CUDA Programming

Week 5. Asynchronized execution, Instructions, and CUDA driver API

Outline

- Asynchronized Transfers
- Instruction optimization
- CUDA driver API
- Homework

ASYNCHRONIZED TRANSFER

Asynchronous transfers

- `cudaMemcpy`: blocking transfer
  - The control will not return to the main thread until the memory copy is complete.
- `cudaMemcpyAsync`: nonblocking transfer
  - Return to the main thread immediately
- Two requirements:
  - Use *pinned host memory*.
  - Contains an additional argument, a *stream ID*.

Pinned memory

- Pinned memory transfers attain the highest bandwidth between host and device.
  - On PCIe ×16 Gen2 cards, pinned memory can attain greater than 5 GBps transfer rates.
- Pinned memory is allocated using the `cudaMallocHost`()
- Should not be overused. Excessive use can reduce overall system performance

Example of using pinned memory

```
//allocate host memory
if( PINNED == memMode )    {
   //pinned memory mode - get OS-pinned memory
   cudaMallocHost((void**)&h_idata, memSize );
   cudaMallocHost((void**)&h_odata, memSize );
}  else {
   //pageable memory mode - use malloc
   h_idata = (unsigned char *)malloc( memSize );
   h_odata = (unsigned char *)malloc( memSize );
}
//initialize the memory
for(int i=0;i<memSize/sizeof(unsigned char);i++){
   h_idata[i] = (unsigned char) (i & 0xff);
}
```

Stream

- A stream is a sequence of operations that are performed in order on the device.
  - Operations in different streams can be interleaved and overlapped, which can be used to hide data transfers between host and device.
- Use `cudaStreamCreate`() to create a stream
  - The default stream (ID=0) need not be create

Example 1: using default stream

```
cudaMemcpyAsync(ad,ah,size,
         cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
//cpu functions...
```

- The first two calls returns immediately
- The kernel function uses the default stream 0
- The kernel function will not be executed until the memory copy is done (they are in the same stream)

Example 2: two streams

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size,
                    cudaMemcpyHostToDevice, stream1);
kernel<<<grid,block,0,stream2>>>(otherData_d);
```

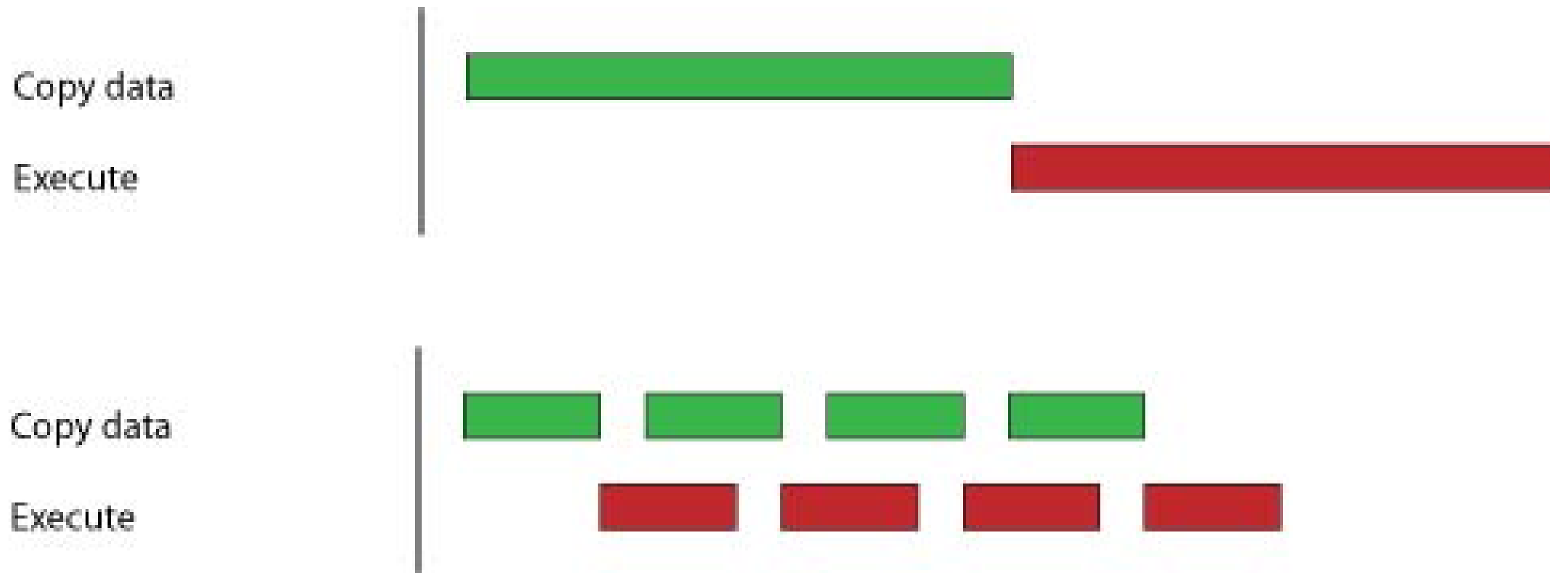
- Kernel execution with asynchronous data transfer.
- The kernel function can be executed while the memory copy is in progress

Example 3:

```
size=N*sizeof(float)/nStreams;
for (i=0; i<nStreams; i++) {
   offset = i*N/nStreams;
   cudaMemcpyAsync(a_d+offset,
     a_h+offset, size, dir, stream[i]);
}
for(i=0; i<nStreams; i++) {
   offset = i*N/nStreams;
   kernel<<<N/(nThreads*nStreams),
   nThreads,0,stream[i]>>>(a_d+offset);
}
```

Comparing to the traditional way

