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

Auto-Tuning Complex Array Layouts for GPUs - Supplemental Material

Nicolas Weber and Michael Goesele

Graduate School of Computational Engineering, TU Darmstadt, Germany

1. Supplemental Material

This is the supplemental material for the paper "Auto-Tuning Complex Array Layouts for GPUs". We provided additional evaluation results for the KD-Tree Binning Example (Section 2), another representation of how the decision tree shown in the paper looks in parameter space (Section 3) as well as the complete source code of an application using CUDA Driver API compared to the same source code implemented using MATOG (Section 4).

2. KD-Tree Binning Example

Figure 1 shows the speedup of the learned solution over the optimal speedup. To obtain these results, we ran the application with all test cases and all possible memory layouts. With these results, we have been able to determine an optimal memory layout for each test case. Then we compared these optimal results with the memory layout that MATOG chose. The highlighted area indicates the area between being faster than the baseline and the optimal speedup. Our maximal speedup for the GTX680 is 9.83, while the complete mode has an average speedup of 2.47 and the small mode of 2.39. If we would have always selected the best solution, our average speedup would have been 2.99. For the GTX570, our maximal speedup is 4.78, while the complete mode has an average speedup of 2.12 and the small mode of 2.28. The average of all optimal solutions for the GTX570 is 2.51.

3. Decision Tree Example

Figure 2 shows the decision tree for choosing the best global memory layout for storing the axis aligned bounding boxes (AABBs) of the triangles in the KD-Tree Binning kernel example as a tree. Figure 3 shows the tree as flattened parameter space. The points in the this figure represent the test samples for the four used scenes, the used bin count and the resulting best memory layout. Each cell represents a memory layout that is assumed to work best for the given arguments. E.g., an execution using a model with 200,000 triangles and 320 bins would select AoSoA as layout. As already mentioned, the decision boundaries are always located in the center between two test samples. As stated in the paper, for the Bitonic Sort example only SoA is chosen for the best layout. Therefore the tree only consists of one node. The layouts chosen for the REYES example are shown in Table 1.

