Eurographics Symposium on Parallel Graphics and Visualization (2014)
M. Amor and M. Hadwiger (Editors)

Auto-Tuning Complex Array Layouts for GPUs

Nicolas Weber and Michael Goesele

Graduate School of Computational Engineering, TU Darmstadt, Germany

Abstract

The continuing evolution of Graphics Processing Units (GPU) has shown rapid performance increases over the years. But with each new hardware generation, the constraints for programming them efficiently have changed. Programs have to be tuned towards one specific hardware to unleash the full potential. This is time consuming and costly as vendors tend to release a new generation every 18 months. It is therefore important to auto-tune GPU code to achieve GPU-specific improvements. Using either static or empirical profiling to adjust parameters or to change the kernel implementation. We introduce a new approach to automatically improve memory access on GPUs. Our system generates an application specific library which abstracts the memory access for complex arrays on the host and GPU side. This allows to optimize the code by exchanging the memory layout without recompiling the application, as all necessary layouts are pre-compiled into the library. Our implementation is able to speedup real-world applications up to an order of magnitude and even outperforms hand-tuned implementations.

Categories and Subject Descriptors (according to ACM CCS): D.3.3 [Programming Technique]: Language Constructs and Features—Data types and structures I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

In recent years Graphics Processing Units (GPU) have emerged as an alternative to classical CPUs. Their massively parallel architecture provides a tremendous amount of compute power and can significantly improve performance of an application. Unfortunately using GPUs is much more difficult. The architectural differences compared to a CPU require often low level programming and are much harder to optimize. Unleashing the full potential requires the programmer to have good knowledge of the actual hardware. The vast plethora of hardware constraints for different GPU vendors and generations interfere with the optimization of applications. New GPU architectures are released every 18 months and require time-consuming changes in the implementations.

It is well known that memory access and in particular the memory layout of complex arrays is essential for performance on GPUs (as described in chapter 5.3.2 in [NVIb]) due to their lack of significant caching and hardware prefetching. This applies not only to the memory access pattern in the GPU code but also to the way data is provided to the GPU in the first place. Therefore we propose an auto-tuning approach which generates optimized host and device code based on empirical memory usage observations and machine learning techniques. We focus on the optimization of Array of Structs (AoS) and multi-dimensional arrays. Our approach trains a model that is capable of finding the best suited memory layouts. Furthermore, we are able to train our model for varying array sizes and select the best performing implementation depending on the amount of data.

We developed a prototype of our approach called "MATOG Auto-Tuning on GPUs" (MATOG) which is able to optimize the usage of device memory as well as shared memory. We further designed MATOG in a way, that allows easy integration in existing applications without the need to reimplement vast parts of the code. MATOG does not rely on a specific compiler, as it generates code for a library, which is then linked to the actual application. This library then is used to access GPU data in the host and device code. Experiments have shown a speedup for existing applications of up to an order of magnitude. To summarize our contributions are as follows:

1. An automated memory layout optimization system, that can be easily integrated into existing applications with a very low code footprint.
2. An approximation to limit the solution space, which

DOI: 10.2312/pgv.20141085

speeds up the construction of the decision model, without introducing a significant performance drop compared to a model trained on the entire solution space.

3. An evaluation of our automatic memory layout optimizations for three complex applications (Bitonic Sort, KD-Tree Binning and REYES rendering) with different characteristics in terms of computation and memory access.

