Guided Profiling for Auto-Tuning Array Layouts on GPUs

Nicolas Weber
TU Darmstadt
Graduate School of
Computational Engineering

Sandra C. Amend
TU Darmstadt

Michael Goesele
TU Darmstadt
Graduate School of
Computational Engineering

ABSTRACT

Auto-tuning for *Graphics Processing Units* (**GPUs**) has become very popular in recent years. It removes the necessity to hand-tune GPU code especially when a new hardware architecture is released. Our auto-tuner optimizes memory access patterns. This is a key aspect to exploit the full performance of modern GPUs. As the memory hierarchy has historically changed in nearly every GPU generation, it was necessary to reoptimize the code for all of these new architectures. Unfortunately, the solution space for memory optimizations in large applications can easily reach millions of configurations for a single kernel. This vast number of implementations cannot be fully evaluated in a feasible time. In this paper we present an adaptive profiling algorithm that aims at finding a near optimal configuration within a fraction of the global optimum, while reducing the profiling time by several orders of magnitude compared to an exhaustive search. Our algorithm is aimed at and evaluated on large real-world applications.

1. INTRODUCTION

Leveraging the full potential of a *Graphics Processing Unit* (**GPU**) is a difficult task, as there are many influencing factors such as the specific algorithm, used scheduling, resource utilization, hardware architecture, and many more. In the last couple of years, several auto-tuning approaches targeting different optimization aspects have emerged, including application [10, 26] and domain specific solutions [12, 16, 22, 28], high level languages [1, 4, 8, 9, 11] and frameworks or tool sets, which optimize applications domain independently [13, 25, 27].

Optimal memory access is arguably one of the most crucial performance aspects, especially since the memory hierarchy has been altered in nearly every GPU generation, which consequently required to adapt existing code for each new architecture. Our auto-tuner optimizes the array access in NVIDIA CUDA [18] applications. It is able to choose different transpositions and Array of Struct layouts (such as *Array of Structs* (**AoS**), *Structure of Arrays* (**SoA**) or *Array of Structure of Arrays* (**AoSoA**, often also referenced as tiled-AoS)), as well as to optimize the usage of memory types such as global and texture memory. Each array can be an AoS, *multidimensional* (**n-Array**), or both (**n-AoS**). These optimizations allow a high flexibility but introduce a very vast number of possible implementations that can be greater than a million per application.

Like many other state of the art auto-tuning approaches we rely on empirical profiling as most of the optimization effects are difficult to model, especially for data dependent effects such as bank conflicts and serialization of atomic operations. Empirical profiling has the advantage that it executes the application directly on the specific hardware for realistic data. On the other hand empirical profiling is very time intensive since it requires many different trial runs of the application. Further each kernel configuration that is evaluated has to be explicitly compiled prior to its execution, which often takes significantly more time than the execution itself. Our work mainly focuses on real-world applications whose total execution time is quite long. This poses a difficult task for empirical profiling as only a few configurations can be evaluated and compiled in an acceptable time frame if a single application run does not only take a couple of seconds.

In this paper we focus on the question, how to find a nearly optimal memory configuration for a single kernel, with no dependencies to other kernels, in as little time as possible. Our contributions are:

1. We introduce a novel adaptive profiling algorithm, that uses a predictor with domain knowledge to concentrate the search on the approximated location of the global minimum. It requires only few profiling runs to find a near optimal solution and is several orders of magnitude faster than an exhaustive search.
2. We evaluate our approach against a series of state of the art adaptive algorithms
3. We present a GPU watch-dog to reduce the profiling time, which can be combined with many different profiling algorithms

The outline of this paper is as follows: Section 2 relates our work to current state of the art methods. A brief introduction to our auto-tuner is then given in Section 3. Section 4 explains our algorithm and Section 5 shows how we integrate it into the auto-tuner. In Section 6 we explain our evaluation method, followed by the results in Section 7. Section 8 gives a conclusion and an outlook on future work.

2. RELATED WORK

In the following we introduce previous work that is closely related to ours. We start with other GPU memory access optimization frameworks, continue with exhaustive search based auto-tuning approaches and end with adaptive profiling methods.

2.1 GPU Memory Access Optimizations

Optimal memory access has been specifically targeted by Rubin et al. [23] who published the MAPS framework. It provides STL-like data access containers that map on common GPU algorithms. Their approach optimizes memory access but is no real auto-tuning framework, as no automated decisions are made. Instead the programmer has to explicitly decide which data structures to use and if a matrix should be stored transposed. The only automated adaption their framework is applying is to detect the GPU architecture and to apply certain optimizations specifically targeting this architecture.

Kofler et al. [14] optimize the tile size of AoSoA data structures for OpenCL applications. They analyze the GPU code and build a memory access graph model. This is combined with a generic GPU performance model, which they establish using a predefined benchmark. Both models together allow to estimate the optimal tile size. In contrast, our auto-tuner relies on empirical profiling of the targeted GPU kernel without predefined benchmark or model. We have observed that the properties of real data can have a great impact on the performance of an algorithm if the data itself influences the execution, e.g., through bank conflicts or atomic operation serialization.

2.2 Exhaustive Search Based Auto-Tuning

Muralidharan et al. [17] proposed an optimization framework, which selects the correct algorithm for a problem depending on application specific meta data provided by the programmer. Unfortunately, the authors do not mention how much time their approach requires to acquire the necessary classification data and how much time is spent to convert the input data between different formats, e.g., in their Sparse Matrix optimization benchmark. Weber and Goesele [25] optimize memory access patterns for AoS and n-Arrays. They learn a decision model, which enables them to predict a good working memory layout during runtime. All these approaches, however, have in common that they need to run the application multiple times using different input data and perform an exhaustive profiling to establish their decision models. Depending on the number of configurations, an exhaustive search requires a prohibitively large amount of time.

2.3 Adaptive Profiling Methods

Adaptive profiling tries to reduce the effort required for profiling by selecting a subset of configurations instead of performing an exhaustive search. Jordan et al. [13], e.g., use a genetic algorithm to steer the profiling. Their approach solely relies on the genetic algorithm and does not introduce any kind of domain knowledge. Chung et al. [5] proposed Active Harmony, a general framework with API interface, which could be adapted for GPUs. This framework tries to profile and adapt an application during runtime. They use a heuristic to prioritize the search of their Nelder-Mead based algorithm. Ansel et al. [2] introduced a similar generic auto-tuning approach called OpenTuner. Instead of optimizing an application at runtime, it compiles and restarts it for each new configuration. They use multiple search algorithms to find an optimal solution and can execute the application on multiple machines simultaneously to speed up the optimization. Given our focus on tuning complex applications which require a significant amount of time to compile, it would take too much time to recompile the entire application hundreds or thousands of times. Instead we apply an in-application profiling which compiles only necessary kernels in parallel. This is much faster than recompiling and restarting the entire application. Their approach, however, seems to be well suited for small applications. Liu et al. [15] introduced an input adaptive optimization framework. They use a greedy algorithm which optimizes each dimension separately until it reaches a maximum. Their method also has to recompile the entire application for each evaluated configuration. All of these approaches use no knowledge of the optimization domain while profiling. In contrast, we developed a predictor, which analyzes the optimization space and estimates the location of the optimal solution. This minimizes the number of required samples and ultimately reduces the profiling time.

