Automatic Data Layout Optimizations for GPUs

Klaus Kofler[1], Biagio Cosenza[12], and Thomas Fahringer[1]

[1] DPS, University of Innsbruck, Austria
[2] AES, TU Berlin, Germany
{klaus|tf}@dps.uibk.ac.at, cosenza@tu-berlin.de

Abstract. Memory optimizations have became increasingly important in order to fully exploit the computational power of modern GPUs. The data arrangement has a big impact on the performance, and it is very hard for GPU programmers to identify a well-suited data layout. Classical data layout transformations include grouping together data fields that have similar access patterns, or transforming Array-of-Structures (AoS) to Structure-of-Arrays (SoA).
This paper presents an optimization infrastructure to automatically determine an improved data layout for OpenCL programs written in AoS layout. Our framework consists of two separate algorithms: The first one constructs a graph-based model, which is used to split the AoS input struct into several clusters of fields, based on hardware dependent parameters. The second algorithm selects a good per-cluster data layout (e.g., SoA, AoS or an intermediate layout) using a decision tree. Results show that the combination of both algorithms is able to deliver higher performance than the individual algorithms. The layouts proposed by our framework result in speedups of up to 2.22, 1.89 and 2.83 on an AMD FirePro S9000, NVIDIA GeForce GTX 480 and NVIDIA Tesla k20m, respectively, over different AoS sample programs, and up to 1.18 over a manually optimized program.

1 Introduction

With the advent of new massively parallel architectures such as GPUs, many research projects focus on memory optimizations. In order to exploit the properties of the memory hierarchy, a key aspect is to maximize the reuse of data.

In this context, **data layout transformation** represents a very interesting class of optimizations. Two typical examples are: organizing data with similar access patterns in structures or rearranging array of structures (AoS) as structure of arrays (SoA). Recent work extends the classical SoA layout by introducing AoSoA (Array of Structure of Array) [16], also called ASA [14]. In this paper we prefer the expression **tiled-AoS**, but we remark that all approaches exploit the same idea: mixing AoS and SoA in a unique data layout.

1.1 Motivation

In this work, we investigate an automatic memory optimization method that can be easily ported to different GPU architectures, using OpenCL as programming

model. We combine together two different optimization strategies: we try to group together data fields with similar data access patterns and find the best data layout for each of these clusters.

Considering SAMPO [7] as an example, using a struct containing twelve fields. The number of possible ways to partition these twelve fields is equal to $4,213,597$. Considering that this program has minimum run-time of 65 seconds on an AMD FirePro S9000, depending on the data layout, just evaluating all the possible partitions (i.e., clusters) would take more than eight years.

