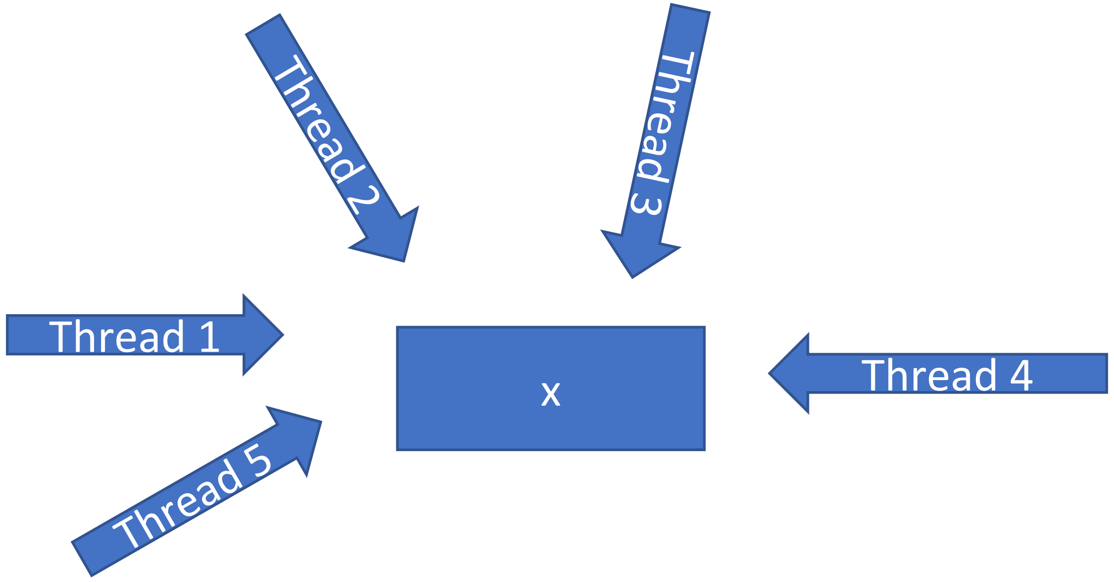
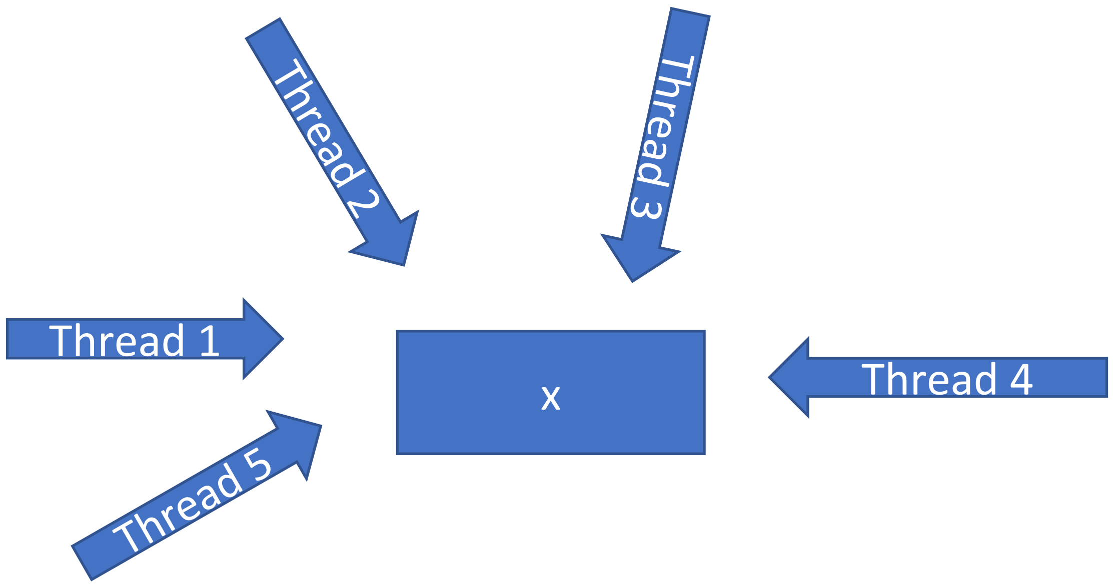
Atomic functions

Issue



- Suppose that multiple threads are trying to use the value of x for some computation and want to update x after the computation
- Each of the computed result is necessary for correct computation
  - For example,
    - Thread 1 uses original value of x to compute some results and save it to x
    - Thread 2 uses the new value of x to computer some results and save it to x
    - And so on….

Issue



- What is the issue?
  - Suppose Thread 3 updates the value of x, then other threads need to use the new value
  - But when Thread 4 access the value of x, it might not have been updated
    - So it will use old value from x, compute and update
    - The updated value is wrong since it used the wrong value of x
  - Other threads might also have this issue

Issue

- With multiple threads, there is an issue with read-modify-write operations on shared data
- A thread may read wrong value, modify the new value by computation using old value or write an older update and replace the new value
- There is a problem of coordination and overwriting among threads
- There is a need for uninterrupted read-modify-write operations
- Atomic functions (atomic memory options) is one of the method to solve it

Atomic functions

- A mechanism to ensure that all the atomic operations are performed correctly among threads
- Guarantees the atomic operations by allowing a thread to read-modify-write a shared memory location with respect to other threads
  - No other threads can operated on the same memory location until the current operation is complete
- Ensures that the atomic operations are performed concurrently and observed by all other threads

