# MI3.22
# Advanced Programming for HPC
# Master ICT, USTH, $2^{nd}$ year

Aveneau Lilian

lilian.aveneau@univ-poitiers.fr
XLIM/SIC/IG, CNRS, Computer Science Department
University of Poitiers

Year 2019/2020

# Lecture 4 – CUDA: Atomics & Stream

- Atomic Operations
- Locking Memory Pages
- CUDA Streams
- Host Memory Access since GPU

# Overview

- Atomic Operations
- Locking Memory Pages
- CUDA Streams
- Host Memory Access since GPU

# Compute Capability

Previous SON's examples use only basic CUDA capacities

## Notion of compute capacity

- Growing versions: 1.0, 1.1, 1.2, 1.3, 2.0, ... 7.5 (Turing)
- "Russian dolls" model: version 2.0 includes the previous, and so on ...
- Basically it defines the instruction set, types, behavior ...
- List on *CUDA Zone*
  `http://www.nvidia.fr/object/cuda_home_new_en.html`

| | | | | |
|---|---|---|---|---|
| | GeForce 470 GTX | 2.0 | Quadro M2020 | 5.2 |
| Examples: | GTX Titan Black | 3.5 | GeForce GTX 1080 | 6.1 |
| | Tesla K80 | 3. 7 | GeForce 2080 | 7.5 |

## Which capacities for Atomic Operation

- SM $\geq 1.1 \Rightarrow$ Atomic Operations on global memory
- SM $\geq 1.2 \Rightarrow$ Atomic Operations on shared memory

Compilation: option `-arch sm_11` or `-arch sm_12` or above

# Atomic Operations in a nutshell

## Parallelism induces new needs

Example with following instruction: `x++`

- Load x - Add 1 - Write x

  $\Longrightarrow$ Operation *read-modify-write*
- Race condition using many PEs: execution order problem
- Example with only 2 PEs: $\mathcal{C}_6^3 = \binom{6}{3} = 20$ possibilities!
  - R1 - A1 - W1 - R2 - A2 - W2 : ok
  - R1 - R2 - A1 - W1 - A2 - W2 : error
  - Only two good orders: 90% lead to a bad behavior!

## Solutions

- Mutex, Semaphore, ... not CUDA, or using `atomicCas()`
- Synchronization (barrier, ...): too slow, not always possible (only inside a same block)
- Atomic Operations, the best way with CUDA

# Example: histogram computation

## Generic tool, with multiple uses

- Image analysis, compression, computer vision, AI learning, audio codecs ...
- Example with "PROGRAMMING WITH CUDA C" :

| | $A$ | $C$ | $D$ | $G$ | $H$ | $I$ | $M$ | $N$ | $O$ | $P$ | $R$ | $T$ | $U$ | $W$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |

Example of histogram computation, on a random byte array

```
const int SIZE=100*(1<<20); // 100 Mega
int main () {
  // Allocate and random initialization
  unsigned char *buffer =
                big_random_block( SIZE );
  // Histogram computation
  unsigned int histo[256];
  memset ( histo, 0, sizeof(histo) );
  for (int i=0; i<SIZE; i++)
    ++ histo[ buffer[i] ];
```

```
  // Naive computation check
  long histoCount = 0;
  for ( int i=0; i<256; i++ )
    histoCount += histo[ i ];
  // Computation ok -> histoCount == SIZE ...
  std::cout<< "Histogram sum: "
           << histoCount << " == " << SIZE
           << " ?" << std::endl;
  return 0;
}
```

On my computer, computation time is 55 ms (without initialization)

# Histogram on GPU

It is a *read-modify-write* operation! So atomic add:

```
__global__ void computeHisto ( const unsigned char*const buffer,
                               unsigned int*const histo, const long size ) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  while ( i< size ) {   atomicAdd( &histo[buffer[i]], 1 );   i += stride;  }
}

int main () { // initializations, GPU allocations ...
  unsigned char *buffer = big_random_block( SIZE );
  unsigned char*dev_buffer;
  HANDLE_ERROR( cudaMalloc( (void**)&dev_buffer, SIZE ));
  HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE,  cudaMemcpyHostToDevice )); // upload
  unsigned int*dev_histo;
  HANDLE_ERROR( cudaMalloc( (void**)&dev_histo, 256*sizeof( int ) ));
  HANDLE_ERROR( cudaMemset( dev_histo, 0, 256*sizeof( int) )); // Yes, it exists!
  cudaDeviceProp props; // calcul heuristique du nombre de blocs
  HANDLE_ERROR( cudaGetDeviceProperties( &props, 0 ));
  const int blocs = props.multiProcessorCount;
  computeHisto<<<blocs*2,256>>> ( dev_buffer, dev_histo, SIZE );
  // download the histogram, cleanup GPU resources
  unsigned int histo[ 256 ];
  HANDLE_ERROR( cudaMemcpy( histo, dev_histo, 256*sizeof(int), cudaMemcpyDeviceToHost ));
  HANDLE_ERROR( cudaFree( dev_buffer ));  HANDLE_ERROR( cudaFree( dev_histo ));
  long histoCount = 0; // (too) Naive check the result
  for (int i=0; i<256; i++) histoCount += histo[ i ];
  if ( histoCount != SIZE ) { std::cerr << "Erreur de calcul de l'histogramme !!\n"; }
  return 0;
}
```