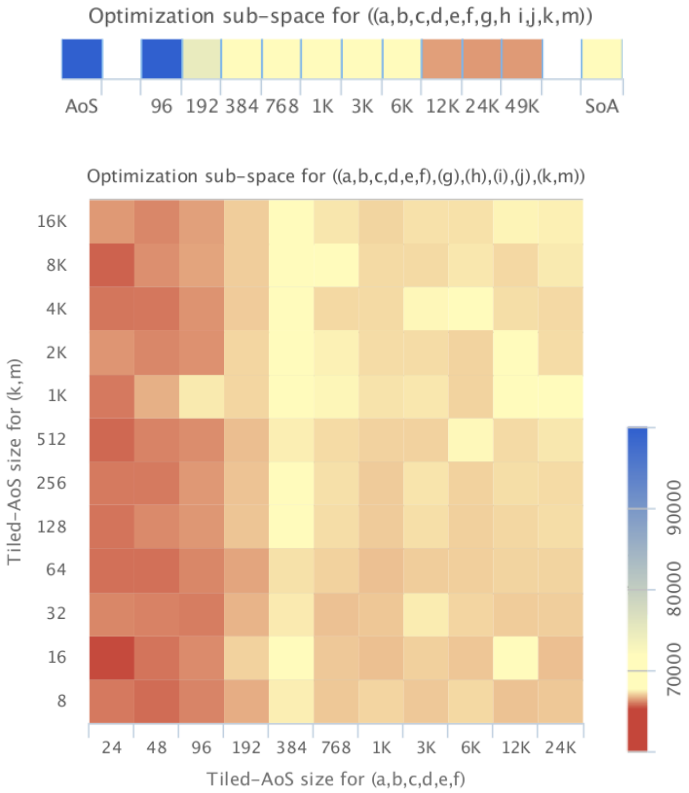


Fig. 1: Excerpt of SAMPO's Optimization Space. Execution times vary from 65 seconds (in red) to 104 seconds (in blue).

The exploration of the whole search space, including both fields' clustering and data tiling (i.e., finding the best data layout for each of these clusters) would take more than 400 years.

Figure 1 shows a subset of the optimization space for SAMPO. The heat-map on top depicts all possible data tiling for the one-cluster grouping of all the twelve data fields. For this partition, the un-tiled AoS layout is slow (blue);

by increasing the data tile-size the run-time decreases (shown in red), and with data tile-size bigger than $12K$ it also outperforms the SoA layout. The lower heat-map shows the performance results while applying the specific data tiling suggested by our algorithm (Section 3.1). The fastest version of the shown optimization sub-space is achieved when we use a tile-size of 16 for the smaller struct containing two fields, 24 for the bigger struct with six fields, and having the other fields in a SoA layout. This example program also shows that the best tile-size can be different within the same code and different clusters: when using only one cluster, the highest performance is achieved with large data tiles; however, different clustering delivers better performance with smaller data tiling sizes. This suggests that the optimal data tile-size highly depends on the size of the individual cluster.

Our work is the first approach which automatically tackles the two problems mentioned above. Our contributions are:

- A Kernel Data Layout Graph ($KDLG$) model extracted from an input OpenCL kernel; each vertex weight represents structure field's size and the edge weight expresses intra-data field memory distance.
- A two-phase algorithm: first, a $KDLG$ partitioning algorithm — driven by a device-dependent graph model — splits the original graph into partitions with similar data access patterns; second, for each partition we exploit a data layout selection method — driven by a device-dependent layout calculation — selects the most suitable layout from AoS, SoA and tiled-AoS layouts.
- An evaluation of five OpenCL applications on three GPUs showing a speedup of up to 2.83.

2 Related Work

The problem of finding an optimal layout is not only NP-hard, but also hard to approximate [11]. Raman et al. [9] introduced a graph based model to optimize structure layout for multi-threaded programs. They developed a semi-automatic tool which produces layout transformations optimized for both false sharing and data locality. Our work uses a different graph based model encoding the variables memory distance and data structure size, in order to provide a completely automatic approach; we also support AoS, SoA and tiled-AoS layouts. Kendermi et al. [5] introduced an inter-procedural optimization framework using both loop optimizations and data layout transformation; our method does not apply to a single function only, but can span over multiple functions.

Data layout transformations such as SoA conversion have been described to be the core optimization techniques for scaling to massively threaded systems such as GPUs [13]. DL presented data layout transformations for heterogeneous computing [15]; DL supports AoS, SoA and ASTA and implements and automatic data marshaling framework to easily change data layout arrangements. Our work supports similar data layouts, but we provide an automatic approach for the layout selection. MATOG [16] introduces a DSL-like, library-based approach which optimizes GPU codes using either static and empirical profiling to

adjust parameters or to change the kernel implementation. MATOG supports AoS, SoA and AoSoA with 32 threads (to match the warp size on CUDA) on multi-dimensional layouts and builds an application-dependent decision tree to select the best layout. Dymaxion [4] is an API that allows programmers to optimize memory mapping on heterogeneous platforms. It extends NVIDIA's CUDA API with a data index transformation and a latency hiding mechanism based on CUDA stream. Dymaxion C++ [3] further extends prior work. However, it does not relieve the programmer from selecting a good data layout.

