharan et al. (2014) use user-defined callback functions that calculate some arbitrary, user-determined metrics for this distinction. We instead use a fully automatic way. As analyzing and categorizing arbitrary data is a very difficult, often time-consuming task (depending on which analyses are performed), we chose to use the metadata that is already available in our system such as the sizes of arrays and the launch configurations of kernels. In contrast to our previous publications, our improved metadata gathering system not only tracks the most recent arrays in the system, but also previous data, using unique ids for each allocated array. We additionally allow the user to register variables, which are monitored during the execution. These can contain data that has been explicitly calculated by the users, or e.g., counts of items in a preallocated, not-resized array. This information is usually available in the application without any need to specifically calculate these. For our decision model creation, we gather all decision events from all DDGs and group them by their decision event. We create one model per decision event. For this we collect all metadata that was present in the profiling when the decision event occurred and store it in a matrix, in which each column represents one kind of metadata (e.g., size of a specific array or the launch configuration of a kernel) and each row represents one decision event. At this point it does not matter if it is an array allocation or kernel execution decision. This gathered metadata usually contains a high amount of redundant, linear-dependent or constant data. To simplify the decision model creation and to reduce the metadata gathering overhead during runtime, we preprocess this data by removing all redundant data. If multiple rows with equal metadata but different optimal configurations occur, we perform a majority voting and keep the configuration which was chosen most.

4.4.1 Directional Model. In Weber and Goesele (2016) we used a SVM as decision model. This proved to be suitable but may not be optimal (depending on the application). This is rooted in the kind of metadata that we use in MATOG, consisting of array sizes and launch configurations. These come with some difficulties. For example, as we do not know the limits of our gathered metadata, there is no way to normalize it, which can result in bad decisions if the values differ too much from the training data. Further, as array sizes and launch configurations are usually used as iteration counts for loops, they imply a certain linear scaling, which can be leveraged in a decision model. We therefore experimented with other kinds of decision methods and found that in some

Fig. 8. Comparison between a linear SVM (top) and the directional model (bottom). On the left, the data used for training and the resulting models are shown. On the right, the models are applied to the testing datasets and false decisions are highlighted with a red cross. Be advised that the meta dimensions are hand picked and that the actual model does not only decide whether to use local or shared memory, but also decides on layouts, transpositions, and texture memory usage.

of the failure cases that we had encountered, a specialized nearest-neighbor search model worked better; in the following called Directional Model (DM). The DM uses the cosine similarity distance and interprets the training ($\vec{t}$) and meta ($\vec{m}$) data as vectors. It then applies the normalized dot product on these vectors in order to compute the metric: $\lambda = \cos(\gamma) = \vec{t} \cdot \vec{m} / |\vec{t}| |\vec{m}|$. The resulting λ is in the range $[-1; 1]$. To evaluate the model, we iterate over all training samples, calculate λ , and select the data point where λ is maximal. The advantage of this method is that it performs an implicit normalization, resulting in better decisions even if the meta and training data values significantly differ.

Figure 8 shows a simple example for a linear SVM compared to the DM, both trained and evaluated on data taken from our KD-Tree benchmark, run on a Tesla K20c. Every dot represents the metadata of a kernel execution and colors indicate if either using *local* (orange) or *shared memory* (blue) is the better choice. In the upper part, we show the results for a linear SVM, with the training (left) and testing (right) data. The support vectors and false decisions are highlighted by black circles and red crosses, respectively. The hyperplane of the SVM is shown as a black line and the colored background visualizes the classification of the SVM. In the lower part, we show the same diagrams for the DM. The decision vectors are shown as arrows, with the color of the corresponding decision. As can be seen, the number of false decisions is significantly lower for the DM than for the linear SVM.

