CUDA Driver API

Alexey A. Romanenko
arom@ccfit.nsu.ru
Novosibirsk State University

* Is it possible to distribute kernel separately?
* Is it possible to launch kernel without C/C++ extensions?
* Is it possible to program GPU without C/C++?

CUDA-objects in driver API

* **Device** — CUDA-compatible device
* **Context** — same as process for CPU
* **Module** — same as dynamic library
* **Function** — kernel
* **Heap memory** — pointer to device memory
* **CUDA Array** — container for 1D or 2D arrays
* **Texture reference** — object to describe texture object

Workflow

* Init driver
* Select device (GPU)
* Create context
* Work within context
  * Kernel in PTX or CUBIN cormat
* Destroy context

Initialize driver

* CUresult cuInit(unsigned int flag);
  * Flag = 0
* CUT_DEVICE_INIT_DRV(cuDevice, ARGC, ARGV)
* Working with non-initialized driver result in CUDA_ERROR_NOT_INITIALIZED

Device management (1)

* CUresult **cuDeviceGetCount**(int *count)
* CUresult **cuDeviceGet**(CUdevice *device, int ordinal)
* CUresult **cuDeviceComputeCapability**(int *major, int *minor, CUdevice dev)
* CUresult **cuDeviceTotalMem**(unsigned int *bytes, CUdevice dev)
* CUresult **cuDeviceGetAttribute**(int *pi, CUdevice_attribute attrib, CUdevice dev)

Device attributes (1)

* CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
* CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X
* CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y
* CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z
* CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X
* CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y
* CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z
* CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
* CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY
* CU_DEVICE_ATTRIBUTE_WARP_SIZE
* CU_DEVICE_ATTRIBUTE_MAX_PITCH

Device attributes (2)

* CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
* CU_DEVICE_ATTRIBUTE_CLOCK_RATE
* CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT
* CU_DEVICE_ATTRIBUTE_GPU_OVERLAP
* CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT
* CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT
* CU_DEVICE_ATTRIBUTE_INTEGRATED
* CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY
* CU_DEVICE_ATTRIBUTE_COMPUTE_MODE
  * CU_COMPUTEMODE_DEFAULT
  * CU_COMPUTEMODE_EXCLUSIVE
  * CU_COMPUTEMODE_PROHIBITED
* etc.

Device management (2)

* CUresult cuDeviceGetProperties (CUdevprop *prop, CUdevice dev)

```
typedef struct CUdevprop_st {
        int maxThreadsPerBlock;
        int maxThreadsDim[3];
        int maxGridSize[3];
        int sharedMemPerBlock;
        int totalConstantMemory;
        int SIMDWidth;
        int memPitch;
        int regsPerBlock;
        int clockRate;
        int textureAlign
} CUdevprop;
```

CUDA context

* CUDA context — same as process for CPU
* CPU thread has only one active CUDA context
* Creating context (cuCtxCreate) initiate context usage counter to 1
* **cuCtxAttach**()* increase counter, **cuCtxDetatch**()* decrease counter
* Context is destroyed when counter = 0 or user called **cuCtxDestroy**()
* User can change active context. **cuCtxPopCurrent**(), **cuCtxPushCurrent**()

* - Deprecated in CUDA 4.0

CUDA module (1)

* Module — Dynamically loaded library with kernels.
* Module is build with nvcc.
* Could be distributed separatelly
  * nvcc --keep
* CUmodule cuModule;
  cuModuleLoad(&cuModule, "module.cubin");
  CUfunction cuFunc;
  cuModuleGetFunction(&cuFunc, cuModule,
  "myKernel");

CUDA module (2)

```
#define ERROR_BUFFER_SIZE 100
CUmodule cuModule;
CUjit_option options[3];
void* values[3];
char* PTXCode = "some PTX code";
options[0] = CU_ASM_ERROR_LOG_BUFFER;
values[0] = (void*)malloc(ERROR_BUFFER_SIZE);
options[1] = CU_ASM_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1] = (void*)ERROR_BUFFER_SIZE;
options[2] = CU_ASM_TARGET_FROM_CUCONTEXT;
values[2] = 0;
cuModuleLoadDataEx(&cuModule, PTXCode, 3,
                        options, values);
for (int i = 0; i < values[1]; ++i) {
   // Parse error string here
}
```

Module management (2)

- **cuModuleLoad()** - load module from cubin file
- **cuModuleLoadData()** – load module from PTX string
- **cuModuleLoadDataEx()** - load module from PTX string, return compiling result/errors
- **cuModuleLoadFatBinary()** – load module from fat cubin file
  - Available from CUDA 4.0
  - nvcc -fatbin
- **cuModuleUnload()** – unload module