3 Method

Our approach tries to answer two complex questions: (1) What is the best way to group data fields? (2) For each field cluster, what is the best data layout?

Once clusters have been identified, for each cluster we try to find the best possible layout within that cluster (i.e., *homogenous layout*). Our model supports AoS, SoA, as well as tiled-AoS with different tile-sizes.

In the next section we introduce a novel graph based model, where we encode data layout, field's size and field locality information. The presented two-step approach (1) identifies field partitions (i.e., clusters of fields) with high locality within intra-partition fields and (2) determines an efficient data layout for each partition.

3.1 Kernel Data Layout Graph Model

We define a Kernel Data Layout Graph (*KDLG*) as an undirected, complete graph whose nodes represent fields of the input *struct* (assumed to have AoS layout). The *KDLG* has two labeling functions: σ for verteices, representing the field's data size; δ for edges, representing the memory distance (or inverse-affinity) between fields. Formally, a *KDLG* is a quadruple defined as follows:

$$KDLG = (F, E, \sigma, \delta)$$

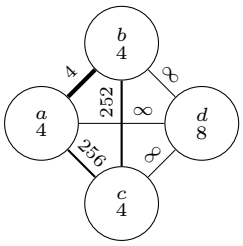
where F is the set of all fields of the struct, which corresponds to the set of nodes in the *KDLG*. $E = F^2 \setminus \{(x, x) | x \in F\}$ is the set of all edges $e = \{(f_1, f_2) | f_1, f_2 \in F\}$. The mapping function $\sigma : F \to \mathbb{N}$ returns the size of a field f in bytes, e.g., if f refers to a field of type *int*, then $\sigma(f) = 4$, according to the OpenCL specifications. $\delta : E \to \{\mathbb{N} \cup \infty\}$ returns the weight of an edge e. The mapping function $\delta((f_1, f_2))$ is defined as the *memory distance* between the two fields f_1 and f_2 by counting the number of unique memory locations, in bytes, touched by the program between the instruction where they are accessed.

We borrow the idea of memory distance from [9] and extend it with the actual data type size, which is important to distinguish different memory behaviors.

The *KDLG* is based on an OpenCL kernel. The set F will have a vertex for each field defined in the structure, which is passed as an argument to the device kernel function. For each vertex f, the σ function returns the actual type's size in bytes of the corresponding field of f .

```
struct T {
    float a, b, c;
    double d;
};
__kernel fun(__global T *t) {
    float a, b, c;
    double d;
    int id = get_global_id(0);
    double sum = 0;
    for(int i=id; i<id+32; i++)
        sum += t[i].a * t[i].b;
    t[id].c = sum;
};
```

(a) Kernel code



(b) Generated *KDLG*

Fig. 2: A *KDLG* generated by a sample input data layout and kernel. Darker edges show fields that are closer in memory (smaller δ).

Figure 2b displays the *KDLG* generated from the code shown in 2a: The fields a and b are always accessed consecutively, therefore $\delta(a, b)$ is 4 bytes. c is accessed after the for loop with 32 iterations, therefore $\delta(c, b) = 252$ and $\delta(c, a) = 256$ bytes, resulting from the 32 iterations that access $2 \cdot 4$ bytes in each iteration. d is never accessed, therefore its distance from other fields is ∞.

Our graph based model unrolls all loops before starting the analysis. Therefore, it assumes that loop bounds are known at compile time. If not known, we use a OpenCL kernel specific loop size inference heuristic to have a good approximation (see Section 3.1). Our analysis focuses on global memory operations, as they are considerably slower than local and private memory operations

Let $MI(f)$ define the set of all global memory instructions (loads and stores) involving the data field f. Our distance function δ between two fields f_1 and f_2 is defined by taking into account the maximum-memory-distance path between the accessing instructions $i_1 \in MI(f_1)$ and $i_2 \in MI(f_2)$.