5 MATOG RUNTIME SYSTEM

In this section, we briefly explain our runtime system. At the beginning of the application run, MATOG is automatically initialized together with CUDA. Whenever an array is allocated, we

determine which decision event it belongs to and store its metadata in a centralized metadata storage. If an array of the same decision event is allocated, it gets a new instance id assigned. If this array is a host array, its decision model is evaluated immediately and the layout is set (Type 1). Whenever a memcpy occurs, MATOG propagates the layout from the source to the destination array. If a kernel is executed, all non-initialized device arrays evaluate their decision model and set their layouts (Type 2). Further, the kernel evaluates its own decision model (Type 3). The final executed configuration of the kernel is the combination of all layouts of the device arrays (which have been separately evaluated before) and the configuration from the kernel's decision model. If this configuration has been compiled before, it is loaded from the database, and then executed. If not, it will be compiled and then executed. In Weber and Goesele (2016), we compiled all available combinations that could occur after our application analysis process; however, depending on the complexity of the application this can be several million configurations, making this approach not usable. We discuss in our future work how this high number of configurations could be reduced to improve the compilation time, but also reduce the number of implementation switches during runtime.

6 EVALUATION

In this section, we evaluate the different analysis steps of MATOG, their efficiency, and the achieved performance of MATOG on six applications. All tests have been performed on dual Intel Xeon E5649, 48GB RAM, Ubuntu 16.04 and CUDA 8.0 (driver version 367.57). We evaluated the last four GPU architectures on low-end, high-end, and HPC GPUs: Fermi (GT 440, GTX 480, GTX 570), Kepler (GTX 680, GT 730, K20c), Maxwell (GTX 980), and Pascal (GTX 1080). The results are compared to an unmodified but optimized reference code and our previous version of MATOG (Weber and Goesele 2016). Changes to the auto-tuner in our new version include the following:

- Support of hierarchical data structures (e.g., `array[x][y].subarray[z].substruct.field[w]`).
- Support for Fermi GPUs.
- Support to store data in constant memory (if size is constant and known at compile time).
- New “read once” hint, allowing one to use texture memory to read non-read-only data once. This is helpful, if data is once read at the beginning of the kernel and then overwritten but never read again in the same kernel call.
- Improved metadata system (as described in Section 4.4).

Providing a fair and direct comparison to related approaches on our benchmarks is difficult. On one hand, there is, to our knowledge, no publicly available code for several other memory access auto-tuners (Sung et al. 2012; Kofler et al. 2015; Peng et al. 2016). On the other hand, auto-tuners such as Nitro (Muralidharan et al. 2014) or OpenTuner (Ansel et al. 2014) do provide code but miss specific, automatically generated optimizations. Instead, they require all optimizations considered to be explicitly hand-coded by the user, a stark contrast to MATOG's automated approach. We can, however, make indirect but meaningful comparisons: Nitro is designed to handle only a small number of configurations and thus relies on exhaustive search. Since we provide both timing as well as performance evaluations for exhaustive search as part of our results below, these can be seen as representative for Nitro. OpenTuner uses a genetic algorithm. In Weber et al. (2015), we showed that our predictive-based search uses a minimalistic set of configurations that suffices to find comparable results and is faster for the optimization problem that we face in MATOG, compared to a genetic algorithm. This performance argument is still valid for the current version of MATOG and thus provides a clear advantage.

Table 1. Theoretical Number of Configurations (T , Entire Solution Space), the Number of Configurations an Exhaustive Search has to Execute (E), and the Number of Configurations Required for Our Method (P) for All Evaluated GPUs and Benchmarks. The Exhaustive Search (E) contains Less Configurations as the Theoretical (T), as the Texture Memory can Only be used if the Data is Read-Only

	Bitonic			SRAD			Hotspot			COMIC			REYES			KD-Tree		
	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P
GT 440/ GTX 480 / 570	48	42	18	768k	147k	62	1024	512	18	3112	780	54	27.2M	13.6M	158	1.3B	404M	200
GTX 680 / GT 730 / Tesla K20c	72	63	27	1.2M	221k	93	1536	768	27	4668	2808	60	40.8M	20.4M	237	1.9B	605M	300
GTX 980 / 1080	24	21	9	393k	74k	31	512	256	9	1872	936	20	13.6M	6.8M	79	645M	202M	100

6.1 Benchmark Applications

We evaluate six different GPU applications, ranging from very simple algorithms with regular workload up to very irregular algorithms with varying workload. Table 1 shows the number of possible configurations per GPU and benchmark. Please refer to our project page² for details on our training and test datasets.