Computation time: 67 ms on Quadro M2200 (Maxwell device)

# We can do better!

Solution to calculate faster: add more atomic instructions!

```
__global__ void computeHisto ( const unsigned char*const buffer,
                               unsigned int*const histo,
                               const long size )
{
  // shared histogram inside each block
  __shared__ unsigned int temp[ 256 ];
  // initialization, and so barrier
  temp[ threadIdx.x ] = 0;
  __syncthreads();
  // initialization and "classical" computation
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  const int stride = blockDim.x * gridDim.x;
    while ( i< size ) {
    atomicAdd( &temp[buffer[i]], 1 ); // Warning: SM >= 12 !!!
    i += stride;
  }
  // Waiting all block threads before to write the result in GLOBAL memory
  __syncthreads();
  atomicAdd( &histo[threadIdx.x], temp[ threadIdx.x ] ); // Now SM >= 11
}
```

Computation time: 5 ms on Quadro M2200

- More difference on old Fermi devices
- Kepler/Maxwell improves the atomic operations

# Locks (or mutex)

No specific instruction, but can be done (as *spinlock* on CPU):

- Use an integer stored on GPU's DRAM
- `AtomicCAS()`: Compare And Store

```
#ifndef __LOCK_H__
#define __LOCK_H__
#include "common.h"

class Lock {
  int *mutex; // plays the ``mutex'' role, accessible by all threads
 public:
  Lock( void ) { // only for the host (GPU receives a lock by copy)
    HANDLE_ERROR( cudaMalloc( (void**)&mutex, sizeof(int) ) ); // allocated on GPU
    HANDLE_ERROR( cudaMemset( mutex, 0, sizeof(int) ) );
  }

  ~Lock( void ) {
    cudaFree( mutex );
  }

  __device__ void lock( void ) {               // How works atomicCAS( ptr, old, new ):
    while( atomicCAS( mutex, 0, 1 ) != 0 ); // IF *ptr == old THEN *ptr=new; returns old;
  }                                            // ELSE returns *ptr;

  __device__ void unlock( void ) {
    atomicExch( mutex, 0 );                    // Atomically set *mutex=0
  }
};
#endif
```

# Example with dot product

```
__global__ void dot( Lock lock, float *a, float *b, float *c ) { // receive lock + c
  extern __shared__ float cache[]; // allocated at kernel call (triple bracket)
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int cacheIndex = threadIdx.x;
  float temp = 0.f;
  while (tid < N) {   temp += a[tid] * b[tid];   tid += blockDim.x * gridDim.x;   }
  cache[cacheIndex] = temp;
  __syncthreads();          // threads synchronization into this block
  int i = blockDim.x>>1;   // to reduce, blocDim.x must be a power of 2
  while (i != 0) {
    if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i >>= 1;
  }
  if ( cacheIndex == 0 ) {
    lock.lock();     // take the spinlock (waiting loop)
    *c += cache[0]; // -> we are in an exclusive area, so we can modify c!
    lock.unlock();  // unlock the spinlock
  }
}
```

Call:

```
// allocate c: only one float!
HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float), cudaMemcpyHostToDevice ));
Lock lock; // The constructor is called on CPU
dot <<<blocksPerGrid,threadsPerBlock, sizeof(float)*threadsPerBlock>>>
     ( lock, dev_a, dev_b, dev_c );
// No more sum on CPU! We can use directly "c" ...
```

While it is not really faster (26,8 ms for 256 Mega), it is now fully on GPU ...

# Overview

- Atomic Operations
- Locking Memory Pages
- CUDA Streams
- Host Memory Access since GPU

# cudaHostAlloc()

Allocating into the heap: `malloc()` or `cudaHostAlloc()`?

## Using `malloc()`

- Classical mechanism: allocates pages on main memory
- Memory is uploaded/downloaded on disk (swap mechanism)
- Leads to slow-down when page fault occurs

## Using `cudaHostAlloc()`

- Allocation of pin-pages, so with constant address!
- So, can be accessed directly to/from the GPU using DMA (Direct Memory Access)
- No more slow-down (no page fault), but: system saturation risk
- Such an allocation provides "pinned memory"

In practice memory transfer always USES pinned memory: we can use it explicitly to avoid CPU copy to pinned memory ...