In order to calculate δ, we use a data flow analysis where each node of the control flow graph (CFG) consists of a single instruction. The function $\sigma(i)$ returns the number of bytes which are written to/read from the global memory in instruction i. We define IN and OUT as

$$IN_i[j] = \min_{x \in pred(j)}(OUT_i[x]) \qquad OUT_i[j] = \begin{cases} 0 & \text{if } i = j \\ IN_i[j] + \sigma(j) & \text{if } i \neq j \end{cases}$$

We define a instruction-memory distance function $MD(i_1, i_2)$ as

$$MD(i_1, i_2) = \max(OUT_{i_1}[i_2], OUT_{i_2}[i_1])$$

so that $MD(i_1, i_2) = MD(i_2, i_1)$. We calculate $\delta(f_1, f_2)$, the memory distance between the fields f_1 and f_2, as the maximum memory distance between all instructions in $MI(f_1)$ and $MI(f_2)$ as follows:

$$\delta(f_1, f_2) = \max\left(\max_{i \in MI(f_1), j \in MI(f_2)} MD(i, j)\right)$$

Therefore, we can use $\delta(f_1, f_2)$ to assign a weight to each edge $(f_1, f_2) \in E$.

We conservatively use the maximum, which leads to higher weights on the *KDLG*'s edges and leads to more clusters; since more clusters have a lower risk of performance loss on our target architectures.

***KDLG* Partitioning** The first step of our algorithm identifies which fields in the input data structure should be grouped together. Formally, we assume that a field partitioning C of the *KDLG* (i.e, field clusters) is *good* if $\forall e \in C | \delta(e) < \epsilon$, where ϵ is a device dependent threshold. We define ϵ as the L1 cache line size of the individual GPUs. The values of ϵ are listed in Table 1. We use this value as it is the smallest entity that can be loaded from the L1 cache and therefore should be loaded at once.

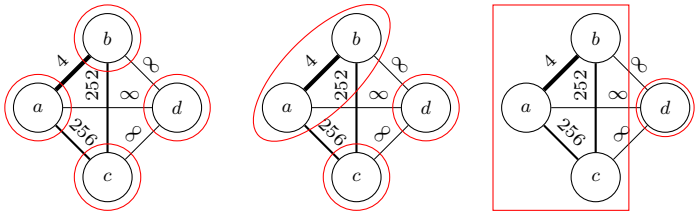
We propose a strategy based on Kruskal's Minimum-weight Spanning Tree (MST) algorithm [8] that extends the classical MST algorithm with an ϵ-based early termination criteria and multiple clusters of nodes (i.e., struct fields).

KDLG-PARTITIONING(F, E, δ, ϵ)

```
1  C = ∅
2  for each field f ∈ F
3        C = C ∪ {{f}}
4  E_ε = {e ∈ E : δ(e) < ε}
5  for each edge (f1, f2) ∈ E_ε
6        c1 = {x ∈ C | f1 ∈ x}
7        c2 = {x ∈ C | f2 ∈ x}
8        if c1 ≠ c2
9              C = (C ∖ {c2, c1}) ∪ {c1 ∪ c2}
10 return C
```

It takes as input a *KDLG*, previously computed from an input kernel, and a threshold ϵ. It starts by creating a partitioning with $|F|$ sets, each of which contains one field in F (lines 1–3). Line 4 initializes E_ϵ for all edges in E with a weight smaller then ϵ, according to the weighting function δ. The **for** loop in lines 5–9 checks, for each edge (f_1, f_2), whether the endpoints f_1 and f_2 belong to the same set. If they do, then the edge is discarded. Otherwise, the two sets are merged in line 9. The complexity of this algorithm is $\mathcal{O}(|E| \cdot |F|)$. Figure 3 shows three possible output partitions that can be generated from the graph seen in Figure 2b using different ϵ values.