Bitonic Sort (Batcher 1968) is a widely used parallel sorting algorithm. In our implementation, it sorts a 1D AoS with four integer fields (8, 4, 2, and 1 B), ensuring that the 8B value is sorted first and only if this value is equal, the 4B 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 1,023 (255 for the 1B field). The application consists of two kernels: One uses shared memory in loop iterations where it can be used efficiently, while the other directly operates on global memory. The reference implementation is not optimized and uses a naïvely AoS layout. We train on two datasets and evaluate on five others, all with varying element counts (64k to 4M).

Speckle Reducing Anisotropic Diffusion (SRAD) (Che et al. 2009) is a diffusion method for ultrasonic and radar imaging applications. The benchmark’s computations are quite simple and use a straightforward implementation without much program logic or dynamic allocations. We train on three parameter sets with the same grid size and one iteration, and evaluate 12 other parameter sets with varying grid sizes and 100 iterations.

Hotspot (Che et al. 2009) calculates the processor temperature based on an architectural floor plan and simulated power measurements. The overall execution time of the application is quite low. The benchmark comes with three datasets, so we train on the medium-sized one and evaluate on the other two.

Coevolution via MI on CUDA (COMIC) (Waechter et al. 2012) 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. The original implementation was optimized for a GTX 480. We train on two datasets with one iteration and evaluate on 11 others using 100 iterations.

Renders Everything You Ever Saw (REYES) (Cook et al. 1987) is a technique used in movie productions. In contrast to classic 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 micro polygons after each iteration. A third kernel uses the final polygons and renders the resulting image into a depth buffer which contains depth and

²matog.org.

color information for each pixel. Finally, a fourth kernel extracts the color information from this buffer into a 2D texture which then can be displayed. The benchmark was originally developed for a GTX 480. As this benchmark has a very high number of configurations, we do not show any exhaustive profiling results as it would require several weeks to profile the application. We train on one model, rendering two frames on two different resolutions and evaluate on eight models, 100 frames, and varying resolutions. The models are rotated after each frame to change the workload, as depending on the viewing angle, more or less polygons have to be split and rendered.

The **KD-Tree** benchmark constructs acceleration structures for 3D ray tracing and resembles the work of Popov et al. (2006). The application consists of two main and six maintenance kernels, which have a low total execution time. The first main kernel discretizes a triangulated scene in multiple bins which are separated by equidistant planes in all three dimensions. It then calculates a building heuristic to determine the best split plane. The second main kernel performs the actual splitting of a subtree and stores all necessary data in two different data segments, one for each child tree. All kernels are built in a fashion that they can process multiple subtrees in parallel. More information on this benchmark can be found in Weber and Goesele (2016). The difficulty of this benchmark is the changing data characteristics over time. At the beginning, only very few but very big subtrees are processed. While the number of subtrees significantly increases during the processing, the size of these is reduced and the number of leaf subtrees constantly increases. In the end many, very small subtrees are processed. This change has a significant impact on the performance and should benefit from MATOG's adaptiveness. This benchmark was optimized for a GTX 590. Due to the extreme high number of configurations, we do not show any exhaustive profiling results. Further, as each iteration memory is reallocated, our implementation of the exhaustive post-processing cannot be used, as the possible combinations exceed 2^{64} . We train on one medium-sized 3D scene and evaluate on nine others. As some of these are very big (up to 12M triangles), not all GPUs are able to process these because of memory size limitations.

6.2 Execution Performance

In this section, we evaluate the achieved execution performance of MATOG against an unmodified reference code and our previous version (Weber and Goesele 2016). All executions have been repeated five times (except for the application profiling runs), to reduce the influence of measurement noise. All evaluations have been performed on separate training and evaluation datasets.