Another way to limit the number of evaluated configurations are search space pruning algorithms such as proposed by Castro et al. [7]. They sample a certain number of values per dimension to estimate the location of important regions in their search space. In these regions the number of samples is increased, to improve the search for the minimum. Unfortunately, memory access patterns pose a multidimensional discrete optimization problem with very few (usually less than five) values per dimension. This prevents us from applying these techniques to our problem, as the search for important regions, would already execute all possible values of a dimension.

3. AUTO-TUNER DESIGN

Our auto-tuner is designed to optimize complex array layouts in NVIDIA CUDA [18] applications. The main goals for our design are to apply our optimizations with a minimal programming overhead (by using mostly native CUDA or C++ syntax), high compatibility and portability between platforms (Windows <> Linux). As we only want to remove the daunting task of finding an optimal memory layout from the programmer, we do not want to introduce any new programming paradigm.

3.1 Supported Optimization Dimensions

The auto-tuner can optimize several dimensions that have a key impact on memory related performance. It supports different transpositions of arrays, layouts (AoS, SoA and AoSoA) and the placement of data in different memory types (e.g., in global or texture memory) for AoS, n-Array or n-AoS data structures. For AoSoA we use a tile size of 32 which matches the warp size. While the layouts for global and shared memory access can be varied, we do not perform

Listing 1: Code changes necessary to apply auto-tuner.

—— CUDA kernel code ——

```
__global__ void kernel(Data* data, int w, int h) {
  int a = 0, b = 0;

  for(int y = 0; y < h; y++) {
    a += data[threadIdx.x + w * y].a;
    b += data[threadIdx.x + w * y].b;
  }

  ...
}
```

—— GPU code for auto-tuner ——

```
__global__ void kernel(Data data) {
  int a = 0, b = 0;

  for(int y = 0; y < data.getCount<1>(); y++) {
    a += data[threadIdx.x][y].a;
    b += data[threadIdx.x][y].b;
  }

  ...
}
```

—— CUDA Driver API CPU code ——

```
Data* host = new Data[w * h];
CUdeviceptr device;
cuMemAlloc(&device, sizeof(Data)*w*h);

for(int x ...) for(int y ...) {
  host[x + w * y].a = ...;
  host[x + w * y].b = ...;
}

cuMemcpyHtoD(device, host, sizeof(Data)*w*h);

cuModuleLoad(&module, "file.ptx");
cuModuleGetFunction(&func, module, "kernel");

void* args[] = {&device, &w, &h};
cuLaunchKernel(func, ...);
```

—— CPU code for auto-tuner ——

```
Data::Host host(w, h);
Data::Device device(w, h);


for(int x ...) for(int y ...) {
  host[x][y].a = ...;
  host[x][y].b = ...;
}

cuMemcpyHtoD(device, host, host.getSize());

cuModuleLoad(&module, "file.cu");
cuModuleGetFunction(&func, module, "kernel");

void* args[] = {device};
cuLaunchKernel(func, ...);
```

an automated partitioning of which data should be placed in shared memory and which should not. In our opinion, this decision should be made by the programmer, as this impacts the semantics and limit the design space of the code. Further, the auto-tuner adjusts the size of the L1 cache on Fermi and Kepler cards. On these architectures it is possible to increase the amount of L1 cache by reducing the size of shared memory. This can improve the performance as it increases the chance of a cache hit in the L1 cache. On the other hand it can reduce the occupancy of the kernel, depending on the amount of shared memory used, often resulting in a performance drop. Another feature are user defined preprocessor based optimizations, that can evaluate different code paths. This can be used to implement a variety of different algorithms.

We also experimented with more optimization options, but these either have changed the result (e.g., *–use_fast_math* compiler flag) or had no impact on the performance (e.g., adjusting the shared memory bank width). Further we do not optimize the kernel block size as this requires suitably structured kernels and a mechanism to provide valid launch configurations.

Equations 1 to 5 show a summary of all possible optimization dimensions with their respective dependencies.

$$D_{L1Cache} = \begin{cases} [SM, L1] & \text{on Fermi} \\ [SM, L1, EQ] & \text{on Kepler} \\ [None] & \text{otherwise} \end{cases} \quad (1)$$

$$D_{Layout} = [AoS, SoA, AoSoA] \quad (2)$$

$$D_{Memory} = [Global, Texture] \quad (3)$$

$$D_{Transposition} = [1, \ldots, d!] \quad (4)$$

$$D_{Preprocessor} = [\ldots] \quad (5)$$

Using these dimensions, we can define a configuration $C = (d_1, \ldots, d_n)$ of a kernel as tuple of values for the specific dimensions D of the kernel.

3.2 Watch-dog Timer

As the time difference between the best and the worst possible configuration of a kernel can be very high, we implemented a watch-dog, which can terminate the kernels during profiling, if the execution exceeds the time of the best found configuration. The watch-dog uses the internal nano second timer to check if the timeout has been reached. It is implemented inside the kernel as — to our knowledge — this is the only safe way to terminate a kernel without destroying the GPU context. It requires some manual interaction, as the programmer has to specify the location of the watch-dog checkpoints. Further some additional registers are used, which, as well as the checkpoints, can influence the performance of the kernel. We will show in our evaluation that the additional overhead caused by the watch-dog has minimal impact on the decision on the optimal kernel configuration. Further the watch-dog is only compiled into the kernels for profiling and is removed in production runs.

3.3 Code Modifications

As adapting a compiler chain is a big problem for many application maintainers and can be very difficult if maintained for different platforms, we rely on code generation. The auto-tuner generates application specific code, that can be used with any compiler. The programming interface intercepts most CUDA Driver API calls, so that nearly no changes to the code have to be done. Solely the memory access of multidimensional arrays and additional features such as user defined compiler flags or optimization hints (e.g. array sizes that are known at compile time), have to be explicitly coded. To make it easy to access memory, we use a default AoS syntax (*array[x].item*) for one dimensional AoS and mimic a Java-like syntax for n-Array (*array[x][y]*) and n-AoS (*array[x][y].item*). These are realized using C++ operators. The number of used dimensions is arbitrary (e.g. *array[a][b][c][d][e][f]*). Listing 1 shows an example for necessary changes to the code.