2. Related Work

Simulation tools, metrics, and models are often used to predict different execution properties (e.g., L1 cache rate, memory bandwidth, ...). Schaa [Sch09] developed a model to predict the performance of applications in a multi-GPU environment. He determined multiple influencing performance factors for the execution time and evaluated them with empirical data. Baghsorkhi et al. [BDP*10] profiles a GPU to determine its characteristics and predict the performance of code on that particular hardware. Ganestam et al. [GD12] use an analytical GPU model to predict the performance in a ray-tracing application, and to adjust the calculation to improve interactive performance. Models like these are widely used in auto-tuning compilers, which use statical profiling to optimize the code to perform best on a given hardware. Some compilers require another language, e.g., Rudy et al. [RKH*11] developed CUDA-Chill, an abstract language for parallel processing. This was later used by Khan [Kha12], who developed several code transformations and applied these to GPU code. Copperhead [CGK10], Sequoia [FHK*06], and Terra [DHA*13] hide the low level code using a high level language. This allows to optimize the actual code using their compiler, without any interaction of the programmer. Other systems try to apply their optimizations using preprocessor defines (e.g., HiCUDA [HA09]), which can apply optimizations such as special loop transformations. All these compiler-bound approaches have in common, that they optimize the code with static analysis. This has the advantage, that it is much faster than an empirical optimization. But it is unable to model effects caused by data properties. Instead of being limited to one optimal implementation, our approach allows use of multiple implementations adapted to specific data properties.

Another common approach are Domain Specific Languages (DSL). Halide [RKBA*13] is embedded in C++ and can be used to optimize image processing pipelines, GreenMarl [HCSO12] is focused on graph analysis and Zhang et al. [ZM13] developed a framework to optimize the execution of 3D stencil codes on heterogeneous GPU clusters. These approaches focus on a specific application area which narrows down the space of possible domain specific optimizations. In contrast, our approach can be applied to almost any application without restriction to a specific application domain.

In some auto-tuning applications, empirical data is used to learn or build a model, which predicts the performance of a specific GPU implementation. Guo et al. [GHC*11, GW10] and Choi et al. [CSV10] applied empirical tests and model-driven partitioning to optimize sparse- and dense-matrix multiplications on GPUs. Sorensen et al. [Sør12] uses empirical tests to optimize memory access and the launch configuration of matrix-multiplication kernels. These approaches have shown promising results, but are limited to their specific application. Bergstra et al. [Ber12] uses machine learning techniques and empirical tests to predict the performance of GPU code on other hardware. This approach has some similarities to our approach, as we also learn a decision tree based on empirical measurements. In contrast, we do not learn how an unknown GPU would perform but focus instead on optimizing the execution for a known or related GPU with unknown data properties.

3. Design

In the following we use NVIDIA's CUDA [NVIa] model as a generic GPU model. We focus our optimizations on Arrays of Structs (AoS) and multi-dimensional arrays in global and shared memory. Our optimizations are able to select different implementations depending on the problem size. Further we evaluate if using the texture memory or adjusting the size of the L1 cache (by reducing the size of shared memory) has a positive effect on the performance. The advantage of texture memory is, that it uses an additional read-only cache which can lead to a higher total bandwidth.

3.1. Array of Structs

To optimize an Array of Structs we decided to implement three different layouts. The first one is the intuitive way to simply store the structs in an array, which we refer to as the baseline implementation. The second layout we use is Structure of Arrays (SoA). In this approach all components are stored in separate arrays. This can lead to coalesced memory access, if the access pattern reads adjoining elements. Further we use a hybrid format called Array of Structure of Arrays (AoSoA), as e.g. used by Wald et al. [Wal12]. Data is partitioned in chunks according to the GPU's SIMD width (warp size). Figure 1 shows a comparison between all three formats. For all memory layouts we support to store them in global and shared memory. If data is read-only, we can store each component in a separate texture array and use texture memory to access it. Further, if a struct only needs to be partially updated, it is possible to declare parts of the struct to be read-only, so that they can be stored in texture memory while applying the previously mentioned memory layouts to the other components.

3.2. Multi-dimensional Arrays

Multi-dimensional arrays can be transposed which changes coalescence of the memory access. For these arrays, we refer to a non-transposed array as baseline implementation.

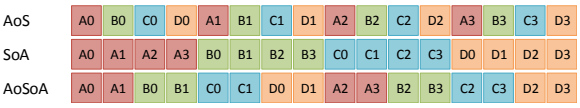


Figure 1: *This Figure shows the differences between AoS, SoA and AoSoA. Each color represents a field in a struct with four different fields. For simplification we used a SIMD width of 2 for the AoSoA case.*

Again, we support storing these in global and shared memory as well as using the texture cache (if declared as read-only).