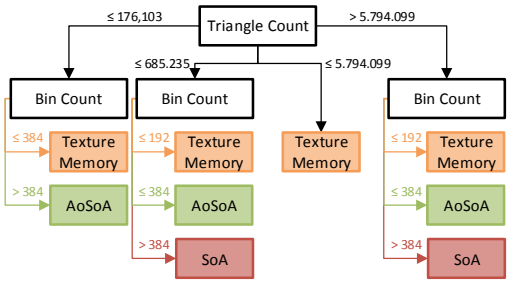

Figure 2: *Tree representation of decision tree*

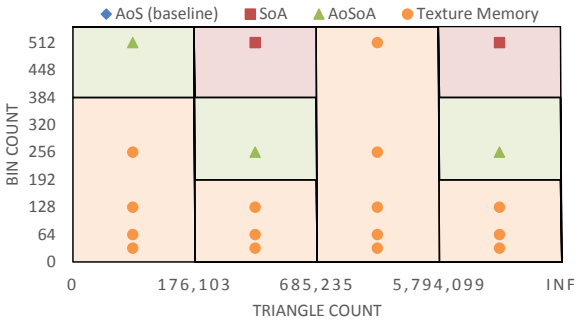

Figure 3: *Parameter space representation of decision tree*

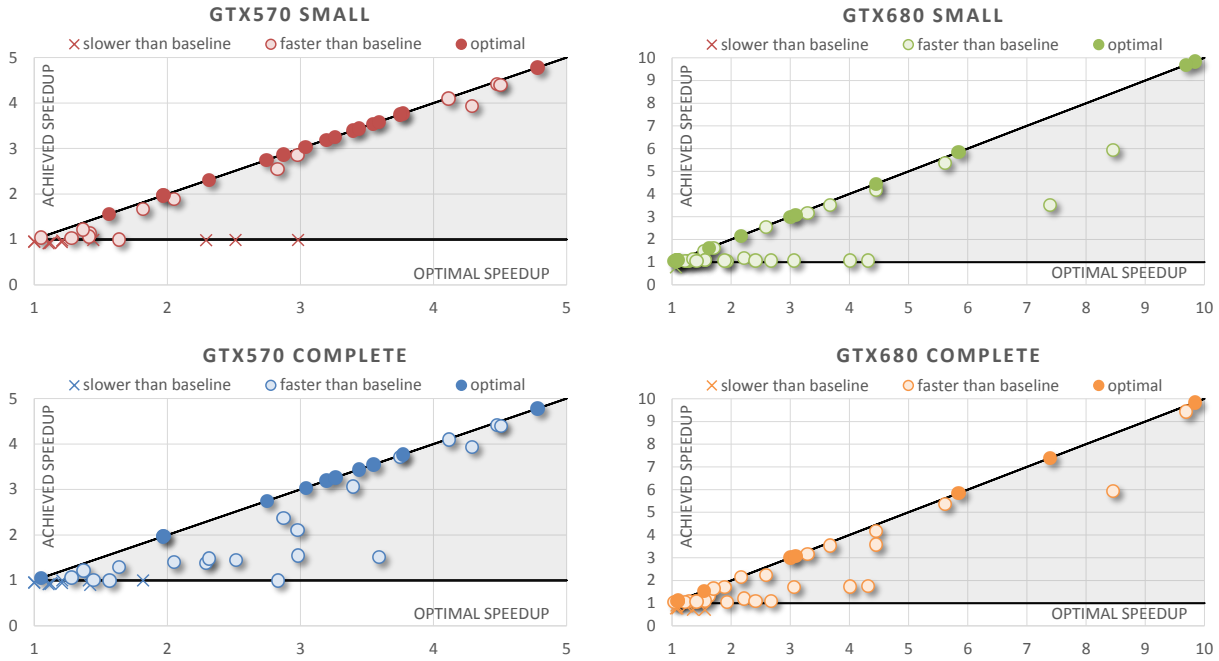

Figure 1: *Evaluation results for the KD-Tree Binning example for GTX570 and GTX680 containing all 40 test results.*

Variable	Complete	Small
Global Memory (all kernels)		
Prefer L1 Cache	false	false
Primitives	AoS	AoS
Z-Buffer	AoS	AoS
Texture	untransposed	untransposed
Projection Matrix	transposed	transposed
Bound and Split		
Prefer L1 Cache	false	false
Primitives	AoS	AoS
Projection Matrix	untransposed	untransposed
Compact		
Prefer L1 Cache	true	true
Dice and Shade		
Prefer L1 Cache	false	false
Control Points	SoA	AoS
Primitives	AoSoA	AoS
Projection Matrix	untransposed	untransposed
Triangles	AoS	AoS
Paint Texture		
Prefer L1 Cache	false	false

Table 1: *This table shows the chosen memory layouts for the REYES example for global and shared memory. Each kernel can use different layouts for the shared memory as well as preferring L1 cache or shared memory.*

4. MATOG Code Example

This section shows the changes that have to be applied to an application using CUDA Driver API to be ported to MATOG. We show the original source code on the left side, while showing the changed code on the right side. The application itself consists of five files.

We highlighted lines which differ in both codes. If the line marked magenta it has changed while if it is marked green it means that there is no corresponding line in the other code.

4.1. CMakeLists.txt

This file defines a CMake project, which is used to generate platform independent projects e.g. for Make or Visual Studio. MATOG uses CMake as well to compile its library and the kernels. That is why the MATOG variant does not require any CUDA_COMPILE_PTX calls, as these are done automatically by including the MATOG generated CMake project. Further the application has to be linked to the MATOG library.