Loop Bounds Approximation When generating the test data to select ϵ we use loops with a fixed number of iterations, in order to to accurately understand the memory distance between two memory accesses. In real world codes, the actual number of iterations is often not known at compile time. Therefore we use a heuristic that is specifically designed for OpenCL kernel codes. If the number of loop iterations are determined by compile-time constants, we use the actual number of iterations. If not, we apply a heuristic to approximate the number of iterations: When a loop performs one iteration for each OpenCL work-item [6]



(a) $\epsilon = 1$ (b) $\epsilon = 100$ (c) $\epsilon = 10000$

Fig. 3: Different output partitions using different ϵ values on a *KDLG*.

of the work-group [6], we estimate it has 256 iterations, as the work-group size is usually in this range. When a loop performs one iteration for each work-item of the NDRange [6], we assume it will have $1 \cdot 10^6$ iterations. If the number of iterations is neither constant nor linked to the work-group size or NDRange, we estimate it to have $512 \cdot 10^3$ iterations. The estimation of loop bounds is not very sensitive: we only need to distinguish short loops, which may not completely flush the L1 cache, from long ones.

3.2 Per-Cluster Layout Selection

After KDLG-Partitioning, we assume that each field in the same cluster has similar memory behavior. Therefore, all the fields within a cluster should have the same data layout arrangement, e.g., tiled-AoS with a specific tile-size.

To understand what layout is best for a given cluster, we generate different kernels corresponding to a simple one-cluster *KDLG* where δ is roughly the same for each pair of fields. The kernel consists of a single for-loop with a constant number of iterations n. The value of n comprises all powers of two from 128 to 16384. We evaluated the performance of these kernels with different combinations of loop size n, number of structure fields m, and tile-size t.

From the results we derive a device-dependent function Select-Tilesize($\sigma(c)$) which returns the suggested layout for a cluster c, where $\sigma(c) = \sum_{f \in c} \sigma(f)$ and $\sigma(f)$ returns the size of the field f in bytes. Select-Tilesize is implemented using a decision tree, constructed by the C5.0 algorithm [12]. $\sigma(c)$ is the only attribute the decision tree depends on. The potential target classes are AoS , SoA and all powers of two from 2^1 to 2^{15}. The performance measurements of the aforementioned kernels are used to generate the training data.For each kernel we create a training pattern for the fastest tile-size as well as all other tile-sizes that are less than 1% slower than the fastest one. These training patterns consist only of the size of the structure $\sigma(c)$, which is the only feature while the used tile-size acts as the target value. Generating training patterns not only for the fastest tile-size but for all which achieve at least 99% of it, as well as several training patterns for different structures with the same size, may lead to contradicting training patterns. However, our experiments demonstrated that the

resulting decision tree is more accurate and less prone to overfitting. C5.0 was used with default settings; its run-time was about 1ms, depending on the input.

3.3 Final algorithm

In order to achieve best results, we combine the two algorithms described in Section 3.1 and Section 3.2. Before applying these algorithms, one has to identify the device dependent factor ϵ and construct a decision tree to be used in function SELECT-TILESIZE, as described in the previous sections. Furthermore, the *KDLG* graph is constructed and the actual memory layout for the program to be optimized is selected at compile time. The selection of the memory layout is described by the following pseudo code:

```
LAYOUTOPTIMIZE(F, E, δ, ε)
1  L = ∅
2  C = KDLG-PARTITIONING(F, E, δ, ε)
3  for each cluster c ∈ C
4        t = SELECT-TILESIZE(σ(c))
5        L = L ∪ {(c, t)}
6  return L
```

Line 2 calls the KDLG-Partitioning algorithm and returns a set of clusters C in which the corresponding structure should be split. Then the decision tree determines an efficient tiling factor for each of these clusters and stores the resulting pair (cluster, tile-size) (line 3-5).