Additionally we support a special type of data structure for histograms, as many applications require to create a histogram from a certain data set. It is common to use shared memory and increment the necessary cells by an atomic add. Depending on the data this could result in complete serialization and many bank conflicts. Storing all results in a local memory segment for each thread and merging these at the end of the execution can significantly improve the performance if the problem size is large enough to compensate for the additional data merging step. This additional feature can significantly improve performance in some applications.

3.3. Kernel and Data Properties

The optimal solution in terms of memory access for a kernel depends (at least) on the algorithm, the size of the data it operates on and (in some cases) even on the actual data. Some algorithms are only influence by the access pattern. Optimizing such an algorithm can usually be done by hand. Another property that influences the kernel performance is the amount of data. Executing a kernel on 1 kB of data, e.g., might be fast using a simple memory layout as all the data fits into the cache, while using the same kernel with 1 GB of data could result in severe performance problems as cache efficiency can be very low for certain memory layouts. To handle this, we keep track of all variables that are allocated during the execution. With this information we are able to reference each variable and its data to each kernel execution. This allows us to build a decision tree, determines the optimal memory layout for a given data size. The optimization towards the data values are very difficult to handle as our approach handles optimization for arbitrary input data and does therefore not know anything about data properties such as the sorting order. During the learning phase, our approach is able to find the best possible solution for a given data set. But it is unable to guarantee the best performance for other input data.

4. Learning

In order to train decision trees for the automatic memory layout selection we first need to gather the necessary training data. To obtain this data we run the application multiple times using different memory layouts and measure the runtime of each kernel execution. Since the space of possible solutions is vast, we introduce two learning modes: *complete* and *small*. The complete mode tests all possible combinations of memory layouts. Further we allow that each layout is run with several different test cases provided by the programmer. This way we can learn a separate decision tree for each kernel, which can determine the best layout for several different data sets. In small mode we test all global and all shared memory layouts separately. In the global memory tests we use the baseline implementation for shared memory and vice versa. This approximation drastically reduces the amount of test cases. However it is unable to profile the relationship between global and shared memory layouts. This relationship can, however, be neglected for many kernels, if the amount of time to copy data from global to shared memory and vice versa is significantly smaller than the execution time of the kernel itself (see Section 6 for a detailed evaluation on several examples). The equation $|R_c| = |G| \cdot |S| \cdot |P| \cdot |T|$ defines the number of required runs for the complete mode and $|R_s| = (((|G| + |S|) \cdot |P| - 1) \cdot |T|$ for the small mode, where G is the set of available global memory layouts, S is the set of shared memory layouts and T is the set of test cases. P decides if the kernel shall prefer L1 cache or shared memory. Depending on the number of global and shared memory layouts that have to be tested, the difference between both modes can be huge so that the small mode drastically reduces the time needed to optimize the application.

4.1. Decision Tree Construction

Given the profiling data, we are now able to build a non-binary space partitioning tree, where each tree level represents one variable. Each node in the tree has a threshold. If the data amount is smaller than the threshold, the corresponding subtree contains the solution. If the node is a leaf, it directly contains the solution. If the data amount is bigger than the threshold, the next node on the same tree level will be evaluated. As we cannot expect to cover the entire argument space of the profiled application, we apply some assumptions. If we have different best implementations for two array sizes, we subdivide the array size space in the center of both data points. For regions of the argument space, where we do not have any profiling data, we apply another best solution determined by matching the arguments. We prioritize variables by the order of their occurrence in the kernel arguments. This might not lead to the best possible solution in this particular region of the argument space, but gives a stable tree construction. This representation is able to provide a definite decision for each combination of input data. Figure 2 shows the decision tree of one of our evaluation examples for a two dimensional argument space.