6.2.1 GPU Execution Time. First, we take a look at the achieved speedup for the kernel execution time. We have evaluated three different decision model types: SVM, DM, and a non-adaptive model, that always selects the single configuration that was chosen most, to verify if adaptive decisions are necessary to achieve better performance. Further, we evaluated three analysis strategies: exhaustive profiling/exhaustive analysis, predictive profiling/exhaustive search (method used in Weber and Goesele (2016)), and predictive profiling/predictive analysis (our new method). As our old auto-tuner does not support Fermi GPUs, no results are available for the GT 440 and GTX 480 and 570. The results are shown in Figure 9. As mentioned before, Bitonic Sort, SRAD, and Hotspot are simple algorithms without much divergence and a very regular processing. SRAD and Hotspot are already optimally optimized in terms of memory access. MATOG is slightly slower than the reference code, as MATOG data structures employ a certain overhead compared to hard-coded memory access. One reason for this is pointer aliasing, as the way we have implemented the MATOG data structures does not allow one to use the *restrict* keyword to define the internal pointers as non-overlapping. Therefore, the compiler is unable to perform certain optimizations, which can slightly decrease the performance. In most cases, no real difference between the different analysis methods and decision models can be seen, implying that these are invariant to adaptive

Fig. 9. Measured GPU execution time on the evaluation datasets. Our previous version (Weber and Goesele 2016) is indicated by the *O*. The naming scheme for the others starts with the used profiler (exhaustive (*E*) or predictive (*P*)), analysis method (exhaustive (*E*) or predictive (*P*)), and decision model (SVM (*S*), DM (*D*) or non-adaptive (*N*)). Our previous method with our new auto-tuner (PES) and our proposed method (PPD) are highlighted. Results that differ too much from the other results are shown as numeric values.

optimizations. For SRAD, the exhaustive profiling usually finds slightly better solutions than our predictive profiling, but has to profile over $2,371\times$ more configurations to achieve this result. Our new auto-tuner achieves significantly better performance in SRAD and Hotspot compared to our previous version. We are unsure why this happens and assume that internal improvements of the data structures are responsible for this. On the GT 440 and Tesla K20c, MATOG is even able to achieve higher performance than the hand-optimized implementations. The irregularities for the

Fig. 10. CPU, GPU, and total speedup compared between the reference code and our proposed method. For Bitonic and Hotspot, we can see that the overhead introduced into the MATOG system is not too high. Through the extremely low execution time of SRAD, the setup overhead of MATOG decreases the CPU performance. COMIC, however, benefits on the CPU and GPU side from the MATOG data structures. For REYES and KD-Tree, the optimized layouts for the GPU somewhat decrease the performance of the CPU, as they perform less optimally. However, for the three complex benchmarks and the Bitonic Sort we mostly see an improved overall application performance. SRAD's performance is decreased and Hotspot is neutral.

results of the GT 730 on Hotspot are difficult to explain. We think this is caused by the heat of the chip, as this GPU is passive cooled and therefore can only regulate its temperature by reducing the clock frequency. So far, all benchmarks have been simple algorithms, consisting of only one to two kernels with a very low execution time, that do not benefit from adaptiveness during runtime. The three remaining applications are much more complex, consisting of three to eight kernels. For COMIC we can see that the reference code is optimal for the GTX 480 and 570, which it had been developed for. On the newer cards, however, MATOG is able to leverage up to 34% more performance. MATOG chooses no texture memory on all GPUs prior the GT 730. The main speedup is achieved through transposition of the data stored in shared memory. Local memory is never used. This benchmark also does not benefit from adaptiveness. Although REYES has been optimized for the GTX 480, MATOG is able to slightly improve the performance on this GPU. The highest speedup of 37% is achieved on the Tesla K20c. As can be seen, our adaptive methods achieve up to 7% higher performance. The KD-Tree is by far the most complex and difficult to optimize applications we evaluate on. It has a lot of changing data properties, a huge number of reallocations, and an irregular work processing. The non-adaptive model achieves decent speedups, but the DM always achieves the best and is up to 32% faster than the non-adaptive solution. The SVM always performs less optimally than the DM, especially on the GT 730. The SVM is here unable to understand the underlying coherence of the metadata since the values differ too much from the training data, as it cannot be normalized. This causes the SVM to use local memory very often, whereas this is only optimal in the very first iterations with few big subtrees and performs very badly in all subsequent iterations. To summarize, we can say that MATOG achieved the highest speedups on the Tesla K20c. There is no real difference between the analysis methods (in terms of quality), except for the SRAD benchmark, where the exhaustive profiling often is able to find better configurations. Further, the DM has achieved the highest performance (compared to the other methods). In the very complex applications, our adaptive optimizations outperform the static optimizations by up to 27%.