# Benchmark (1/3)

```
#include <iostream>
#include "common.h"
const unsigned NbLoops = 100u;
float cuda_malloc_test( const long size, const bool up )
{
  ChronoGPU chr;

  char *a = new char[size]; // or using cudaHostAlloc()
  char *dev_a;
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a, size ) );

  chr.start();
  for (int i=0; i<NbLoops; i++) {
    if (up)
      HANDLE_ERROR( cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice ) );
    else
      HANDLE_ERROR( cudaMemcpy( a, dev_a, size, cudaMemcpyDeviceToHost ) );
  }
  chr.stop();
  float elapsedTime = chr.elapsedTime();

  delete a;
  HANDLE_ERROR( cudaFree( dev_a ) );

  return elapsedTime;
}
```

# Benchmark (2/3)

```
float cuda_host_alloc_test( const long size, const bool up )
{
  ChronoGPU chr;

  char *a, *dev_a;
  // Allocation in ''pinned'' mode
  HANDLE_ERROR( cudaHostAlloc( (void**)&a, size, cudaHostAllocDefault ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a, size ) );

  chr.start();
  for (int i=0; i<NbLoops; i++) {
    if (up)
      HANDLE_ERROR( cudaMemcpy( dev_a, a, size,cudaMemcpyHostToDevice ) );
    else
      HANDLE_ERROR( cudaMemcpy( a, dev_a, size, cudaMemcpyDeviceToHost ) );
  }
  chr.stop();
  float elapsedTime = chr.elapsedTime();

  // Take a look at the cleanup using cudaFreeHost() ...
  HANDLE_ERROR( cudaFreeHost( a ) );
  HANDLE_ERROR( cudaFree( dev_a ) );

  return elapsedTime;
}
```

# Benchmark (3/3)

```
int main( void ) {
  const long SIZE = 1<<28; // so 256 Mb
  const float  MB = static_cast<float>( NbLoops * (SIZE>>20) );

  // Bench using classical 'cudaMalloc', uploading (host to GPU)
  float elapsedTime = cuda_malloc_test( SIZE, true );
  cout<< "Time using cudaMalloc: "<<elapsedTime<<" ms\n";
  cout<< "\tMB/s during copy up: "<<(MB/(elapsedTime/1000.f))<<endl;

  // Bench using classical 'cudaMalloc', downloading (GPU to host)
  elapsedTime = cuda_malloc_test( SIZE, false );
  cout<< "Time using cudaMalloc: "<<elapsedTime<<" ms\n";
  cout<< "\tMB/s during copy down: "<<(MB/(elapsedTime/1000.f))<<endl;

  // Bench using 'cudaHostAlloc', uploading (host to GPU)
  elapsedTime = cuda_host_alloc_test( SIZE, true );
  cout<< "Time using cudaHostAlloc:  "<<elapsedTime<<" ms\n";
  cout<< "\tMB/s during copy up:  "<<(MB/(elapsedTime/1000.f))<<endl;

  // Bench using 'cudaHostAlloc', downloading (GPU to host)
  elapsedTime = cuda_host_alloc_test( SIZE, false );
  cout<< "Time using cudaHostAlloc:  "<<elapsedTime<<" ms\n";
  cout<< "\tMB/s during copy down:  "<<(MB/(elapsedTime/1000.f))<<endl;

  return 0;          //Time using cudaMalloc: 4908.47 ms
}                    //        MB/s during copy up: 5215.47
                     // Time using cudaMalloc: 5106.2 ms
                     //        MB/s during copy down: 5013.51
                     // Time using cudaHostAlloc:  2460.02 ms
                     //        MB/s during copy up:  10406.4
                     // Time using cudaHostAlloc:  2596.96 ms
                     //        MB/s during copy down:  9857.67
```

# Overview

- Atomic Operations
- Locking Memory Pages
- CUDA Streams
- Host Memory Access since GPU

# Introduction

Pinned memory useful, but it is mainly a necessity for Streams ...

## Streams

- Queue of operations on GPU, which are executed in the order ...
- Contains: kernel, memory transfers, event recording
- Utility: mainly to execute different tasks in parallel
  - Needs `cudaDeviceProperties.deviceOverlap == true`

## How to use them

1. Dedicated structure `cudaStream_t`
2. Asynchronous memory copies `cudaMemcpyAsync()`
3. Launch kernel using `kernel<<<bl,th,0,stream>>>()`
4. Synchronization using `cudaSynchronize(stream)`