To achieve optimal performance, it is crucial that the exact layout implementation is directly compiled into the kernel, because a variable implementation would drastically decrease the kernel performance. The entire kernel compilation process is performed by the auto-tuner. To provide the data in the correct format to the GPU, our host implementation is able to adjust itself during runtime.

3.4 Execution & Runtime

During the normal execution of an application, our auto-tuner intercepts all kernel calls, selects an optimal kernel configuration ($C_{opt} \in C$) and executes it. This interception does not produce any measurable overhead.

In profiling mode, we intercept all kernel calls and pass the control of the application to our profiling adapter. This performs an in-application profiling, which allows a fine granular steering of the process for each kernel and ensures that application setup and finalize procedures are not executed multiple times, which would be the case if the entire application is restarted. The profiling adapter takes care of all necessary profiling operations, data restoration, compilation, data format conversions and initialization of the watchdog. Compilation and format conversions are automatically executed on the CPU in parallel to the GPU profiling. The adapter makes it very easy to implement different search algorithms on top of it. It is built in a blocking queue fashion, so that a search algorithm is filling the queue and then waits until it is processed. The algorithm then can either refill the queue with other configurations or return control to the application. Some hardware limitations, such as the maximal size for texture memory [19], can only be evaluated during runtime. All configurations that do not fulfill these limitations are automatically skipped, regardless of the used profiling algorithm.

4. PREDICTOR

For the search of an optimal kernel configuration, we use a predictor, which estimates the location of the minimum. Our method only requires to sample a very limited amount of configurations of the entire optimization space. The total configuration count $|C|$ can be very high, as it is scales exponentially:

$$|C| = \prod_{d \in D} |d| \tag{6}$$

4.1 Prediction Theory

Our predictor is based on the assumption that different options have a linear contribution to the total execution time. E.g. moving a specific array from global to texture memory or switching it from AoS to SoA will always change runtime by $\Delta_{global \to texture}$ and $\Delta_{AoS \to SoA}$ respectively. While this may sound unintuitive, extensive tests on real applications performed prior to this work suggest that this assumption is actually true for a surprising number of cases.

The explanation for this lies in the way GPUs work. In a warp, all threads either execute the same operation or are deactivated. That is why accessing one array does not influence any other array access. However this only applies to the array access itself. Dimensions such as the L1 cache influence the performance of all arrays. We therefore subdivide all dimensions into two classes, one which contains all dimensions, which are independent D_I and all others who share their contribution to the execution time D_S. As layouts, transpositions and the used memory only influence the performance of a specific array, we define all of our dimensions to be independent, except for the L1 cache size. User defined dimensions can either be declared as independent or shared.

These assumptions allow us to calculate a predicted execution time $P(C_P)$ for a specific configuration as the sum of the execution time $T(C_B)$ of a base configuration plus the time differences $\Delta(C_{S,d_I}, C_B)$ between C_B and a set of support configurations C_S.

$$P(C_P) = T(C_B) + \sum_{d_I \in D_I} \underbrace{\left(T(C_{S,d_I}) - T(C_B)\right)}_{\Delta(C_{S,d_I}, C_B)} \tag{7}$$

This predictor has the advantage that it only requires very few samples to estimate the performance of the entire optimization space. It is important to select the necessary base and support configurations correctly. For the base configurations we require each combination of the shared dimensions.

$$|C_B| = \prod_{d_S \in D_S} |d_S| \tag{8}$$

The values for the independent dimensions do not matter and can be initialized with a default value. We use $Layout = SoA$, $Transposition = untransposed$ and $Memory = global$ as well as the first option of the user defined dimensions. The number of support configurations calculates itself by the sum of all values for the independent dimensions, subtracted by one, as the default value is already covered by the base configurations. Further this number has to be multiplied by the number of base configurations.

$$|C_S| = |C_B| \cdot \sum_{d_I \in D_I} (|d_I| - 1) \tag{9}$$

Figure 1 shows an example with three dimensions: L1 cache, Layout and Memory. The L1 Cache is the only shared dimension, both others are independent. With the 12 highlighted configurations, we are able to estimate the performance of the entire optimization space consisting of 18 configurations.

5. IMPLEMENTATION

To apply the predictor in the auto-tuner, some more work has to be done. Let us assume a very simple kernel with a 5x5 AoS consisting of two fields, a memory access as shown in Listing 2, and a device whose memory controller can fetch one memory bank with 4 adjoining items at once. For our theoretical system each memory bank load requires 10 clock cycles and one additional clock cycle for reading memory banks that are not adjoined. 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

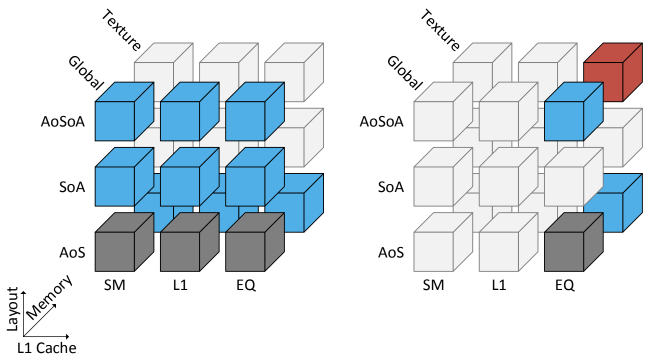


Figure 1: **Left:** To predict the performance of all configurations, we require three base (dark grey) and nine support (blue) configurations. With these we can predict the time all of non-profiled configurations (light grey). **Right:** The estimated performance of the red configuration is calculated by using one base and two support configurations.

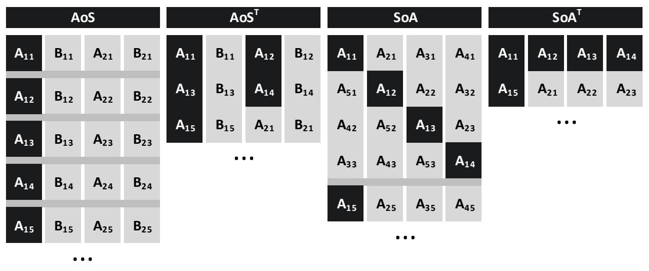


Figure 2: Example for storing of a 5x5 AoS. Each line represents a memory bank with 4 items. Black boxes show accessed items in the first iteration of Listing 2. Gray bars indicate where the data is scattered over the memory.

of four configurations. Figure 2 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. Further we require two support configurations, which are AoS^T and SoA. Their measured execution time is $T(AoS) = 54$, $T(AoS^T) = 30$ and $T(SoA) = 51$ clock cycles for the inner loop. When we apply our predictor (Equation 7) we get the predicted execution time $P(SoA^T) = 27$ which is the best result.