6.2.2 Application Execution Time. Next, we take a look at the application execution time, containing not only the GPU but also the CPU computation time. In Figure 10, we show the speedup of CPU, GPU, and the total application speedup separated, in relation to the reference code. Except for COMIC and some cases of the Bitonic Sort, all benchmarks push the execution more toward

Fig. 11. Percentage of CPU time for the reference and our method. As can be seen, the three simple benchmarks have a significantly higher portion of CPU time, therefore solely optimizing the GPU does not necessarily benefit the application. For the other benchmarks, the CPU time is much lower. Further can be seen that newer/faster GPUs push the ratio more to the CPU.

Fig. 12. GPU (left) and application (right) efficiency for CUDA and our proposed method. Except for SRAD and Hotspot, MATOG is superior to CUDA for both metrics.

the CPU. This can be explained by the fact that the MATOG runtime system produces a certain overhead through an additional setup time, metadata tracking, decision making, exchanging of different GPU codes, and the MATOG host arrays, as they use a dynamic data layout which is determined by runtime that does not perform as fast as a static hard-coded memory access. Further, data layouts that perform optimally on GPUs do not necessarily perform well on CPUs. For the GPU, we have already seen the achieved speedups, which are mostly positive. By looking at the total execution speedup, we see that except for SRAD, all benchmarks are faster, or at least as fast as the reference implementation. Figure 11 shows that the simple applications (Bitonic Sort, SRAD, and Hotspot) spend most of their execution time in CPU code. The up to 80% GPU speedup for the Bitonic Sort does, therefore, yield only a small total speedup. Pennycok et al. (2016) proposed a method to measure performance portability for applications across multiple platforms. They calculate the harmonic mean over the execution time for one approach over a series of executed applications and normalize it with the harmonic mean of the best results for the applications. Figure 12 shows the results for CUDA and our proposed method, compared to all other methods we have analyzed. Except for SRAD and Hotspot, our method is superior to the pure CUDA implementation for the GPU and application efficiency. Overall CUDA achieves an average efficiency of 87.06% GPU and 94.79% application efficiency, while our proposed method achieves 98.65% and 97.03%, respectively. Assuming that an exhaustive search would yield 100.0% GPU efficiency, our method comes pretty close, whereas it requires significantly less time to achieve its results, as we show in the next section.

6.3 Analysis Time

Finally, we take a look at the time required for the analysis. Figure 13 shows the measured times for the profiling and analysis of the benchmarks on the GTX 1080. As stated before, REYES and

Fig. 13. Shown results are for the GTX 1080. (Be aware of the log scale!) **Left:** Execution time for the profiling. It is easy to see that the predictive profiling is significantly faster than the exhaustive profiling and can profile all applications within a few minutes. For reference, the execution time required to run the application with the training datasets but without profiling is also provided. As can be seen, for SRAD the exhaustive search requires more than 12.000× more time for the profiling, compared to the application runtime, whereas our method only requires 4.5×. **Right:** Time required for the analysis of the profiling data. The analysis is quite fast for all methods (compared to the profiling); however, our proposed method (green) is always the fastest. Further, it can easily be seen that with increasing complexity the execution time rises. The exhaustive analysis requires more time using exhaustive data (blue) than with predictive data (orange) caused by much more data that needs to be loaded and processed.