```
cudaMemcpy(a_d, a_h, N*sizeof(float), dir);
kernel<<<N/nThreads, nThreads>>>(a_d);
```



INSTRUCTION OPTIMIZATION

Arithmetic Instructions

- Single-precision (SP) floats
  - SP add, multiply, and multiply-add are 8/cycle.
  - SP reciprocal, reciprocal square root, and __logf(x) are 2/cycle.
- Integer
  - 32-bit integer multiplication is 2/cycle.
  - __mul24 and __umul24 *provide signed and unsigned 24-bit integer multiplication with a throughput of 8/cycle*. (not support in future)

Math Libraries

- Two types of runtime math operations: `__functionName()` and `functionName()`.
- `__functionName()` maps directly to the hardware level. Faster but lower accuracy.
  - 1/cycle
- `functionName()`: slower but higher accuracy.
- The `-use_fast_math` compiler option of `nvcc` converts to fast version automatically

Double and other math functions

- Double is much slower than float
  - Avoid automatic conversion of double to floats
  - EX: use f suffix: 3.141592653589793f
- Use expf2, expf10, exp2, exp10, instead of pow or powf
- Use reciprocal Square root: rsqrtf and rsqrt, instead of 1.0f/sqrtf()
  - Used for normalization

Memory Instructions

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

- 8/cycle to issue a read from global memory,
- 8/cycle to issue a write to shared memory,
- 400 to 600 clock cycles to read data from global memory.

Ref Programming Guide

CUDA DRIVER API

CUDA SDK

Libraries:FFT, BLAS,...
Example Source Code

Integrated CPU
and GPU C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing

CPU Host Code

CUDA
Driver

Debugger
Profiler

Standard C Compiler

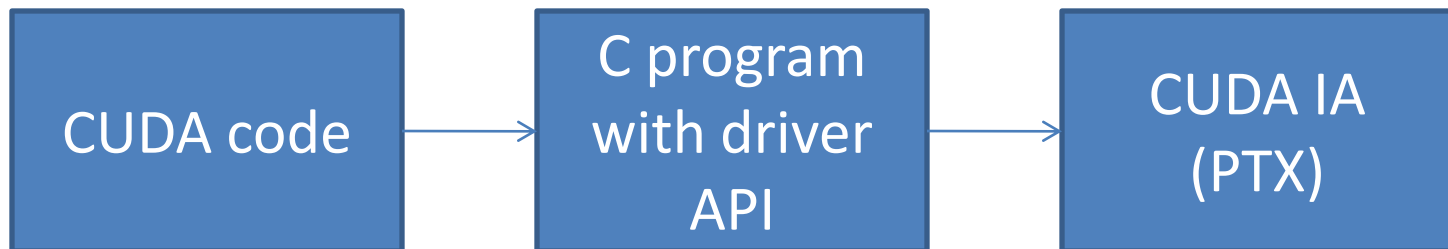
GPU

CPU

© NVIDIA Corporation 2007

CUDA driver API

- A handle-based, imperative API
- implemented in the **nvcuda** dynamic library
- All its entry points are prefixed with **cu.**



Objects in driver API

Object	Handle	Description
Device	CUdevice	CUDA-enabled device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA array	CUarray	Opaque container for one-dimensional or two-dimensional data on the device, readable via texture references
Texture reference	CUtexref	Object that describes how to interpret texture memory data

The CUDA code

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C){
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main(){
...
// Kernel invocation
VecAdd<<<1, N>>>(A, B, C);
}
```

Using driver API

```
#define ALIGN_UP(offset, alignment) \
(offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)

int main(){
    // Initialize
    if (cuInit(0) != CUDA_SUCCESS) exit (0);
    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }
    // Get handle for device 0
    CUdevice cuDevice = 0;
    cuDeviceGet(&cuDevice, 0);
    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);
```

```
// Create module from binary file
CUmodule cuModule;
cuModuleLoad(&cuModule, “VecAdd.ptx”);

// Get function handle from module
CUfunction vecAdd;
cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

// Calculate parameter size position
int offset = 0;
void* ptr = (void*)(size_t)A;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);

ptr = (void*)(size_t)B;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
```

```
ptr = (void*)(size_t)C;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
cuParamSetSize(vecAdd, offset);

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
(N + threadsPerBlock - 1) / threadsPerBlock;
cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
cuLaunchGrid(vecAdd, blocksPerGrid, 1);
```

CUDA context and module

- A CUDA context is analogous to a CPU process.
  - Each context has distinct 32-bit address space.
  - Use **CUdeviceptr** to access memory locations.

- Modules are dynamically loadable packages of device code and data, akin to DLLs.
  - Maintain names for symbols of functions, global variables, and texture references

cuParam* functions

- The cuParamSetv specifies the parameters for the next time cuLaunchGrid() or cuLaunch()
- The second argument specifies the offset of the parameter in the parameter stack.
  - Must match the alignment requirement for the parameter type in device code.

Kernel execution

- cuFuncSetBlockShape() sets the number of threads per block for a given function, and how their threadIDs are assigned.
- cuFuncSetSharedSize() sets the size of shared memory for the function.
- cuLaunchGrid() invokes the kernel function on a grid_width x grid_height grid of blocks.

HOMEWORK