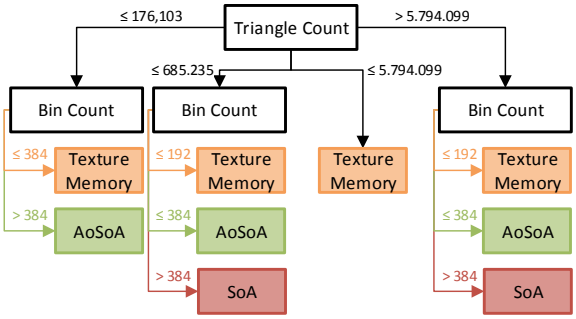


Figure 2: *Decision tree for the KD-Tree binning example (see Section 6.2). The tree determines the best global memory layout for storing the triangle bounding boxes of the 3D scene depending on the triangle and the bin count. A figure showing the position of the test samples is provided in the supplemental material.*

5. Implementation

Our prototype design focuses on achieving the optimization goal while keeping the required adjustments to an existing application to an absolute minimum. To achieve this goal the application must be written using the CUDA Driver API. This API allows much more control over the execution on the GPU. We use to exchange the kernel implementation during the runtime by switching the CUDA module. We create the different CUDA modules using CMake by compiling the kernels in all necessary variants. Further we use NVIDIA's CUDA Profiling Tools Interface (CUPTI). This profiling API allows us to register callbacks and gives us the opportunity to track kernel parameters as well as the performance of the kernels. Our system consists of three components. The first one is a *code generation tool* which generates a library that abstracts the optimizations from the programmer and provides simple transparent access to the data by using classes with getter and setter functions. The second component is a *shared library* which performs all operations inside the application which are necessary to apply the optimizations. The final component is a tool which profiles the applications and creates decision trees for the best solutions.

5.1. Library Generation and Profiling

For library generation we require an XML description (e.g. Listing 1) of the application that lists in particular the data structures to be optimized and provides information about the kernels for the automatic CMake project generation. Using this XML description, the generator decides which optimizations can be applied to the project and generates the corresponding code. The generated CMake project automatically compiles the library and creates platform independent PTX files for all kernels. We pre-compile all possible combinations of memory layouts for each kernel. This allows us to optimize the application without any recompilations and makes it easy to distribute it to different architectures. For development one can compile only the baseline implementation, so that recompilation during development can be done very quickly. Listing 2 shows an example host implementation for an application using MATOG. The code demonstrates the usage of a generated AoS called *MyStruct* and loading of the best suited kernel implementation using *loadFunction*, which masks the Driver API calls of *cuModuleLoad* and *cuModuleGetFunction*. Listing 3 shows the corresponding implementation of a kernel, with shared memory use. As previously stated, we use a separate tool to optimize the application. Our optimization tool starts the application with a set of previously user defined test cases.

```
<matog>
  <cuda mincc="2.0" />
  <cmake libname="myLib">
    <kernels>kernels/myKernel.cu</kernels>
  </cmake>
  <code>
    <struct name="MyStruct" shared="true">
      <field name="a" type="long" />
      <field name="b" type="int" />
    </struct>
    <array name="MyArray" type="float" />
  </code>
</matog>
```

Listing 1: *MATOG XML-description*

```
#include "MyStruct.h"

MyStruct& data = *new MyStruct(count);

for(...) {
  data[index].a = valueA;
  data[index].b = valueB;
}

data.copyToDevice();

CUmodule module; CUfunction func;
loadFunction("module", "function", module, func);

const MyStruct::GPUObject obj = data.getGPUObject();
void* args[] = {&obj, 0};

cuLaunchKernel(func, ...);

// ...

delete &data;
```

Listing 2: *A host code example showing the conceptional usage of MATOG.*

```
#include "MyStruct.cu"
__global__ myFunctionName(MyStruct data) {
  __shared__ MyStructShared<N> shared;
  shared.copyToShared(data, offset);

  for(...)
    shared.b[index] += shared.a[index];

  shared.copyToGlobal(data, offset);
}
```

Listing 3: *A kernel example showing the usage of MATOG.*

If the programmer has not provided any test cases, it assumes a single default test case with no command line arguments. Depending on the selected learning mode, the optimization tool executes the application several times using different memory layouts. During these runs, we use CUPTI to trace the performance of each kernel separately for each of the available implementations. We further extract all kernel arguments and use them to reference each variable to each kernel that is using it. At the end of the optimization step, all data is gathered and the per-kernel decision trees as well as the tree for the usage of global memory are build as described in Section 4.1.