KD-Tree require too much time for the exhaustive profiling and have therefore been excluded. Except for the Bitonic Sort, our predictive profiling is always significantly faster than the exhaustive search. The reason for this is that the Bitonic Sort only has very few possible configurations, so that the number of configurations hardly differ.

7 DISCUSSION

Summarizing our results, we can see that it is essential to adapt the code to the specific GPU architecture, as code written for older cards does not necessarily perform well on newer hardware, especially for complex applications with irregular or changing workload. Depending on the complexity of the application, static optimizations (Bitonic Sort, SRAD, Hotspot, COMIC) may suffice. Anyway, these optimal parameters have to be found in the first place. In other cases (REYES and KD-Tree), dynamic optimizations achieve a much higher performance. Further, we can see that optimal memory access is crucial in many applications and can significantly influence the performance of the code. However, solely optimizing the GPU code does not necessarily improve the overall performance as can be seen in the Bitonic Sort benchmark, where the 80% speedup of the GPU code is nearly entirely consumed by an increase of the CPU execution, as the optimal layouts for GPUs can be suboptimal for CPUs.

7.1 Analysis of Optimal Configurations

In Figure 14, we show how often which layout, cache size, transposition, or memory has been used by our method. The results differ not only between the different benchmarks significantly, but also between the GPUs even within the same architecture. There is no clear tendency toward a specific struct layout, transposition, L1 cache size, or the usage of texture memory. An analysis of the resource usage of the code and the SASS assembly code showed, however, some cases that could also easily be optimized by a programmer.

7.1.1 Simple Memory Bound Benchmark. The Bitonic benchmark mainly uses SoA since all threads access adjoining indices. In some cases, AoSoA is equally good because all tiles are stored

Fig. 14. Percentage of how often which layout, cache size, transposition, or memory was chosen to be optimal by our method on the testing datasets. As can be seen, the results significantly differ between the benchmarks but also between the GPUs, even within the same architecture.

in a SoA-manner. However, AoSoA requires more calculations to determine the correct memory address (`soa.field[index]` vs. `aosoa[index/32].field[index%32]`). The kernels of the Bitonic benchmark are memory bound and far from the resource border where the occupancy could drop because of a higher register usage, so that additional computations are no limiting factor for the performance. SoA and AoSoA therefore perform similarly, and the variation in our results is probably mainly caused by measurement noise.

7.1.2 Complex Compute Bound Benchmark. For the more complex benchmarks, e.g., the third kernel of the REYES benchmark, it is more difficult to explain all effects. In contrast to the Bitonic kernel, it is compute bound and much more complex, as it uses multiple multi-dimensional arrays with struct types in global and shared memory. Although it is compute bound, the worst configurations we found are 23% slower on a GTX 980 than the best. We have mainly two observations for this kernel. First, on Fermi GPUs an AoS layout works optimally for the data stored in shared memory, while on newer Maxwell or Pascal cards, SoA is consistently used. As the way in which shared memory is supposed to be used (according to the programming guide (NVIDIA 2016)) has not changed since Fermi, we did not expect this result. Second, on the Tesla K20 the micro polygons are stored as SoA in device memory, while they are stored as AoS on the GTX 980. The projection matrices (in constant memory) and micro polygons (in shared memory) are stored transposed on the Tesla K20, while on the GTX 980 they are untransposed. Analyzing the SASS assembler of both configurations, we realized that the code for the GTX 980 is longer and issues more instructions per instruction block than the code optimized for the Tesla K20. The results of the NVIDIA

profiler confirm this, as the code executes 2.4% more integer instructions, but achieves also 1.1% more instructions per cycle. On the Tesla K20, it is behaving differently as the optimized code for the Tesla K20 issues 1.8% less integer instructions than the code optimized for the GTX 980, and achieves 0.4% higher instruction per cycle. It seems almost as if the different memory layouts trigger the compiler to perform very different code optimizations which have a strong implication on the achieved performance for compute bound kernels.

This example shows the need for auto-tuners such as MATOG, as solely concentrating on one optimization goal (e.g., optimal memory access) does not necessarily lead to maximal performance. MATOG is designed to optimize memory access, but through the usage of empirical profiling it implicitly optimizes resource usage and computations.