4 Experimental Results

To verify the validity of our approach we implemented a prototype of our framework and observed its performance on several OpenCL applications. The deployment of our system is split into two parts: A device dependent part which has to be performed once for each GPU (installation time), and a program dependent part, which is executed at the compile time of the program. These two parts are depicted in Figure 4. The device dependent part consists of identifying the L1 cache line size to be used as ϵ and running a set of training programs to collect the information needed to build the decision tree as defined in Section 3.2. Collecting all the necessary data requires to run many benchmarks takes 196, 158 and 299 minutes on the FirePro, GeForce and Tesla, respectively. The program dependent part constructs a *KDLG* graph for the structure to be optimized in the corresponding program. This graph is hardware independent.By combining the *KDLG* graph with the hardware depended ϵ we split the struct into several clusters (Section 3.1). For each of these clusters we query the hardware dependent decision tree to obtain the tile-size to be used (Section 3.2).

To evaluate our framework we run different programs on three different GPUs. The test programs are listed in Table 2. In each program we focus on the structure with the most instances and try to optimize its layout. The result

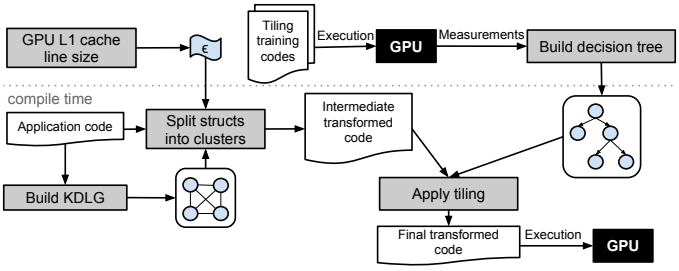


Fig. 4: Work-flow of our data layout optimization process.

Table 1: Properties determined using our algorithms

Hardware	ϵ	Decision tree
AMD FirePro S9000	64	≤ 20: > 48: ≤ 32: 32; AoS; 16384 — ≤ 12: 512; 1024
NVIDIA GeForce GTX 480	128	≤ 12: ≤ 96: SoA; 512 — ≤ 8: SoA; AoS
NVIDIA Tesla K20m	128	≤ 48: ≤ 96: 32768; 16384 — ≤ 20: 8192; SoA

of our framework on five tested programs is shown in Table 2. The data layouts proposed by our system always reach at least the performance of the AoS data layout. In the following paragraphs we give more details about three example test cases. For all charts we use AoS as a baseline and report the speedup of four transformed versions: SoA, the version generated after applying the *KDLG* algorithm and splitting the structure if applicable, a tiled AoS version were we use the tile-size proposed by our hardware dependent decision tree-based algorithm, and the final result of our framework as described in Section 3.3.

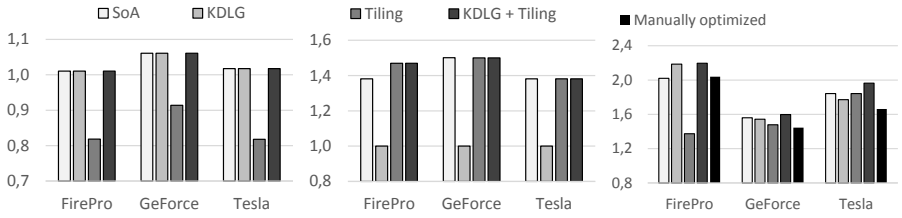
The first test case that we used to evaluate our framework is N-body, which performs a direct summation of the forces of all particles on every other particle. The struct in the used implementation consists of two fields with a size of 16 bytes each. As those fields have a big memory distance, the *KDLG*-based algorithm will split those fields into separate structs. Therefore, the result after applying the *KDLG*-based optimization is the same as when using the SoA layout. Furthermore, applying our tiling algorithm after the *KDLG*-based algo-