Although this value differs from the exact value $T(SoA^T) = 20$, it is still a useful prediction. In fact the difference is caused by the choice of parameters for our artifical example. But we expect a deviation between a prediction and a real

Listing 2: Pseudocode of n-AoS memory access example

```
struct AoS {int a; int b};
AoS array[5][5];

int sum = 0;
for(int y = 0; y < 5; y++)
  for(int x = 0; x < 5; x++)
    sum += array[y][x].a;
```

system anyway. As we only want to locate the minimum, we do not care about the correct predicted value, as long as the ordering is not changed, so that the predicted minimum is at the same location as the real one.

5.1 Real World Implementation

In a real system we encounter some additional difficulties. Noise is in particular a big issue when using empirical profiling. The reasons for this noise are manifold, starting by varying CPU/memory frequencies, PCI-E bus occupancy and utilization, driver or operating system overhead, task scheduler, and many more. To compensate falsely predicted results, we explicitly sample the five best candidates for the minimum, yielding ground truth timings instead of predictions, which are, however, still contaminated by remaining noise.

To demonstrate that our predictor works as expected, we show an example based on the binning kernel of our KD-Tree benchmark, running on a Tesla K20. The kernel consists of one 1D-AoS (layout), two read only 2D-AoS (layout, transposition, memory), two user defined preprocessor dimensions and the setting for the L1 cache, which is a total of 10 dimensions. Each of the user defined preprocessor dimensions has two options, while all other dimensions use the values defined in Equation 1 to 5. This results in a total of 5184 configurations for the kernel.

$$\underbrace{3}_{\text{L1 cache}} \cdot \underbrace{(3)^1}_{\text{AoS}} \cdot \underbrace{(3 \cdot 2 \cdot 2)^2}_{\text{2x 2D-AoS}} \cdot \underbrace{(2)^2}_{\text{2x user defined}} = 5184 \qquad (10)$$

To estimate the performance of all configurations, we require three base configurations C_B and 36 support configurations C_S.

Figure 3 shows the measured (black line) and predicted (blue line) execution time as well as the location of the base (black crosses) and support (red crosses) configurations. All results are sorted by the measured execution time, so that the x-axis shows the rank of the configuration. Our five best candidates are indicated as orange crosses. On the right we show a closeup of the region around the minimum. As expected, there is a difference between the predicted and measured results, but the configurations that we have found are very close to the optimal solution. To achieve this, we only had to sample 44 out of 5184 configurations, which is 0.8% of the entire solution space.

5.2 Watch-Dog

A naïve watch-dog implementation would always terminate the execution of a kernel, when the execution time exceeds the time of the best configuration. As this would break the predictor since necessary values are missing, some rules have to be applied to the watch-dog. First of all, a base configuration cannot be killed, as these are essential for the prediction. Further, all support configurations which have the same values for the shared dimension and vary only in the same independent dimension, can be put into a so called watch-dog group. As we are only interested in finding a minimum, we are satisfied with the fact, that for each independent dimension, we have the minimal value available. This allows us to kill all configurations inside this group, if

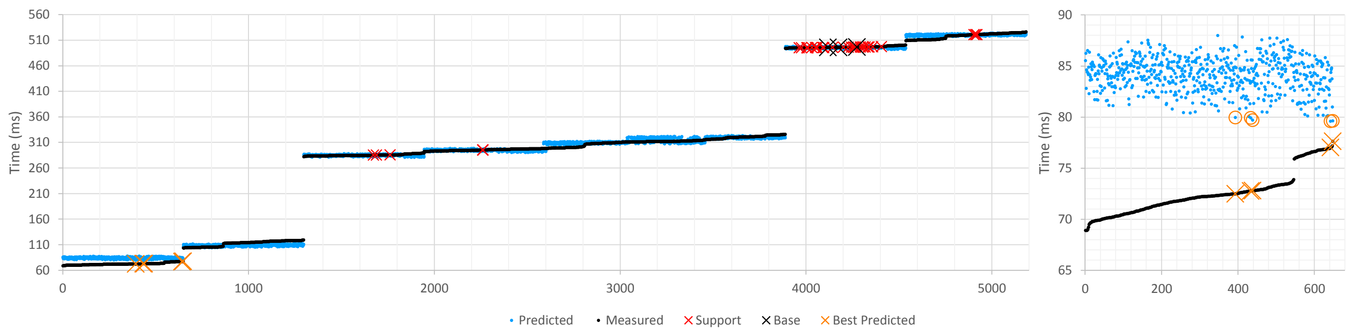


Figure 3: Prediction results for the binning kernel of our KD-Tree benchmark on a Tesla K20. All 5184 measured results (black) are sorted ascending. The x-axis shows the rank of the configurations. Our predictor requires 3x C_B (black cross) and 36x C_S (red) samples to predict the performance of the remaining C_P (blue). The five best predicted configurations are indicated in orange. On the right we show a detail view of the best performing configurations, where circles indicate the predicted and crosses the real value.

it is slower than the best found configuration in the same group.

6. EVALUATION

To evaluate our predictor based profiling, we use multiple benchmarks with different memory access characteristics. We apply three benchmarks ranging from simple algorithms up to entire applications. Each of these has other optimization difficulties, such as only one optimal solution for any architecture or input data (BitonicSort), input sensitive optimal solutions (KD-Tree), and an extremely high number of degrees of freedom (Reyes).

For the evaluation we compare an *exhaustive search* (**E**), a simple *greedy algorithm* (**G**), an *evolutionary algorithm* (**A**), a *random sampler* (**R**) (which samples the same amount of configurations as our approach) and our *predictor* (**P**). If applicable for the benchmark and search algorithm, we execute them with and without watch-dog (a **W** is added to the abbreviation of the algorithm if enabled). For completeness we also show the results for always choosing AoS or SoA as layout, while all other parameters such as memory placement, transposition and L1 cache config, are set fixed to their default value.

The greedy algorithm, as it is also used by Liu et al. [15], only varies the values for one independent dimension at the same time. Every value of this dimension is executed in all combinations with the shared dimensions. If the watch-dog is enabled, it always kills a kernel execution, as soon as the time exceeds the time of the best configurations, that has been found so far.