Driver API

```
1  # minimum cmake version
2  CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
3
4  # project name
5  PROJECT("Example␣Driver␣API")
6
7  # find cuda
8  FIND_PACKAGE(CUDA)
9
10
11
12
13 # target
14 CUDA_ADD_EXECUTABLE(example main.cpp)
15 TARGET_LINK_LIBRARIES(example ${CUDA_CUDA_LIBRARY})
16 SET(CUDA_NVCC_FLAGS "${CUDA_NVCC_FLAGS}␣-arch=compute_20
       ␣--code=sm_20")
17
18 # compile ptx
19 CUDA_COMPILE_PTX(PTX_FILE "module.cu")
20 CUDA_ADD_LIBRARY(module "module.cu" ${PTX_FILE})
21
22 # rename ptx file after compilation
23 GET_FILENAME_COMPONENT(FILENAME ${PTX_FILE} NAME)
24 STRING(REGEX REPLACE "^cuda_compile_ptx_generated_" ""
       FILENAME ${FILENAME})
25 STRING(REGEX REPLACE ".cu.ptx" ".ptx" FILENAME ${
       FILENAME})
26
27 # create PTX folder
28 EXECUTE_PROCESS(COMMAND ${CMAKE_COMMAND} -E
       make_directory "ptx")
29
30 # move to build/ptx folder
31 ADD_CUSTOM_COMMAND(TARGET module PRE_LINK COMMAND ${
       CMAKE_COMMAND} -E rename ${PTX_FILE} "ptx/${
       FILENAME}")
```

MATOG

```
1  # minimum cmake version
2  CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
3
4  # project name
5  PROJECT("Example␣MATOG")
6
7  # find cuda
8  FIND_PACKAGE(CUDA)
9
10 # include MATOG lib
11 INCLUDE("CMakeLists_myLib.txt")
12
13 # target
14 CUDA_ADD_EXECUTABLE(example main.cpp)
15 TARGET_LINK_LIBRARIES(example ${CUDA_CUDA_LIBRARY} ${
       MATOG_LIBRARIES} myLib)
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

4.2. Matog.xml

This Matog.xml is used by the MATOG library generator. It defines the used data structures and GPU code files. In this example we use an Array of Structs with two integer and one float field. We further declare this type to be shared, so that a implementation for the shared memory is generated as well. The file *module.cu* is the only CUDA module we are using.

Driver API

```

```

MATOG

```
<matog version="1.0">
    <cuda mincc="2.0" />
    <cmake libname="myLib">
        <kernels>module.cu</kernels>
    </cmake>
    <code>
        <struct name="Data" shared="true">
            <field name="a" type="int" />
            <field name="b" type="int" />
            <field name="c" type="float" />
        </struct>
    </code>
</matog>
```

4.3. DataFormat.h

The file DataFormat.h contains the struct definition, which is used by the host and the device. It is not necessary for the usage with MATOG, as the library generator provides headers to access the data.

Driver API

```
struct Data {
  int a;
  int b;
  float c;
};
```

MATOG

```

```

4.4. Module.cu

This file contains the GPU implementation of our example. In contrast to the existing implementation, four changes are necessary. The first one is to include the generated implementation. The second change is, that MATOG does not use pointers to pass data to the kernel but a real object instance. Further the initialization of the shared memory is performed by a template. The last change is, that MATOG does not allow to directly access the struct itself, and therefore it is not possible to copy data from global to shared memory by copying the array positions. Therefore MATOG has a special copy method, which takes care of this process.

Driver API

```
1  #include <cuda.h>
2  #include "DataFormat.h"
3
4  extern "C" {
5      __global__ void function(Data* data) {
6          // define shared
7          __shared__ Data shared[128];
8
9          // copy to shared
10         shared[threadIdx.x] = data[threadIdx.x +
               blockIdx.x * blockDim.x];
11
12         // sync
13         __syncthreads();
14
15         // calculate something
16         for(int i = 0; i < threadIdx.x; i++)
17         {
18             shared[threadIdx.x].c = shared[threadIdx.x].
                   c + shared[threadIdx.x].a / (float)
                   shared[i].b;
19         }
20
21         // copy to global
22         data[threadIdx.x + blockIdx.x * blockDim.x].c =
               shared[threadIdx.x].c;
23     }
24 }
```

MATOG