5.2. Program Execution

During the normal operation of the application, we disable CUPTI, so that no additional overhead is created. As we have to decide on a global memory layout as soon as data is written to memory or the first kernel is loaded, we implemented a lazy allocation scheme. This way, MATOG does not decide on a memory layout when the array is instantiated but when the first data is written to it. As we do not support to transform data, the decision on the global memory layout is fixed for the entire execution of the application as soon as data is written to the first array. In contrast the decision for the shared memory access of a kernel is done every time the kernel loading function is executed (see Listing 2). It returns the optimized kernel, by traversing the decision tree, using the sizes of allocated arrays. If there are no decision trees available for the used GPU we check if another device with the same compute capability has been profiled and use its results. This assumes that different GPUs of the same compute capability and therefore of the same generation have similar effects on code adjustments. If there are no decision trees for a GPU of the same generation available we use our baseline implementation.

5.3. Usage

As already mentioned using MATOG is quite easy. The first step is to create the XML description of the required data structures. With this document, the library generator is able to generate the necessary code and CMake project. MATOG's compile process automatically detects changes to the XML description, so that adding new structures during the development is easy and does not require to rerun the generator by hand. There are no additional steps required to develop the application. It also can be run without any restrictions directly after compilation but will always use the baseline implementation, as long as it was not optimized using the MATOG optimizer. A complete code example is provided in the supplemental material.

6. Evaluation

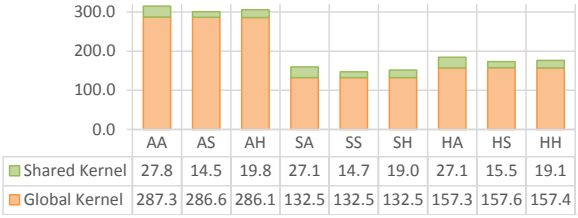
In the following sections we evaluate MATOG using three existing applications of various complexity. We performed our tests on a NVIDIA Geforce GTX570 and GTX680 (both run in headless configuration). All timings for learning are given for the GTX680.

6.1. Bitonic Sort

Bitonic Sort is a parallel sorting algorithm which was introduced by Batcher [Bat68] and is widely used in GPU programming. Our implementation consists of two kernels, that are chosen according to the step size in the sorting procedure. For small step sizes, we use a kernel which uses shared memory. As big step sizes do not allow to use shared memory efficiently, we have a second kernel, which directly operates on global memory. As MATOG does not optimize simple arrays, we decided to sort an AoS with four columns. For these columns we use unsigned integers with 64 bit, 32 bit, 16 bit and 8 bit as variable sizes. The data is sorted column wise. For the evaluation we use a data set with 4.1 M elements. The numbers are generated randomly in the range of 0 through 1023. In the case of the 8 bit field we truncate the upper bits, so it contains values between 0 and 255. We limit the number range to increase the likelihood of conflicts, so that our algorithm has to compare not only the first column for sorting the array.

For training we use a limited data set with 1 M random elements. The small learning mode requires 2.2 s for 9 runs, while the complete mode requires 4.6 s for 18 runs to learn. Both methods come to the conclusion that using SoA is always the best access pattern for global and shared memory access. Each thread inside a warp has to check the first column inside the struct at the same time. By using SoA all these values are stored at adjoined positions. Assuming a 128 B cache line, using SoA the GPU would be able to read up to 16 items from our first column in the struct. With AoS the GPU would only be able to load a maximum of 8 items per cache line. This results in twice the amount of load operations than using SoA. Figure 3 shows the total time required for all memory access layouts with the 4.1 M random data set on the GTX680. We omit the results using the option to use 48 kB instead of 16 kB L1 cache as they are significantly slower than using 48 kB shared memory. The reason for this is the kernel implementation as it does not make excessive usage of local memory so that reducing the shared memory results in lower occupancy. Further we do not show the results of the GTX570 as they are similar.