The evolutionary algorithm is based on the work of Storn and Price [24] which is also the foundation of the approach of Jordan et al. [13]. Our implementation uses a population size of 10 and terminates if it has not found a better solution in 3 generations. These parameters have shown to be a good trade off between good quality and low profiling time for our optimization problem. For kernels with very few variants it supports a shortcut option to directly execute an exhaustive search instead if the number of configurations is very low, which is faster in this case, as the compilation for all configurations can be done in parallel, instead in multiple population runs. The nature of an evolutionary algorithm prohibits the usage of the watch-dog as this would remove the ability to cross-over or mutate configurations, as only the best would not be killed.

As mentioned in the introduction, this paper focuses on searching a good configuration for a single kernel and therefore does not find a global optimal solution for all kernels in an application. This would require to find a common configuration for all arrays, that are shared between different kernels. Currently our algorithm does not supported this (neither do the other adaptive algorithms). We will discuss this matter later as part of the future work.

6.1 Hardware and Software

All evaluations have been performed on two systems. The first is equipped with 2x Intel Xeon E5-2670, 32 GB RAM, NVIDIA Tesla K20, SUSE Enterprise Server 11.3 and CUDA 7.0 (driver version 346.46). The second system consists of 2x Intel Xeon E5649, 48 GB RAM, NVIDIA GeForce GTX980, Ubuntu 14.04 and CUDA 7.0. (driver version 346.59)

Depending on the architecture, the number of possible configurations varies as, e.g., the GTX980 no longer supports adjusting the size of the L1 cache. Configurations that are limited by the hardware are automatically excluded from the profiling.

To compare the quality of the algorithms, we compare the kernel execution time that our auto-tuner achieves with the time that the original implementation of the benchmarks takes. Except for the BitonicSort, all of these have been hand tuned for a particular architecture by their respective authors. For time measuring we use the *nvprof* command line tool.

6.2 Benchmarks

In the following we will explain the benchmarks we are using. Square brackets indicate which kernel is performing the described operation.

BitonicSort: BitonicSort [3] is a parallel sorting algorithm used in many GPU applications. In our implementation we are sorting an AoS consisting of four integer values with 64, 32, 16 and 8 Bit, which are used as columns. If the first column matches, we sort the values of the second column and so on, so that we end up with a list of items sorted for each column in ascending order. The application consists of two kernels. One of these [**BS**] is used for iterations where shared memory can be efficiently used to cache data, while the other [**BG**] directly operates on global memory. We limit the input data to integer values between 0 and 1023 (for the 8 Bit value, it is truncated) so that the probability of equal numbers in one column is increased, which ensures that we do not only sort the first column. For the evaluation we execute seven different data sets with item counts ranging from 64 Ki to 4 Mi. The data sets vary significantly with sets consisting of random numbers, but also sets with partially ascending or descending sorted segments. The implementation we are testing against is the only benchmark that uses a naïve AoS layout. It has a total of 36/12 (K20/GTX980) different kernel implementations.

KD-Tree Builder: This benchmark is a KD-Tree Builder, which resembles the work of Popov et al. [21]. The application consists of eight kernels. Two of these perform the main algorithm, while the others mainly perform maintenance tasks, with a very low total execution time.

The first main kernel [**KB**] discretizes a triangulated scene in multiple bins which are separated by equidistant planes in all three dimensions. Then all triangles in the scene are processed and the number of starting and ending triangles in a bin are counted. In the last step a prefix and postfix sum are executed on the starting and ending values. With these values, the kernel calculates a building heuristic that is used to select the best split plane. Our optimizable version uses an adjustable preprocessor implementation, which is able to buffer the binning results in local memory instead of using an *atomicAdd* on shared memory.

The second kernel performs the splitting of a subtree and stores all necessary data in two different data segments. Additionally it has to perform some recalculations if a triangle is located on the split plane [**KS**].

The maintenance kernel [**KA**] is run at the beginning of the application once, to calculate the Axis Aligned Bounding Boxes for the input geometry. [**KI**] initializes the default data for each iteration step. [**KO**] and [**KL**] calculate necessary offsets for storing the results of the [**KS**] kernel. [**KH**] compacts the header data if subtrees have been marked as a leaf node, which are then no longer present in the next iteration. [**KF**] is a post processing kernel for [**KS**].

All kernels are build in a fashion that they can process multiple subtrees in parallel. This application has a total of 570 k/190 k (K20/GTX980) configurations.

We run this application using 32 bins and the Happy Buddha[1] model, which consists of 1 M triangles. The implementation we are comparing against uses a mixed set of data structures such as AoS, SoA or hierarchical-AoS (e.g. *aabb[a].point[b].dim[c]*).

[1]`http://graphics.stanford.edu/data/3Dscanrep/`

Reyes: Our last benchmark is based on the REYES rendering system [6]. It renders higher order surface patches by adaptively dividing them into micro-polygons of sub-pixel size. Our implementation follows [20] and consists of four kernels. The first kernel subdivides the surface patches into micro-polygons [**RB**]. After each execution of this kernel we run a compaction kernel [**RC**]. These kernels loop until all patches have sub-pixel size. The third kernel [**RD**] projects each micro-polygon into the world coordinate system and shades it, storing the results in a fragment buffer. This buffer stores the depth and color of a pixel and is then used in a fourth kernel [**RT**] to extract the color information, downsample it to the final resolution and store it in an OpenGL texture which can be displayed. This application uses a high amount of different array types which results in a total of 2.4 M/809 k (K20/GTX980) possible combinations.

For our evaluation we run the application and render one frame. As model we are using the Utah Teapot and render it at 7680 x 4320 px which is then downsampled to 1920 x 1080 px in the last kernel to compensate for aliasing effects. The implementation we are testing against has been hand tuned for a GTX680 on which it achieves real-time performance. Further the application is mainly computationally and not memory bound.

7. EXPERIMENTAL RESULTS

In the following we present and analyze our experimental results. We analyze each benchmark separately for quality, time required for the profiling and the impact and overhead of the watch-dog. Figure 4 and 5 show the speed up that all search algorithms achieved compared to the optimal solution found using an exhaustive search. All results are normalized by the performance of the original benchmark implementation. Further we show the speed up for choosing only AoS or SoA as implementation, which a naïve programmer might do. For these cases all transposition, L1 cache and memory dimensions are set to their default value, while the layout dimensions are either AoS or SoA. Figure 8 shows the percentage of total execution time for a single kernel for the AoS, the original and the optimized implementation. In Figure 6 we show the profiling speed up compared to an exhaustive search while in Table 1 the total profiling time as well as how much more time the algorithm required is shown, compared to the fastest. Figure 7 shows the average and 90% confidence interval of the watch-dog overhead compared between the results from the exhaustive search with and without it. We only compare kernel executions, that have not been terminated. In the end we discuss the optimal configurations of the kernels, and what steps have to be done during a normal application run, to achieve an optimal performance. Further we discuss if a purely AoS or SoA layout suffices to achieve an overall satisfactory performance.