7.1.3 Unintuitive Optimizations. Another surprising result is that the optimal size of the L1 cache size seems to often be unpredictable. Reducing the amount of shared memory can reduce the occupancy, which can in turn have positive or negative effects on the execution time (Volkov 2010). Further, since the Kepler architecture, the L1 cache is supposed to only serve local memory, so an intuitive way would be to use more L1 cache in kernels which heavily rely on local memory and use only small amounts of shared memory. However, in some of these cases this approach seems to work, but in others different settings work better.

Overall, the usage of local memory instead of shared memory only proves to be beneficial in some rare cases. REYES is the only benchmark where certain array sizes are known at compile time. This benchmark greatly benefits from using constant memory, although the usage of texture, global, and constant memory varies significantly between the different architectures.

Furthermore, when to use texture memory is difficult to answer. Our results indicate that a certain balance between using texture and global memory has to be found to achieve maximal performance. What we can see is the obvious fact, that using texture memory only benefits arrays that are read more than once. However, not using texture memory in this case allows other data to be cached there instead. To use texture memory always, e.g., through blindly defining all read-only arrays as `const __restrict__ type*` on GPUs with CC ≥ 3.5 , should be avoided.

7.2 Conclusion

In this article, we have provided a unified view of the MATOG components published in the previous conference publications (Weber et al. 2014, 2015, 2016) as well as an improved application analysis algorithm and runtime decision-making system. Our auto-tuner is capable of analyzing applications with billions of possible configurations within a few minutes while achieving comparable results to an exhaustive analysis. MATOG's ability to adjust itself during runtime to react to changing data properties allows one to achieve higher performance than a static hand-optimized implementation. We further showed the flexibility of MATOG to adjust itself to low-end (GT 440 and 730), high-end (GTX 570, 680, 980, and 1080), and HPC GPUs (Tesla K20c).

However, our results also show that there is still room for improvements leading to several different directions that we want to pursue in future: First, MATOG does not take CPU times into consideration. As every GPU application is a heterogeneous device application, this does not guarantee overall optimal performance. In the future, we therefore plan to extend MATOG to different device types. Second, we want to explore more optimizations, as e.g., an explicit cache bypassing (Li et al. 2015a), an automated selection of optimal sparse matrix formats (Muralidharan et al. 2014), or an automatic separation of AoS fields to partially be stored in a AoS or SoA (Peng et al. 2016). Third, we plan to integrate MATOG into an optimizing compiler (e.g., OpenARC (Lee and Vetter 2014)) which then will integrate MATOG automatically into existing applications. Fourth, one of the current shortcomings is the very high number of nearly optimal configurations. This

can lead to many loads/swaps of code modules during runtime. As usually multiple configurations are grouped (in terms of optimal performance, as can be seen in Figure 6), it should be possible to cluster these in order to reduce the number of configurations. Finally, we want to investigate generating automated performance models during runtime to evaluate whether converting an array would yield a speedup that would make a conversion feasible. Also methods that constantly improve the decision models during normal runtime are promising, as this would remove the need to explicitly perform a profiling run. Instead, it could constantly monitor the applications performance and improve the decision making.

MATOG determines only an optimal layout but does not give any insight into why a specific layout is optimal since this would require an analysis of the actual executed instructions, i.e., an analysis of the SASS assembly code. For this article, we have conducted this analysis manually. To automate this process, techniques as presented by Zhang et al. (2017) could be used. They have recently proposed a method to automatically reverse engineer SASS, which is not publicly documented. Our analysis of the third REYES kernel showed that this is likely difficult as the code of the different configurations changes not only memory access but also is optimized in a completely different way, making it hard to directly compare the different versions. This is, however, less problematic than it appears at first sight as our results show that optimal performance can still be achieved by profiling actual executions without performing a low-level analysis of the underlying assembly code.