For this case it is easy to see, that using SoA for global and shared memory is the best solution, as both of our learning methods have predicted. On the GTX680 it is 2.14 times faster than the baseline implementation (AoS). Further we can see, that the execution time of both kernels is approximately halfed.



	AA	AS	AH	SA	SS	SH	HA	HS	HH
Shared Kernel	27.8	14.5	19.8	27.1	14.7	19.0	27.1	15.5	19.1
Global Kernel	287.3	286.6	286.1	132.5	132.5	132.5	157.3	157.6	157.4

Figure 3: *Time (ms) required to sort 4.1 M random elements for each kernel and all memory access patterns using GTX680. A = AoS, S = SoA, H = hybrid (AoSoA). First character specifies the global layout, the second the shared layout.*

6.2. KD-Tree Binning

Our second example is the binning step of binned KD-Tree construction [PGSS06]. These trees are usually used as acceleration structures for ray tracing but can also be used as search trees for other applications. We do not perform the entire construction of the tree, but only the root node binning of a 3D scene. For this step, a histogram of the triangle distribution in the 3D scene with equidistant bins is created. Each dimension is processed independently, yielding three separate binning results. Our kernel uses 128 threads per block and iterates over all triangles in parallel. For each triangle we determine the bin where it starts and the bin where it ends. We implemented these as arrays with our previously introduced histogram implementation. After all triangles have been binned, we have to calculate a prefix sum on the first array and a postfix sum on the second one. To simulate higher load on the GPU, we perform 256 parallel binning operations resulting in a total of 768 blocks.

For this example, MATOG applies AoS, SoA, AoSoA as layout or uses texture memory to access the triangle bounding boxes. The histogram arrays are stored either directly in shared memory or are buffered in local memory. We use twelve 3D scenes from a variety of sources, e.g., architecture, cars, 3D scans and artificial structures, ranging from 69 k to 28 M triangles. For training our decision trees we use two 3D scan and two architectural scenes with 69 k, 282 k, 1 M and 10 M triangles. For the bin count we use 32, 64, 128, 256 and 512 as values. This results in a total of twenty learning cases. Our small mode takes 26 min for 15 runs, while complete mode takes 68 min for 32 runs per test case. Unfortunately, our 3D scene loader requires over 90 % of the total execution time (which does, however, not impact the learning).

For the evaluation we run all possible layout combinations with all remaining 40 evaluation sets (consisting of 8 scenes and 5 different bin counts) and the optimal solution by hand (see Table 1). The results show a maximal speed up of nearly 10 times for the GTX680, while the average speed up is always higher than a factor of 2 and is close to the optimal average for both GPUs. Unfortunately MATOG chose in 25% of the GTX570 test runs a solution which was slower than the baseline implementation (which however did not have a serious drop of the average speed up). The fact that the small learning chose more often an optimal solution shows that brute force learning does not guarantee best results if the input data has different data properties than the learning data. As stated before, our approach is able to learn optimal layouts for given data properties but is unable to distinguish between them (except data amount). Please refer to the supplemental material for more detailed charts of the results.