BitonicSort: As can be seen in Figure 4, all algorithms achieve the same execution speed up on this benchmark, even the random sampler. This is not surprising as all algorithms sample nearly the entire solution space, as a total of 36 configurations is too small. Further, as shown in Figure 6 and Table 1, no algorithm terminates faster than the exhaustive search, often they are slower. The reason for this is the compile process. The exhaustive search schedules all configurations for compilation at the same time. Caused by the

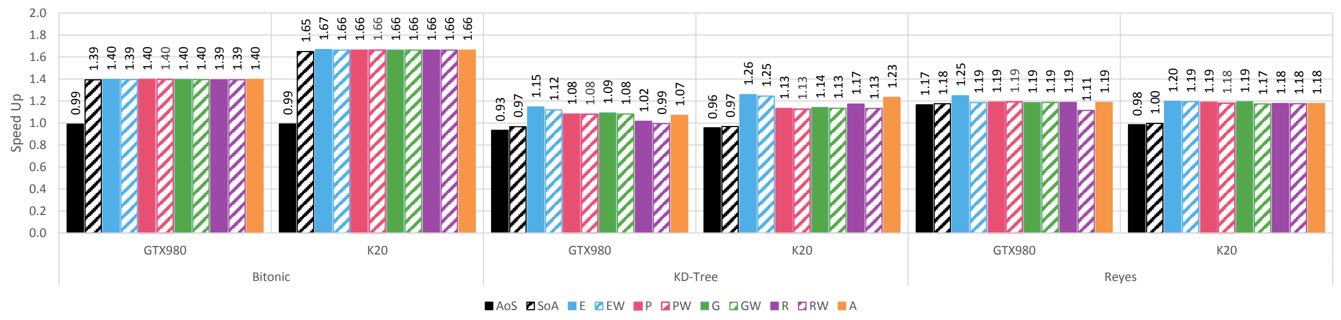


Figure 4: Total execution speed up for all benchmarks.

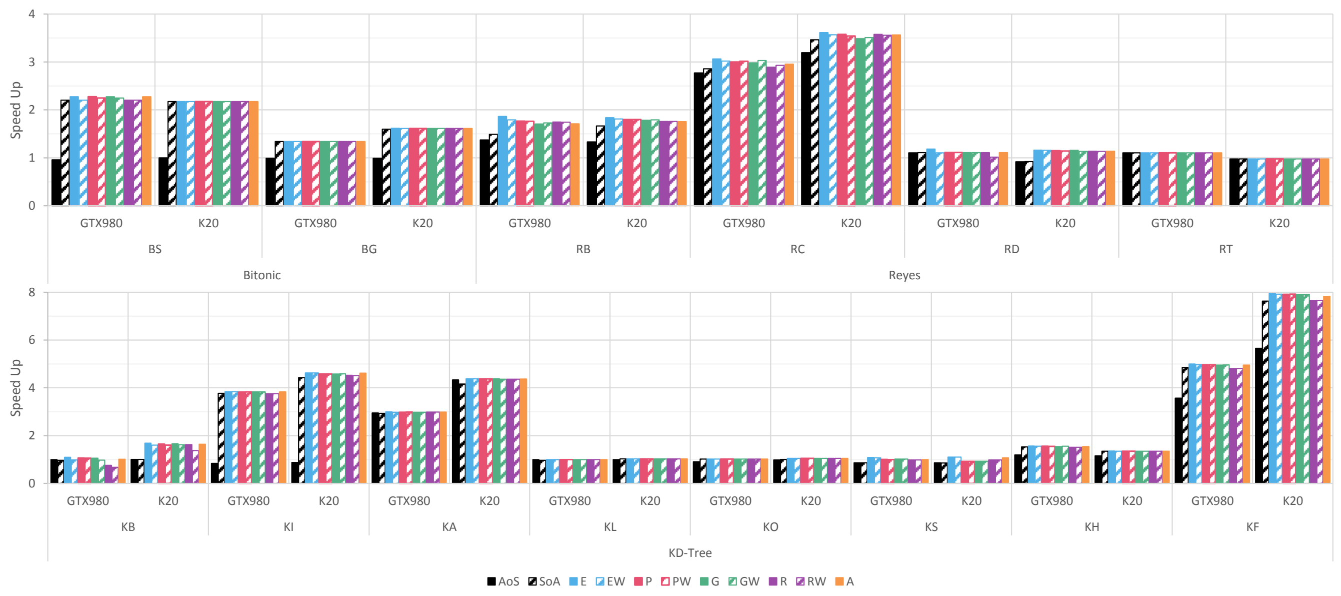


Figure 5: Execution speed up achieved over all kernel executions compared to the original implementation.

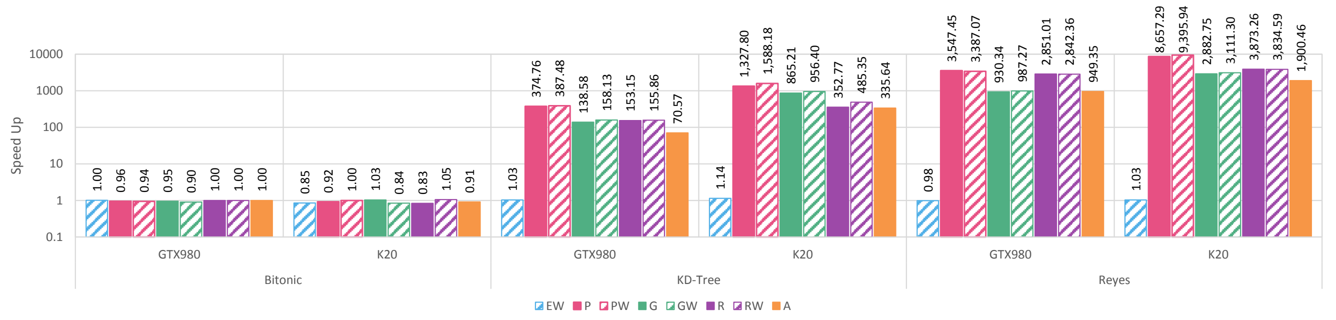


Figure 6: Profiling speed up of all algorithms compared to an exhaustive search. (logarithmic scale)