```
1  #include <cuda.h>
2  #include "Data.cu"
3
4  extern "C" {
5      __global__ void function(Data data) {
6          // define shared
7          __shared__ DataShared<128> shared;
8
9          // copy to shared
10         shared.copyToShared(data, blockIdx.x * blockDim.
               x);
11
12         // sync
13         __syncthreads();
14
15         // calculate something
16         for(int i = 0; i < threadIdx.x; i++)
17         {
18             shared[threadIdx.x].c = shared[threadIdx.x].
                   c + shared[threadIdx.x].a / (float)
                   shared[i].b;
19         }
20
21         // copy to global
22         data[threadIdx.x + blockIdx.x * blockDim.x].c =
               shared[threadIdx.x].c;
23     }
24 }
```

4.5. Main.cpp

This file contains the host implementation of the application. There are some minor changes necessary. First of all, the MATOG and the data structure header have to be included. Further the CUDA module and function have to be loaded using the MATOG function *loadFunction* instead of the Driver API calls. The array itself has to be instantiated as a class and not as an array. As MATOG does not allow direct access to the GPU data from the host, it is necessary to use the special copy methods instead of the *cuMemcpyXtoX* methods. For the arguments passed to the kernel itself, MATOG requires two changes. The first one is, that we have a so called GPUObject, which is a class instance, which can directly be used by the kernel itself. It can be created using the function *getGPUObject*. Further we require the argument list to be zero terminated. The reason for this is, that CUPTI does not tell how many arguments are passed to the kernel but as we read the argument list during our optimization, we need some kind of list terminator. The kernel call itself and the memory access does not have to be changed.

Driver API

```
1  #include <cuda.h>
2  #include <stdio.h>
3  #include "DataFormat.h"
4
5
6
7  int main(int argc, char** argv) {
8      // init driver api
9      cuInit(0);
10
11     // get device
12     CUdevice device;
13     cuDeviceGet(&device, 0);
14
15     // create context
16     CUcontext context;
17     cuCtxCreate(&context, 0, device);
18
19     // load module
20     CUmodule module;
21     cuModuleLoad(&module, "ptx/module.ptx");
22
23     // load function
24     CUfunction function;
25     cuModuleGetFunction(&function, module, "function");
26
27     // init host data
28     Data* host = new Data[1024];
29
30     for(int i = 0; i < 1024; i++) {
31         host[i].a = i;
32         host[i].b = 1023 - i;
33         host[i].c = 0;
34     }
35
36     // init device data
37     CUdeviceptr data;
38     cuMemAlloc(&data, 1024 * sizeof(Data));
39
40     // copy data
41     cuMemcpyHtoD(data, host, 1024 * sizeof(Data));
42
43     // prepare arguments
44
45     void* args[] = {&data};
46
47     // execute kernel
48     cuLaunchKernel(function, 8, 1, 1, 128, 1, 1, 0, 0,
           args, 0);
49
50     // sync
51     cuCtxSynchronize();
52
53     // copy data
54     cuMemcpyDtoH(host, data, 1024 * sizeof(Data));
```

MATOG

```
1  #include <cuda.h>
2  #include <stdio.h>
3  #include "Data.h"
4  #include <Matog.h>
5  using matog::Matog;
6
7  int main(int argc, char** argv) {
8      // init driver api
9      cuInit(0);
10
11     // get device
12     CUdevice device;
13     cuDeviceGet(&device, 0);
14
15     // create context
16     CUcontext context;
17     cuCtxCreate(&context, 0, device);
18
19     // load module + function
20     CUmodule module;
21
22
23
24     CUfunction function;
25     Matog::loadFunction("module", "function", module,
           function);
26
27     // init host data
28     Data& host = *new Data(1024);
29
30     for(int i = 0; i < 1024; i++) {
31         host[i].a = i;
32         host[i].b = 1023 - i;
33         host[i].c = 0;
34     }
35
36
37
38
39
40     // copy data
41     host.copyHostToDevice();
42
43     // prepare arguments
44     Data::GPUObject obj = host.getGPUObject();
45     void* args[] = {&obj, 0};
46
47     // execute kernel
48     cuLaunchKernel(function, 8, 1, 1, 128, 1, 1, 0, 0,
           args, 0);
49
50     // sync
51     cuCtxSynchronize();
52
53     // copy data
54     host.copyDeviceToHost();
```

```
56      // print
57        for(int i = 0; i < 1024; i++) {
58            printf("%4i: %f\n", i, host[i].c);
59        }
60
61      // free
62        delete[] host;
63        cuMemFree(data);
64
65        // return
66        return 0;
67    }
```

```
56      // print
57        for(int i = 0; i < 1024; i++) {
58            printf("%4i: %f\n", i, host[i].c);
59        }
60
61        // free
62        delete &host;
63
64
65        // return
66        return 0;
67    }
```