	GTX 680		GTX 570	
	Complete	Small	Complete	Small
Min	0.70	0.76	0.90	0.92
Max	9.83	9.83	4.78	4.78
Avg.	2.47	2.39	2.12	2.28
$\lvert < 1 \rvert$	6	1	9	10
$\lvert opt \rvert$	7	12	11	16
Opt. Avg.	2.99		2.51	

Table 1: *Results for the binning evaluation showing the used GPU, learning mode, the minimal, maximal and average speedup, as well the counts of cases with a speed up below 100% ($\lvert < 1 \rvert$) and the count of cases where MATOG chose the best possible implementation ($\lvert opt \rvert$). The last row shows the average speed up, if we had selected the optimal solution in all cases.*

Kernel	Hand-Tuned	Complete	Small
Dice & Shade	3834.31	3642.73	3673.72
Paint	630.90	601.69	602.90
Compact	374.01	351.90	351.50
Bound & Split	67.81	64.75	64.90
Total	4903.97	4661.07	4693.02

Table 2: *Accumulated kernel execution time (ms) on the GTX680 required to render 100 frames. It is easy to see, that the Dice&Shade kernel requires 80% of the total execution time.*

6.3. REYES

This application is based on the REYES system [CCC87] which is widely used in feature films. It renders higher order surface patches by adaptively dividing them into micro-polygons of subpixel size. Note that this is a large and complex system — far more complex than typical test cases such as matrix multiplication — which can demonstrate the properties of MATOG under (approximately) real-world conditions. Our implementation consists of four different kernels and follows the work of Patney and Owens [PO08] and Zhou et al. [ZHR*09]. The first kernel performs the bound and split operation which subdivides the micro-polygons until they are smaller than one pixel. This is an iterative process

which requires multiple runs of the same kernel. After each bound and split run, we compact our data using a second kernel. If all micro-polygons have been processed we perform the dice and shade operation, which projects all micro-polygons into the world coordinate system. In the same kernel the micro-polygons are rasterized and stored into a z-buffer. Our fourth kernel extracts image information from the z-buffer and stores it in a 2D image. The original implementation itself was hand tuned using AoS and untransposed matrices.

As input model we use the Utah teapot composed of Bezier-Patches. We use a resolution of 1920×1080 and render 100 frames for each run to reduce the impact of noise. Our small mode takes 19 min for 81 runs and our complete mode takes 86 min for 296 runs to learn. The baseline implementation achieves 20.37 FPS. The frame-rate matches the rate of our original implementation which suggests, that our baseline implementation performs as expected and does not produce any additional overhead. The learned solutions achieve 21.45 FPS for the complete mode and 21.31 FPS for the small mode, which is a speedup of 5.3 % and 4.6 %. Despite the fact that the input application was already hand-tuned, we were still able to improve the performance. Table 2 shows the time required by each kernel.

7. Conclusion and Future Work

In this paper we demonstrate that improving memory layouts on GPUs can have a significant impact on execution time, also for real-world applications. Our evaluation has demonstrated the influence of global and shared memory layouts. Further it showed the necessity of auto-tuning as the input data can significantly influence performance for both global and shared memory layouts. Tuning at runtime is required as the input data can have a significant impact on the decision of which implementation to choose. In the KD-Tree example we have been able to achieve twice the speedup on the newer GTX680, than on the older GTX570. Although we assumed, that it should be easier to unleash the full potential on a newer card without any tuning, we showed the exact opposite. In the REYES example we have been able to improve the execution of a hand tuned application, which has been optimized over several days while including MATOG took only a couple of hours. Further we presented two learning methods to train our decision models. While the complete method showed the best overall speedup, our small method performed similar but required significantly less time to learn. Overall our results show, that auto-tuning is necessary to reach optimal performance for GPU applications as the variety of different implementations and hardware is too vast for a programmer to optimize in acceptable time.

For future work we are considering to improve the usage of MATOG by removing some of the necessary MATOG calls. Further training time would be significantly reduced by buffering the input data and rerun the kernel instead if restarting the application several times during the profiling. This reduces the required time for applications with long setup time. We are also considering to take other metrics into account to remove profiling runs, that will most likely have no benefit. Additionally we do want to be able to change the data formats during runtime, so that the application can dynamically adjust itself to data changes during execution.

The MATOG source code is available at `http://www.gris.tu-darmstadt.de/projects/matog`.

8. 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.