Table 2: Test programs

Test codes	struct size bytes	struct size fields	affected kernels	loop bound approx.[a]	speedup over AoS FirePro	speedup over AoS GeForce	speedup over AoS Tesla
N-body	32	2	1	n	1.01	1.06	1.01
BlackScholes [2]	28	7	1	–	1.00	1.43	2.83
Bitonic sorting	16	4	1	u	1.47	1.50	1.38
LavaMD [10]	36	3	1	c,u	2.22	1.89	2.07
SAMPO	48	12	9	w,u	2.19	1.59	1.96

[a] Used Loop bound approximations Section 3.1: loop over all work-items in the NDRange (n), over the work-group size (g), with constant boundaries (c), with unknown boundaries (u).

rithm has no effect. The speedup achieved is shown in Figure 5a. It clearly shows that the tiled version of the program is not only slower than the one in SoA data layout, but also slower than our baseline implementation. This applies to all tile-sizes we evaluated. However, since our framework uses a combination of two layout optimizations, it still correctly selects SoA, which is the data layout with the highest performance for this program on all tested GPUs.



(a) N-body (b) Bitonic sorting (c) SAMPO

Fig. 5: Speedup over AoS implementation on two example applications using different data layouts.

Bitonic Sort [1] is a sorting algorithm optimized for massively parallel hardware such as GPUs. The implementation that we are using sorts a struct of four elements, where the first element acts as the sorting-key. As all elements are always moved together, the *KDLG*-based algorithm results in one single cluster for any $\epsilon \geq 4$. Therefore, the version generated by the *KDLG*-based algorithm is the same as AoS. The decision-tree-based tiling algorithm converts this layout into a tiled-AoS layout with a tile-size of 512 bytes for the FirePro and GeForce while it suggests to use SoA on the Tesla. The results can be found in Figure 5b. It clearly shows that, although the *KDLG*-based algorithm fails to gain any improvement over AoS, the decision-tree-based algorithm as well as

the combination of both algorithm exceeds the performance of the AoS based implementation by a factor of 1.38 to 1.5. Furthermore, it delivers performance that is comparable or superior than the one achieved by a SoA implementation.

SAMPO [7] is an agent-based mosquito point model in OpenCL, which is designed to simulate mosquito populations to understand how vector-borne illnesses (e.g., malaria) may spread. The version available online is already manually optimized for AMD GPUs. Therefore, we ported this version to a pure AoS layout, where each agent is represented by a single struct with twelve fields. The measurements are displayed in Figure 5c. The results clearly show, that SoA yields a much better performance than AoS on all tested GPUs. Applying the *KDLG*-based algorithm already results in a speedup between 1.54 and 2.18 on the three tested GPUs, which is within ±10% of the SoA version.Applying tiling to the AoS implementation shows good results on the NVIDIA GPUs. Also the AMD GPU benefits from tiling, but it does not reach the performance of the SoA version or the version optimized with the *KDLG*-based algorithm. Applying tiling to the latter version further increases the performance on all evaluated GPUs and leads to a speedup over the AoS version of 2.19, 1.59 and 1.96 on the FirePro, GeForce and Tesla, respectively outperforming any other version we tested. Even the manually optimized implementation is outperformed by 7%, 10% and 18% on the FirePro, GeForce and Tesla, respectively.

5 Discussion

The results clearly show, that programs with AoS data layout are not well suited for GPUs. SoA delivers a much higher performance on all GPU/program combinations we tested. However, also SoA fails to achieve the maximum performance in some cases. We observed that a tiled-AoS can achieve results that are usually equal or better compared to the ones achieved with an SoA layout. But tiled-AoS is not suited for all programs. Similarly, splitting structures in several smaller ones based on a *KDLG* is beneficial for most programs. However, also this technique fails to improve the performance of some applications . Therefore, combining these two algorithms leads to much better overall results than each of them can achieve individually. This is underlined by the results of SAMPO, where the combination of both algorithms does not only outperform the results of each algorithm applied individually, but also leads to higher performance than obtained by both, a SoA layout and the manually optimized layout.