Execution control

* Define Thread block shape
* Define Grid shape
* Set kernel parameters
* Set shared memory configuration

* Launch kernel

Execution control

* cuFuncSetBlockShape()
* cuFuncSetSharedSize()
* cuLaunch()
* cuLaunchGrid()
* cuLaunchGridAsync()

Execution control

* cuFuncSetBlockShape()
* cuFuncSetSharedSize()
* cuLaunch()
* cuLaunchGrid()
* cuLaunchGridAsync()

CUDA 3.2 and older

CUDA 4.0

```
cuLaunchKernel (
        function,
        gDimX, gDimY, gDimZ,
        bDimX, bDimY, bDimZ,
        sharedMemBytes,
        Stream,
        kernelParams, extra
    )
```

Setting kernel parameters

* cuParamSetf ()
* cuParamSeti()
* cuParamSetSize ()
* cuParamSetTexRef()
* cuParamSetv()

Setting kernel parameters

* ~~cuParamSetf ()~~
* ~~cuParamSeti()~~
* ~~cuParamSetSize ()~~
* ~~cuParamSetTexRef()~~
* ~~cuParamSetv()~~

CUDA 3.2 and older

CUDA 4.0

Last 2 parameters of cuLaunchKernel

```
(...
    kernelParams, extra
)
```

kernelParams — pointers to kernel parameters

extra — packed kernel parameters

Setting kernel parameters

* std::vector< void* > kernelParams;
  float *dev_in1; float * dev_in2; float *dev_out;

  kernelParams.push_back(&dev_in1);
  kernelParams.push_back(&dev_in2); kernelParams.push_back(&dev_out);
  kernelParams.push_back(const_cast< int*>(&VEC_SIZE));

* const size_t sharedMemSize = 0;
  const CUstream stream = 0;
* *// equivalent to*
  // vecSum<<<GS, BS, 0, 0>>>(dev_in1, dev_in2, dev_out, VEC_SIZE);
* status = cuLaunchKernel(function,
  GS.x, GS.y, GS.z, BS.x, BS.y, BS.z,
* sharedMemSize, stream, &kernelParams[0], 0);

Setting kernel parameters

```
#define ALIGN_UP(offset, alignment) (offset) = ((offset)+(alignment) - 1) & ~((alignment) - 1)
char paramBuffer[1024];
size_t paramBufferSize = 0;
#define ADD_TO_PARAM_BUFFER(value, alignment) { \
                paramBufferSize = ALIGN_UP(paramBufferSize, alignment); \
                memcpy(paramBuffer + paramBufferSize, &(value), sizeof(value)); \
                paramBufferSize += sizeof(value); }
CUdeviceptr dev_in1, dev_in2, dev_out;
ADD_TO_PARAM_BUFFER(dev_in1, __alignof(dev_in1));
ADD_TO_PARAM_BUFFER(dev_in2, __alignof(dev_in2));
ADD_TO_PARAM_BUFFER(dev_out, __alignof(dev_out));
ADD_TO_PARAM_BUFFER(VEC_SIZE, __alignof(VEC_SIZE));
void *config[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, paramBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE,    &paramBufferSize,
    CU_LAUNCH_PARAM_END
};
status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL, config);
```

Memory management

* CUresult **cuMemAlloc** (CUdeviceptr *dptr, unsigned int size)
* CUresult **cuMemAllocHost** (void **pp, unsigned int bytesize)
* CUresult **cuMemAllocPitch** (CUdeviceptr *dptr, unsigned int *pPitch, unsigned int WidthInBytes, unsigned int Height, unsigned int ElementSizeBytes)
* CUresult **cuMemFree** (CUdeviceptr dptr)
* CUresult **cuMemFreeHost** (void *p)
* CUresult **cuMemcpy*** - copying data from/to device

Texture management

* cuTexRefCreate()
* cuTexRefDestroy()
* cuModuleGetTexRef()
* cuTexRefSetAddress()
* cuTexRefSetArray()
* cuTexRefSetFilterMode()
* cuTexRefSetAddressMode()
  * CU_TR_ADDRESS_MODE_WRAP,
  * CU_TR_ADDRESS_MODE_CLAMP,
  * CU_TR_ADDRESS_MODE_MIRROR,
  * CU_TR_ADDRESS_MODE_BORDER

} Not documented

* cuTexRefSetFlags()

CUDA driver API vs. runtime API

* Runtime API based on driver API
* Runtime API works with a context, created with driver API. If there is no context, it is created implicitly.
* Driver API is more flexible
  * More information about device, for example, free memory (cuMemGetInfo)
  * etc.
* Using driver API
  * Kernel is not linked to program
  * Writing and debugging process become complicated.