References

[Bat68] BATCHER K. E.: Sorting Networks and Their Applications. In *Proc. SJCC* (1968). 5

[BDP*10] BAGHSORKHI S. S., DELAHAYE M., PATEL S. J., GROPP W. D., HWU W.-M. W.: An Adaptive Performance Modeling Tool for GPU Architectures. In *Proc. PPOPP* (2010). 2

[Ber12] BERGSTRA PINTO C.: Machine Learning for Predictive auto-Tuning with Boosted Regression Trees. In *Proc. InPar* (2012). 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Proc. SIGGRAPH* (1987). 6

[CGK10] CATANZARO B., GARLAND M., KEUTZER K.: *Copperhead: Compiling an Embedded Data Parallel Language*. Tech. rep., EECS Department, University of California, Berkeley, 2010. 2

[CSV10] CHOI J. W., SINGH A., VUDUC R. W.: Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proc. PPOPP* (2010). 2

[DHA*13] DEVITO Z., HEGARTY J., AIKEN A., HANRAHAN P., VITEK J.: Terra: a multi-stage language for high-performance computing. In *Proc. SIGPLAN PLDI* (2013). 2

[FHK*06] FATAHALIAN K., HORN D. R., KNIGHT T. J., LEEM L., HOUSTON M., PARK J. Y., EREZ M., REN M., AIKEN A., DALLY W. J., HANRAHAN P.: Sequoia: Programming the Memory Hierarchy. In *Proc. IEEE/SC* (2006). 2

[GD12] GANESTAM P., DOGGETT M.: Auto-tuning interactive ray tracing using an analytical GPU architecture model. In *Proc. GPGPU5* (2012). 2

[GHC*11] GUO P., HUANG H., CHEN Q., WANG L., LEE E.-J., CHEN P.: A Model-driven Partitioning and Auto-tuning Integrated Framework for Sparse Matrix-vector Multiplication on GPUs. In *Proc. TeraGrid* (2011). 2

[GW10] GUO P., WANG L.: Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs. In *Proc. IEEE ICCIS* (2010). 2

[HA09] HAN T. D., ABDELRAHMAN T. S.: HiCUDA: A High-level Directive-based Language for GPU Programming. In *Proc. GPGPU* (2009). 2

[HCSO12] HONG S., CHAFI H., SEDLAR E., OLUKOTUN K.: Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proc. ASPLOS* (2012). 2

[Kha12] KHAN M. M. Z. M.: *Autotuning, Code Generation and Optimizing Compiler Technology for GPUs*. PhD thesis, University of Southern California, 2012. 2

[NVIa] NVIDIA: CUDA Developers Network. http://developer.nvidia.com/. [online, accessed on 04/17/2014]. 2

[NVIb] NVIDIA: Cuda Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [online, accessed on 04/17/2014]. 1

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proc. IEEE SIRT* (2006). 6

[PO08] PATNEY A., OWENS J. D.: Real-Time Reyes-Style Adaptive Surface Subdivision. *ACM TOG* (2008). 6

[RKBA*13] RAGAN-KELLEY J., BARNES C., ADAMS A., PARIS S., DURAND F., AMARASINGHE S.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. SIGPLAN PLDI* (2013). 2

[RKH*11] RUDY G., KHAN M. M., HALL M., CHEN C., CHAME J.: A Programming Language Interface to Describe Transformations and Code Generation. In *Proc. LCPC* (2011). 2

[Sch09] SCHAA D.: *Modeling execution and predicting performance in multi-GPU environments*. PhD thesis, Northeastern University, 2009. 2

[Sør12] SØRENSEN H. H. B.: Auto-tuning Dense Vector and Matrix-vector Operations for Fermi GPUs. In *Proc. PPAM* (2012). 2

[Wal12] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE TVCG* (2012). 2

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: RenderAnts: interactive Reyes rendering on GPUs. *ACM TOG* (2009). 6

[ZM13] ZHANG Y., MUELLER F.: Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *Proc. IEEE TPDS* (2013). 2