6 Conclusions

We presented a system to automatically determine an improved data layout for OpenCL programs.Our framework consists of two separate algorithms: The first one constructs a *KDLG*, which is used to split a given struct into several clusters based on the hardware dependent parameter ϵ. The second algorithm constructs a decision tree, which is used to determine the tile-size for a certain structure when converting it from AoS to tiled-AoS or SoA layouts.

The combination of both algorithms is crucial, as using only one of them often leads to no improvement over AoS. The layouts proposed by our framework result in speedups of up to 2.22, 1.89 and 2.83 on an AMD FirePro S9000, NVIDIA GeForce GTX 480 and NVIDIA Tesla k20m, respectively.

Acknowledgments This project was funded by the FWF Austrian Science Fund as part of project I 1523 "Energy-Aware Autotuning for Scientific Applications" and by the Interreg IV Italy-Austria 5962-273 EN-ACT funded by ERDF and the province of Tirol.

References

1. Batcher, K.E.: Sorting networks and their applications. In: Proc. of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 307–314. AFIPS'68 (Spring), ACM, New York, NY (1968)
2. Black, F., Scholes, M.: The pricing of options and corporate liabilities. The Journal of Political Economy 81 (1973)
3. Che, S., Meng, J., Skadron, K.: Dymaxion++: a Directive-Based API to Optimize Data Layout and Memory Mapping for Heterogeneous Systems. AsHes'14 (2014)
4. Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: Optimizing memory access patterns for heterogeneous systems. In: SC'11. pp. 13:1–13:11. ACM, New York, NY (2011)
5. Kandemir, M., Choudhary, A., Ramanujam, J., Banerjee, P.: A framework for interprocedural locality optimization using both loop and data layout transformations. In: Proc. of the Int. Conference on Parallel Processing. pp. 95–102 (1999)
6. Khronos Group: OpenCL 1.2 Specification (Apr 2012)
7. Kofler, K., Davis, G., Gesing, S.: Sampo: An agent-based mosquito point model in opencl. In: ADS'14. pp. 5:1–5:10. Society for Computer Simulation International, San Diego, CA (2014)
8. Kruskal, J.B.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proc. of the American Mathematical Society 7(1), 48–50 (1956)
9. Raman, E., Hundt, R., Mannarswamy, S.: Structure layout optimization for multithreaded programs. In: CGO'07. pp. 271–282. IEEE Computer Society, Washington, DC (2007)
10. Rodinia: LavaMD (Nov 2014), http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/LavaMD
11. Rubin, S., Bodík, R., Chilimbi, T.: An efficient profile-analysis framework for data-layout optimizations. In: POPL'02. pp. 140–153. ACM, New York, NY (2002)
12. RULEQUEST RESEARCH: Data mining tools see5 and c5.0 (Oct 2014), https://www.rulequest.com/see5-info.html
13. Stratton, J.A., Rodrigues, C.I., Sung, I.J., Chang, L.W., Anssari, N., Liu, G.D., mei W. Hwu, W., Obeid, N.: Algorithm and data optimization techniques for scaling to massively threaded systems. IEEE Computer 45(8), 26–32 (2012)
14. Strzodka, R.: Data layout optimization for multi-valued containers in opencl. J. Parallel Distrib. Comput. 72(9), 1073–1082 (2012)
15. Sung, I.J., Anssari, N., Stratton, J.A., mei W. Hwu, W.: Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. International Journal of Parallel Programming 40(1), 4–24 (2012)
16. Weber, N., Goesele, M.: Auto-tuning complex array layouts for gpus. In: Proc. of Eurographics Symposium on Parallel Graphics and Visualization. EGPGV14, EG