GTX980	**E**	**EW**	**P**	**PW**	**G**	**GW**	**R**	**RW**	**A**
Bitonic	50s (1.0x)	**50s (1.0x)**	53s (1.0x)	53s (1.1x)	53s (1.1x)	56s (1.1x)	50s (1.0x)	50s (1.0x)	50s (1.0x)
KD-Tree	2d 15h 4m 3s (387.5x)	2d 13h 10m 9s (375.8x)	10m 6s (1.0x)	**9m 46s (1.0x)**	27m 18s (2.8x)	23m 56s (2.5x)	24m 42s (2.5x)	24m 17s (2.5x)	53m 37s (5.5x)
Reyes	4d 16h 45m 28s (3547.5x)	4d 19h 5m 45s (3621.0x)	**1m 54s (1.0x)**	1m 60s (1.0x)	7m 16s (3.8x)	6m 51s (3.6x)	2m 22s (1.2x)	2m 23s (1.2x)	7m 8s (3.7x)
K20									
Bitonic	1m 50s (1.1x)	2m 9s (1.2x)	1m 59s (1.1x)	1m 50s (1.1x)	1m 47s (1.0x)	2m 11s (1.3x)	2m 12s (1.3x)	**1m 45s (1.0x)**	2m 1s (1.2x)
KD-Tree	11d 12h 47m 28s (1588.2x)	10d 2h 42m 12s (1392.6x)	12m 30s (1.2x)	**10m 27s (1.0x)**	19m 12s (1.8x)	17m 22s (1.7x)	47m 5s (4.5x)	34m 13s (3.3x)	49m 29s (4.7x)
Reyes	10d 19h 28m 2s (9395.9x)	10d 12h 4m 34s (9128.3x)	1m 48s (1.1x)	**1m 39s (1.0x)**	5m 24s (3.3x)	5m (3.0x)	4m 1s (2.4x)	4m 4s (2.5x)	8m 12s (4.9x)

Table 1: Time required to gather all necessary profiling data. Fastest algorithm is highlighted. Shown factors are relative to best algorithm.

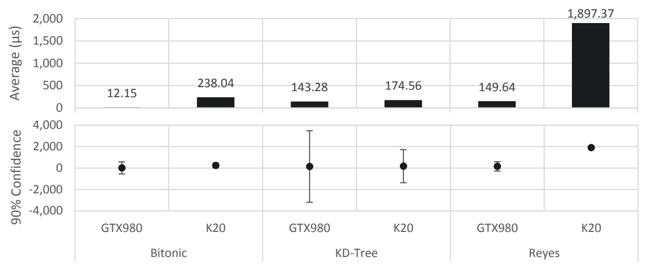


Figure 7: Average overhead and 90% confidence interval caused by the watch-dog compared to the same kernel without it.

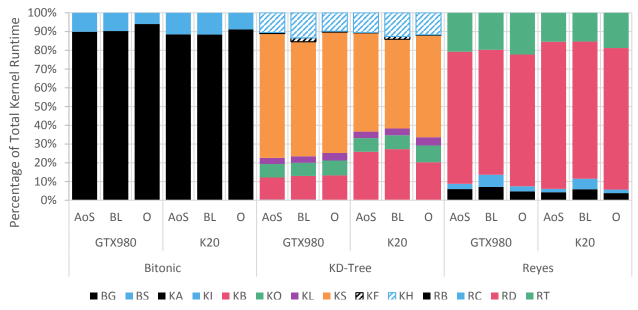


Figure 8: Percentage of total runtime per kernel for AoS, baseline (BL) and the optimized version (O).

very low amount of configurations and high number of cores on the test machines, the process is done entirely in parallel. Contrary the adaptive algorithms schedule multiple compilation runs and therefore have a serialization, which causes longer profiling times. This shows, that for applications with very small configuration counts, an exhaustive search suffices and is most likely faster than any adaptive algorithm.

As can be seen in the total and profiling speed up, the watch-dog neither improves the search speed, nor does it negatively influence the outcome. Furthermore, the overhead caused by the watch-dog is minimal (see Figure 7).

KD-Tree: For the KD-Tree we can see that no algorithm achieves the same performance as the exhaustive search. On the GTX980 all algorithms except for the random sampler reach 7-9% speed up, compared to 15% for the exhaustive search. The reason for this lies in the [**KS**] kernel. The kernel implementation itself is at an occupancy border, so that increasing register usage causes a drop of occupancy. As this resource usage can hardly be predicted, none of the algorithms can compensate this in any way.

On the K20, the evolutionary algorithm achieves nearly the same performance as the exhaustive search, while the other algorithms reach up to 13% less performance. As the register count on the K20 is equal to the GTX980, the explanation is the same.

As before, the watch-dog hardly influences the achieved performance. The speed up drops 1-3% for the exhaustive search and random sampler, where it is 0-1% for the other algorithms. Figure 7 shows that the average overhead is minimal again, but the confidence interval is very high for the GTX980. We actually do not have an explanation for this; it could be caused by the way the internal nano second timer values are polled and distributed onto the threads.

However, our algorithm is 387 to 1588 times faster than an exhaustive search, as well as 5.5 times faster than the other adaptive algorithms, which reduces a 53 minute long profiling down to less than 10 minutes (see Figure 6). Although the random sampler profiles the exact same amount of configurations as our approach does, it is significantly slower as it has to compile nearly every configuration it is executing, while our algorithm mostly reuses the same configurations.

Reyes: In the Reyes benchmark on the K20, all algorithms achieve nearly the same performance as the exhaustive search, while the performance is 6% slower on the GTX980. This is caused by the [**RD**] kernel, which is entire computational bound. This means that most configurations achieve the exact same performance, so that the gradient (which all adaptive algorithms are based on) is close to zero between most of the over 700 k configurations. The Kepler architecture is less efficient on atomic operations, so that there is a slight shift from purely computational to memory boundness. This helps the algorithms to find better solutions.

As can be seen in Figure 7, the average overhead for the watch-dog is very high on the K20. The reason for this is again the less efficient atomic operations. At the beginning of a kernel, our watch-dog stores the initial value of the nano second timer using an atomicMin. The high number of blocks of the [**RD**] kernel causes here a blocking of the kernel. Nevertheless, as can be seen in the benchmarks, this additional overhead has no negative impact on the decision of the algorithms.