Example of a program where atomic operations are not synchronized

- We consider a very simple example to understand the issue
  - Assume that there is a vector with all elements 1
  - We want to add the elements of the vector
  - If the vector has size of 1000, then the result should be 1000 as all the elements contains 1
  - Consider the following kernel for this operation

```
__global__ void simple_count(int *a, int *sum, int n) {
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id < n)
        *sum =*sum+a[thread_id];
}
```

Example of a program where atomic operations are not synchronized

- If you run the program and print the value of sum, you will see that the result is not 1000, but other random number
- The reason is that each of the thread, while accessing sum, do not get the latest value of sum due to lack of synchronization
- So the final value of sum is not correct

```
__global__ void simple_count(int *a, int *sum, int n) {
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id < n)
        *sum =*sum+a[thread_id];
}
```

Example of a program with atomic functions

- We can use atomic function atomicAdd to synchronize the threads
  - We replace *sum =*sum+1 by atomicAdd(sum,1)
- atomicAdd guarantees that only one thread can access memory location for sum at a time
- It ensures that all the atomic operations are executed automatically without conflict from other threads

```
__global__ void atomic_count(int *a, int *sum, int n) {
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id < n)
        atomicAdd(sum,a[thread_id]);
}
```

Here is the main function

```
int main(void) {
//host variables
int *h_var, *h_sum ;
int sum=0;

//device variables
int* d_var, *d_sum;

size_t size_vect = SIZE*sizeof(int); /* size of the total vectors necessary
to allocate memory */

//allocate memory for the variables on host (cpu)
h_var = (int*)malloc(size_vect);
h_sum = (int*)malloc(sizeof(int));
h_sum=&sum;/* h_sum is to store the sum on the host device */

//allocate memory for the variables on device (gpu)
cudaMalloc((void **)&d_var, size_vect);
cudaMalloc((void **)&d_sum, size_vect);
cudaMemset ((void **)d_sum,0, sizeof(int));

//initialize the vectors each with value 1
for (int i = 0; i < SIZE; i++) {
h_var[i] = 1;
}

//Start CUDA processing
// Copy host values to device
cudaMemcpy(d_var, h_var, size_vect, cudaMemcpyHostToDevice);
```

```
//define number of threads
int threads = 1024;
//define block size in integer
int block_size = (int)ceil((float)SIZE / threads);

//execute the kernel with block size and number of threads
simple_count << <block_size, threads >>> (d_var, d_sum, SIZE);

// Copy result back to host
cudaMemcpy(h_sum, d_sum, sizeof(int), cudaMemcpyDeviceToHost);

//Verify the result, should be equal to SIZE
printf("Result without atomic add : %d\n",sum);

//reset d_sum to 0
cudaMemset ((void **)d_sum,0, sizeof(int));

//execute the kernel with block size and number of threads
atomic_count << <block_size, threads >>> (d_var, d_sum, SIZE);

// Copy result back to host
cudaMemcpy(h_sum, d_sum, sizeof(int), cudaMemcpyDeviceToHost);

//Verify the result, should be equal to SIZE
printf("Result using atomic add : %d\n",sum);

// Release all device memory
cudaFree(d_var);

// Release all host memory
free(h_var);
}
```

Atomic functions

- An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.
- It does not act as memory fences and does not imply synchronization or ordering constraints for memory operations .
  - The order in which concurrent atomic updates are performed is arbitrary.
- Atomic functions can only be used in device functions.

Atomic add functions from CUDA Toolkit documentation

- atomicAdd()
  - reads the 16-bit, 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.
- int atomicAdd(int* address, int val);
- unsigned int atomicAdd(unsigned int* address, unsigned int val);
- unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);
- float atomicAdd(float* address, float val);
- double atomicAdd(double* address, double val);
- __half2 atomicAdd(__half2 *address, __half2 val);
- __half atomicAdd(__half *address, __half val);
- __nv_bfloat162 atomicAdd(__nv_bfloat162 *address, __nv_bfloat162 val);
- __nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);

Other atomic functions

- Arithmetic Functions
  - atomicAdd()
  - atomicSub()
  - atomicExch()
  - atomicMin()
  - atomicMax()
  - atomicInc()
  - atomicDec()
  - atomicCAS()
- Bitwise Functions
  - atomicAnd()
  - atomicOr()
  - atomicXor()
- For details: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Atomic functions limitations

- Atomic functions are slower compared to normal accesses
  - Threads might need to wait idly to get access to a memory location for their atomic operations
  - Performance can thus degrade when many threads need to perform atomic operations on a smaller location
- Atomic functions perform a limited set of operations with certain data types and might not be useful for complex operations
- Since there is no ordering constraints on thread, it does not provide synchronization or barrier

Exercise

- Exercise 1: Run the given program for atomic addition and analyze the output of with and without atomic function
  - While using `simple_count` what output do you observe? Run it multiple times and check. Is the output consistent?
- Exercise 2:
  - Using atomicMax to compute max value in the list
  - Create a list of size 900,000 and set the values from 1 to 900,000
  - Write a CUDA kernel to find max from the list using simple comparison
    - Eg. Let max = 0, if val> max, max = val
  - Write a CUDA kernel to find max from the list using atomicMax
    - Eg. Let max = 0, atomicMax(max,val)
  - Check your output to ensure that atomicMax finds the max, that is 900,000

References

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html