## Warning

Asynchronous memory: needs pinned memory

# Example mono-stream (1/3)

```
#include <iostream>
#include "common.h"
#include <random> // Random number generator, with periodicity 2^19937-1
using namespace std;

const int N = 1024*1024 ; // 1 Mo, the number of threads
const int FULL_DATA_SIZE = N * 20; // 20 data per thread

__global__ void kernel( int *a, int *b, int *c ) {
  const int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < N) { // In fact, the following calculation does not matter ;-)
    const int idx1 = (idx + 1) & 0xFF;
    const int idx2 = (idx + 2) & 0xFF;
    const float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
    const float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
    c[ idx ] = (as + bs) * .5f;
  }
}

int main( void ) {
  cudaDeviceProp  prop; // We verify that GPU allows copy + kernel in ||
  int whichDevice;
  HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
  HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
  if (!prop.deviceOverlap) {
    cout<< "Device will not handle overlaps, so no speed up from streams\n";
  }

  ChronoGPU     chr;
  float         elapsedTime;

  cudaStream_t stream; // Stream identifier
```

# Example mono-stream (2/3)

```
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;

// Create/initialize the stream
HANDLE_ERROR( cudaStreamCreate( &stream ) );

// Allocation on GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

// Pinned memory allocation, needed by streams
const unsigned int TRUE_SIZE = FULL_DATA_SIZE * sizeof( int );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, TRUE_SIZE, cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, TRUE_SIZE, cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, TRUE_SIZE, cudaHostAllocDefault ) );

// Initialization of some data
auto mt_rand = std::bind(std::uniform_int_distribution<int>(0,FULL_DATA_SIZE), std::mt1993
     for (int i=0; i<FULL_DATA_SIZE; i++) {
  host_a[i] = mt_rand();
  host_b[i] = mt_rand();
}

// Starts the chronometer
chr.start();
```

# Example mono-stream (3/3)

```
  // Loop on the data, using batch of size N (lot?)
  const int N4 = sizeof( int ) * N;
  for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // Asynchronous copies of data from host's RAM to GPU
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i, N4, cudaMemcpyHostToDevice, stream ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i, N4, cudaMemcpyHostToDevice, stream ) );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c ); // 0: no shared meory

    // Data copy from GPU to host
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c, N4, cudaMemcpyDeviceToHost, stream ) );

  }
  // Waits the last copy to host's RAM
  HANDLE_ERROR( cudaStreamSynchronize( stream ) );

  chr.stop();
  elapsedTime = chr.elapsedTime();
  cout<< "Time taken:  "<<elapsedTime<<" ms\n";

  // Spring cleanup
  HANDLE_ERROR( cudaFreeHost( host_a ) );
  HANDLE_ERROR( cudaFreeHost( host_b ) );
  HANDLE_ERROR( cudaFreeHost( host_c ) );
  HANDLE_ERROR( cudaFree( dev_a ) );
  HANDLE_ERROR( cudaFree( dev_b ) );
  HANDLE_ERROR( cudaFree( dev_c ) );
  HANDLE_ERROR( cudaStreamDestroy( stream ) ); // Note the release of the stream

  return 0;
}
```

# Version with two streams

Main loop: launch the two streams ...

```
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) { // half iteration!
  // Works with the first stream
  HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N4, cudaMemcpyHostToDevice, stream0 ));
  HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N4, cudaMemcpyHostToDevice, stream0 ));
  kernel<<< N/256, 256, 0, stream0 >>>( dev_a0, dev_b0, dev_c0 );
  HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0, N4, cudaMemcpyDeviceToHost, stream0 ));

  // Works with the second stream
  HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N4, cudaMemcpyHostToDevice, stream1));
  HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N4, cudaMemcpyHostToDevice, stream1));
  kernel<<< N/256, 256, 0, stream1 >>>( dev_a1, dev_b1, dev_c1 );
  HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N4, cudaMemcpyDeviceToHost, stream1));
}
```

## Result

- Same timings!
- In fact, we need to understand the GPU:
  - 2 virtual processors: one for copies, the other for kernel
  - Each one has it own order queue
  - Waiting to respect the stream order: induced constraint, we need to take it into account in our stream usage!
- Solution: launch 2nd kernel BEFORE downloading the 1st

# Overview

- Atomic Operations
- Locking Memory Pages
- CUDA Streams
- Host Memory Access since GPU

# Zero-Copy Host Memory

Principle: pinned memory can be directly accessed from the GPU

## Example: dot product

- We reuse the previous code, without allocating memory on GPU
- Instead, we "map" host memory to GPU
  - Function `cudaHostGetDevicePointer( void**, void*, 0)`
- It needs unmovable memory (so, pinned memory)

## Replace the classical host allocation

- `cudaHostAllocPortable`: accessible from each threads on CPU
- `cudaHostAllocWriteCombined` :
  - Do not use L1 and L2 cache: slower when reading,
  - But faster when writing – no more page fault handling
- `cudaHostAllocMapped`: Grants access since GPU

## Results

On my laptop, 2 to 4 times faster!