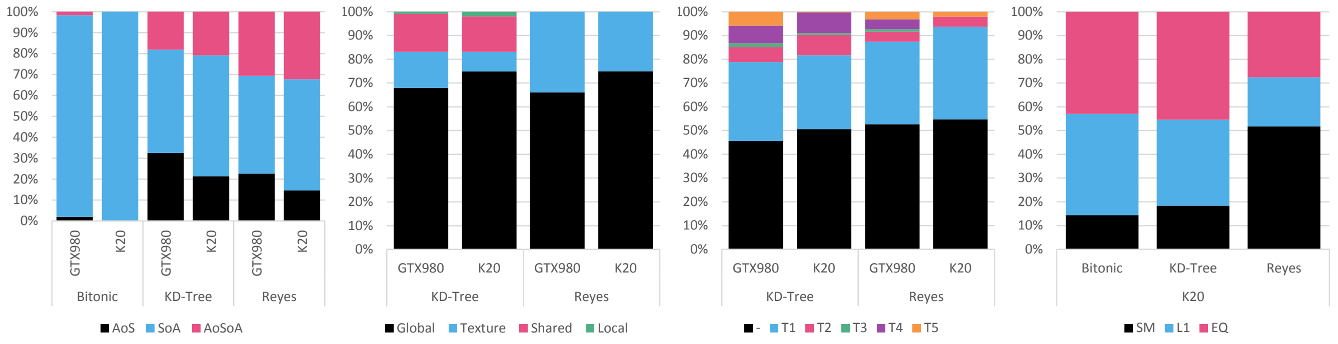
Also in this benchmark, our algorithm outperforms the other adaptive algorithms up to factor 4.9. Further we are 3387 to 9396 times faster than the exhaustive search, which reduces the execution time from nearly 11 days down to 2 minutes.

7.1 Discussion

Figure 9 shows the percentage of how often which layout, transposition, memory or L1 cache option was used in the optimal configurations. The results show that there is no clear option that is superior to the others, but that the optimal solution requires a carefully chosen mixture of all available options.

Very surprising is the result of the optimal L1 cache setting, as preferring shared memory (SM) is the default CUDA behavior. However, in the Bitonic and KD-Tree it is only in less than 20% of the cases the optimal choice. For the Reyes benchmark it is optimal in about 50% of the cases.

Additionally on the memory usage of the KD-Tree we can see, that depending on the input data and workload of a kernel, the optimal implementation varies. The [**KB**] kernel is able to store its binning results in either shared or local memory. As can be seen, in a certain amount of cases, local memory is preferred. The reason for this is, that in the first iterations there are only a few, but very big subtrees, where the binning causes a lot of bank conflicts. In later



(a) Layout (b) Memory (c) Transposition (d) Cache

Figure 9: Percentage of how often which layout, memory, transposition or cache configuration was used to achieved the best performance. - for transposition means untransposed. As high dimensional arrays can be transposed in multiple ways, these transpositions have been assigned different numbers.

iterations the number of subtrees increases while their size shrinks significantly, as the triangles are split among the subtrees. This change causes that the advantage of the local memory turns into an disadvantage. To compensate this, a different implementation of the kernel has to be chosen dynamically during runtime, depending on the workload, to achieve optimal performance.

Further we can see that both GPUs differ in their optimal configurations. This makes it necessary to tune an application towards a specific GPU.

Overall the results clearly show the advantage of auto-tuning frameworks, as these can adapt the application to a specific hardware and changing workload, with hardly any user interaction, while hand tuning always is very time consuming and often also error prone.

8. CONCLUSION

In this paper we have presented a prediction guided profiling algorithm for reducing the time required for empirical profiling. Our approach is designed for complex applications with hundreds of thousands or millions of configurations and is able to speed up the profiling process up to 9396 times over an exhaustive search, reducing the profiling time from 10 days 19 hours down to less than 2 minutes, while achieving nearly the same performance. Compared to other state of the art profiling algorithms we outperform these by up to factor 5.5, while achieving the same quality.

Further we introduced a watch-dog which helps to decrease the profiling time, by terminating the execution of kernel runs in bad configurations. None of the profilings was negatively influenced by the watch-dog.

As previously mentioned, our auto-tuner is at the moment only able to find optimal configurations for each kernel separately, as data that is shared between different kernels is not taken into consideration. Caused by adaptive profiling, it is not guaranteed to have all necessary results available, to calculate an optimal global solution for this (this is a problem for all algorithms that only partially profile the optimization space, not only of ours). That is why we want to investigate how to find global optimal layouts for this shared data, without the need to convert these between the execution of different kernels. One solution would be to use a prediction model to fill the gaps. As shown in Figure 3, our prediction follows the measured results, but the predicted value can have a relative offset. This is too imprecise to actually use it as a real prediction model, so it cannot be used to estimate the performance of the non measured configurations. We want to investigate options to improve our prediction.

Further we have observed that the optimal solution of a kernel can vary, depending on the input data. Therefore we want to establish a prediction comparable to Muralidharan et al. [17], which dynamically chooses the best configuration during runtime, depending on the kernel's input data.

9. 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.

10. REFERENCES

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proc. PLDI*, 2009.

[2] 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 *Proc. PACT*, 2014.

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

[4] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. Technical report, 2010.

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

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

[7] P. de Oliveira Castro, E. Petit, A. Farjallah, and

W. Jalby. Adaptive sampling for performance characterization of application kernels. *CCEP*, 2013.

[8] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *Proc. SIGPLAN PLDI*, 2013.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. IEEE/SC*, 2006.

[10] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen. A Model-driven Partitioning and Auto-tuning Integrated Framework for Sparse Matrix-vector Multiplication on GPUs. In *Proc. TeraGrid*, 2011.

[11] T. D. Han and T. S. Abdelrahman. HiCUDA: A High-level Directive-based Language for GPU Programming. In *Proc. GPGPU*, 2009.

[12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proc. ASPLOS*, 2012.

[13] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proc. SC*, 2012.

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

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

[16] T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM TACO*, 2013.

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

[18] NVIDIA. CUDA Developers Network. http://developer.nvidia.com/. [online, accessed on 06.09.2015].

[19] NVIDIA. Cuda Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [online, accessed on 06.09.2015].

[20] A. Patney and J. D. Owens. Real-Time Reyes-Style Adaptive Surface Subdivision. *ACM TOG*, 2008.

[21] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proc. IEEE IRT*, 2006.

[22] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. SIGPLAN PLDI*, 2013.

[23] E. Rubin, E. Levy, A. Barak, and T. Ben-Nun. Maps: Optimizing massively parallel applications using device-level memory abstraction. *ACM TACO*, 2014.

[24] R. Storn and K. Price. Differential Evolution - A Simple and Efficient Heuristic for Global Optimization ofer Continuous Spaces. *GO*, 1997.

[25] N. Weber and M. Goesele. Auto-Tuning Complex Array Layouts on GPUs. In *Proc. EGPGV*, 2014.

[26] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. SC*, 1998.

[27] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proc. PLDI*, 2010.

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