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

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

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**

```cmake
# minimum cmake version
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

# project name
PROJECT( "Example Driver API" )

# find cuda
FIND_PACKAGE(CUDA)

# target
CUDA_ADD_EXECUTABLE(example main.cpp )
TARGET_LINK_LIBRARIES(example ${CUDA_CUDA_LIBRARY} )
SET(CUDA_NVCC_FLAGS " ${CUDA_NVCC_FLAGS} -arch=compute_20 -code=sm_20 " )
CUDA_COMPILE_PTX(PTX_FILE "module.cu")
CUDA_ADD_LIBRARY(module "module.cu" ${PTX_FILE} )
GET_FILENAME_COMPONENT(FILENAME ${PTX_FILE} NAME)
STRING(REPLACE " ^cuda_compile_ptx_generated." " " FILENAME ${FILENAME} )
EXECUTE_PROCESS(COMMAND $<MAKE,CMDNAME> -E make_directory "pix")
# move to build/pix folder
ADD_CUSTOM_COMMAND(TARGET module PRE_LINK CMDNAME $<MAKE,CMDNAME> -E rename $<PTX_FILE> "pix/" $<FILENAME> )
```

**MATOG**

```cmake
# minimum cmake version
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

# project name
PROJECT( "Example MATOG" )

# find cuda
FIND_PACKAGE(CUDA)

# include MATOG lib
INCLUDE(" CMakeLists_myLib.txt")

# target
CUDA_ADD_EXECUTABLE(example main.cpp )
TARGET_LINK_LIBRARIES(example ${CUDA_CUDA_LIBRARY} ${MATOG_LIBRARIES} myLib )
```

© The Eurographics Association 2014.
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**

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

**MATOG**

```xml
<matog version="1.0">
    <cuda mincc="2.0"/>
    <make libname="myLib">
        <kernels>module.cu</kernels>
    </make>

    <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**

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

**MATOG**

```xml
<matog version="1.0">
    <cuda mincc="2.0"/>
    <make libname="myLib">
        <kernels>module.cu</kernels>
    </make>

    <code>
        <struct name="Data" shared="true">
            <field name="a" type="int"/>
            <field name="b" type="int"/>
            <field name="c" type="float"/>
        </struct>
    </code>
</matog>
```
4.4. 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**

```c
#include <cuda.h>
#include "DataFormat.h"

extern "C" {

__global__ void function(Data* data) {
    // define shared
    __shared__ Data shared[128];
    // copy to shared
    shared[threadIdx.x] = data[threadIdx.x + blockIdx.x * blockDim.x];
    // sync
    __syncthreads();
    // calculate something
    for(int i = 0; i < threadIdx.x; i++)
    {
        shared[threadIdx.x].c = shared[threadIdx.x].a / (float) shared[i].b;
    }
    // copy to global
    data[threadIdx.x + blockIdx.x * blockDim.x].c = shared[threadIdx.x].c;
}
}
```

**MATOG**

```c
#include <cuda.h>
#include "Data.cu"

extern "C" {

__global__ void function(Data data) {
    // define shared
    __shared__ DataShared<128> shared;
    // copy to shared
    shared.copyToShared(data, blockIdx.x * blockDim.x);
    // sync
    __syncthreads();
    // calculate something
    for(int i = 0; i < threadIdx.x; i++)
    {
        shared[threadIdx.x].c = shared[threadIdx.x].a / (float) shared[i].b;
    }
    // copy to global
    data[threadIdx.x + blockIdx.x * blockDim.x].c = shared[threadIdx.x].c;
}
```
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**

```c
#include <cuda.h>
#include <cuda.h>

cuMemcpyHtoD(data, host, 1024);
cuCtxCreate(&context, 0, device);
// create context
// init driver api
#include <stdio.h>

cuCtxCreate(&context, 0, device);
Data
// load module
// init device data
void* args[] = {&data};
// execute kernel
cuLaunchKernel(function, 8, 1, 1, 128, 1, 1, 0, 0, args, 0);
// sync
cuCtxSynchronize();
// copy data
cuMemcpyDtoH(host, data, 1024 * sizeof(Data));
```

**MATOG**

```c
#include <cuda.h>
#include <cuda.h>

cuMemcpyHtoD(data, host, 1024);
cuCtxCreate(&context, 0, device);
// create context
// init driver api
#include <stdio.h>

cuCtxCreate(&context, 0, device);
Data
// load module + function
CUmodule module;

CUfunction function;
cuModuleGetFunction(&function, module, "function");
for (int i = 0; i < 1024; i++) {
    host[i].a = i;
    host[i].b = 1023 - i;
    host[i].c = 0;
}

// init device data
CUdevice device;

matog::Matog;
daDevice device;
cuDeviceGet(&device, 0);
cuDeviceGet(&device, 0);
cuLaunchKernel(function, 8, 1, 1, 128, 1, 1, 0, 0,

args, 0);
// sync
cuCtxSynchronize();
// copy data
host copyingToDeviceToDevice();

// prepare arguments
Data; GPUObject obj = host.getGPUObject();
void* args[] = {&obj};
// execute kernel
cuLaunchKernel(function, 8, 1, 1, 128, 1, 1, 0, 0, args, 0);
// sync
cuCtxSynchronize();
// copy data
host.copyDeviceToHost();
```
// print
for(int i = 0; i < 1024; i++) {
    printf("%4i : %f\n", i, host[i].c);
}
// free
delete[] host;
cuMemFree(data);
// return
return 0;

// print
for(int i = 0; i < 1024; i++) {
    printf("%4i : %f\n", i, host[i].c);
}
// free
delete[] host;
cuMemFree(data);
// return
return 0;