REFERENCES

- Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proc. PACT*.
- K. E. Batchier. 1968. Sorting networks and their applications. In *Proc. SJCC*.
- Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory access patterns: The missing piece of the multi-GPU puzzle. In *Proc. SC*.
- James Bergstra, Nicolas Pinto, and David Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *Proc. INPAR*.
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IISWC*.
- Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. PPOPP*.
- I-Hsin Chung and Jeffrey K. Hollingsworth. 2004. Using information from prior runs to improve automated tuning systems. In *Proc. SC*.
- Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. In *Proc. SIGGRAPH*.
- H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *Proc. XSW*.
- Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2009. Loop transformation recipes for code generation and auto-tuning. In *Proc. LCPC*.
- Tianyi David Han and Tarek S. Abdelrahman. 2009. HiCUDA: A high-level directive-based language for GPU programming. In *Proc. GPGPU*.
- Chia-Chen Hsu, Cheng-Yen Lin, Shin Kai Chen, Chih-Wei Liu, and Jenq-Kuen Lee. 2014. Optimized memory access support for data layout conversion on heterogeneous multi-core systems. In *Proc. ESTIMedia*.
- Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. 2015. Automatic data layout optimization for GPUs. In *Proc. Euro-Par*.
- Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proc. HPDC*.
- Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015a. Adaptive and transparent cache bypassing for GPUs. In *Proc. SC*.
- Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. 2015b. Automatic data placement into GPU on-chip memory resources. In *Proc. CGO*.
- Yixun Liu, E. Z. Zhang, and Xipeng Shen. 2008. A cross-input adaptive framework for GPU program optimizations. In *Proc. IPDPS*.

- Thibaut Lutz, Christian Fensch, and Murray Cole. 2013. PARTANS: An autotuning framework for stencil computation on multi-GPU systems. *ACM TACO* 9, 4 (Jan. 2013), Article 59.
- Thibaut Lutz, Christian Fensch, and Murray Cole. 2015. Helium: A transparent inter-kernel optimizer for OpenCL. In *Proc. GPGPU*.
- Alberto Magni, Christophe Dubach, and Michael O’Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proc. PACT*.
- Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. 2011. GROPHECY: GPU performance projection from CPU code skeletons. In *Proc. SC*.
- Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proc. HiPEAC*.
- Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. 2014. Nitro: A framework for adaptive code variant tuning. In *Proc. IPDPS*.
- Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. In *Proc. EMMSOC*.
- NVIDIA. 2014. NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210 V1.1. <http://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>.
- NVIDIA. 2016. CUDA Programming Guide v8.0. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- Yuhan Peng, Max Grossman, and Vivek Sarkar. 2016. Static cost estimation for data layout selection on GPUs. In *Proc. PMBS*.
- S. J. Pennycook, J. D. Sewall, and V. W. Lee. 2016. A metric for performance portability. In *Proc. PMBS*.
- Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. 2006. Experiences with streaming construction of SAH KD-trees. In *Proc. IEEE IRT*.
- Hans Henrik Brandenborg Sørensen. 2012. Auto-tuning dense vector and matrix-vector operations for fermi GPUs. In *Proc. PPAM*.
- I.-Jui Sung, G. D. Liu, and W. M. W. Hwu. 2012. DL: A data layout transformation system for heterogeneous computing. In *Proc. INPAR*.
- Vasily Volkov. 2010. Better Performance at Lower Occupancy. In *GTC2010*.
- Michael Waechter, Kathrin Jaeger, Stephanie Weissgraeber, Sven Widmer, Michael Goesele, and Kay Hamacher. 2012. Information-theoretic analysis of molecular (Co)evolution using graphics processing units. In *Proc. ECMLS*.
- Nicolas Weber, Sandra C. Amend, and Michael Goesele. 2015. Guided profiling for auto-tuning array layouts on GPUs. In *Proc. PMBS*.
- Nicolas Weber and Michael Goesele. 2014. Auto-tuning complex array layouts on GPUs. In *Proc. EGPGV*.
- Nicolas Weber and Michael Goesele. 2016. Adaptive GPU array layout auto-tuning. In *Proc. SEM4HPC*.
- Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In *Proc. PPOPP*.

Received December 2016; revised May 2017; accepted June 2017