High Energy Physics

Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

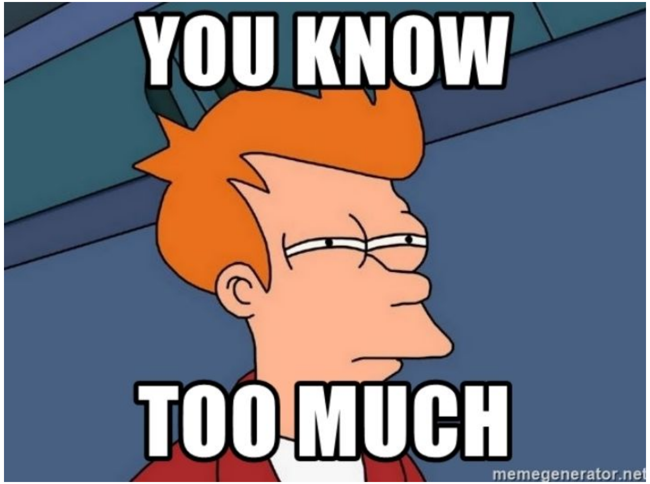
GPU & FPGA module training

Week 4 : Introduction to CUDA

Lecture 8 - February 15th 2023

What we learnt in the previous lecture

- We reminded ourselves of the GPUs memory layout
- We discussed about data locality and the importance of caching
- We understood the coalesced memory data access pattern



Today

Today we will learn about :

- Shared memory
- Atomic operations
- The default CUDA stream

Shared memory

Shared memory

Shared memory works differently from DRAM :

- From the hardware perspective :
    - Resource per SM
- From the software software perspective:
    - Resource per block of threads

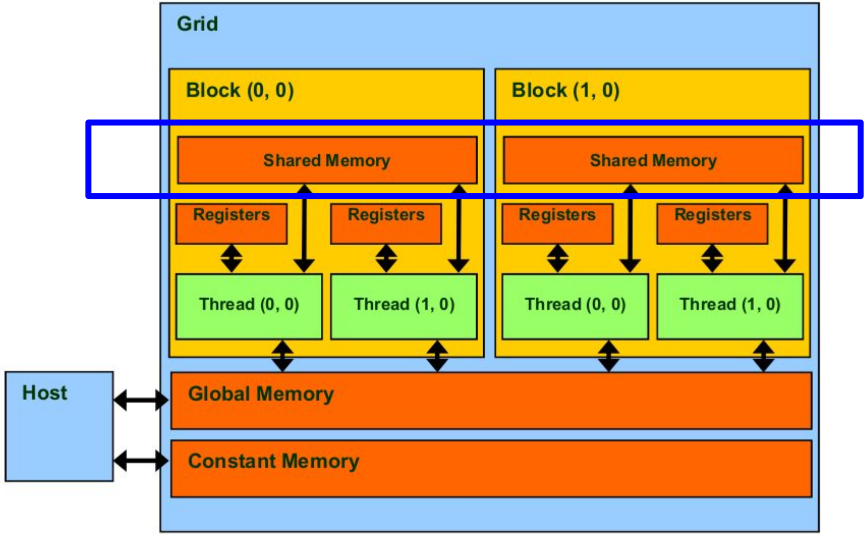
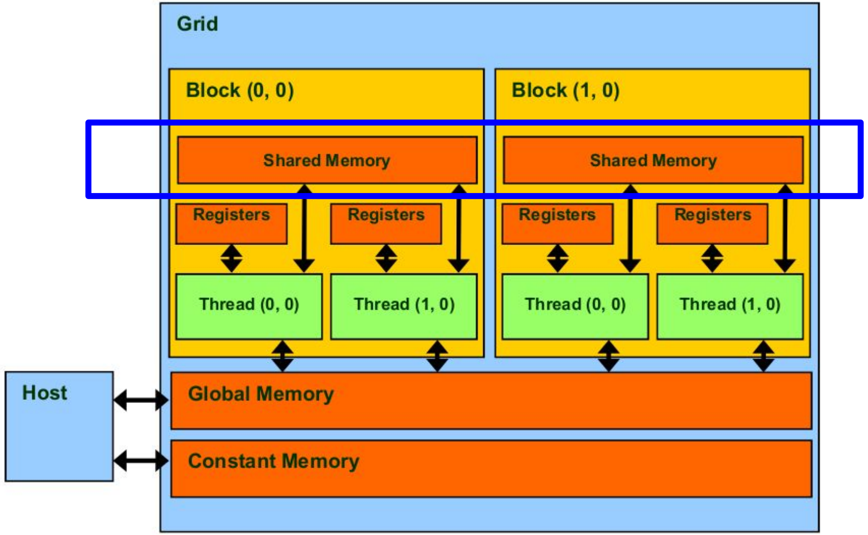


Image source [1]

Shared memory



Shared memory works differently from DRAM :

- From the hardware perspective :
  - Resource per SM
- From the software software perspective:
  - Resource per block of threads

Shared memory is useful :

- Allows inter-thread communication within a thread block
- Allows caching of data to reduce redundant global memory accesses
- Can help improve global memory access patterns

Image source [1]

Shared memory

- Shared memory can be defined by using the **__shared__** qualifier
  - e.g. __shared__ int var;

- Declared in CUDA kernel :
  - Can be **static** or **dynamic**

- Allocated on a per thread block basis :
  - Any variable declared as __shared__ will be accessible by all threads in a block
  - Variable i not visible by threads in other blocks

- It is limited in size:
  - The maximum varies depending on the device architecture.

Static shared memory

- Shared memory is allocated within the kernel

```
__global__ void my_kernel(float *result) {
    // The size of the shared variable is known at
compile time :
    __shared__ float shared_var[N];
    for (int i = threadIdx.x; I < N; i++) {
        shared_var[i] = ...;
    }
    __syncthreads();
    for (int i = threadIdx.x; I < N; i++) {
            result = Do something with shared_var[i]
        }
    }
}
int main(void) {
    ...
    // The kernel launch is as usual
    my_kernel<<gridDim,blockDim>>(result);
    ...
    return 0;
}
```

Static shared memory

- Shared memory is allocated within the kernel
- If the size is known at compile time, it is declared with that size directly in the kernel

```
__global__ void my_kernel(float *result) {
   // The size of the shared variable is known at
compile time :
   __shared__ float shared_var[N];
   for (int i = threadIdx.x; I < N; i++) {
       shared_var[i] = ...;
   }
   __syncthreads();
   for (int i = threadIdx.x; I < N; i++) {
          result = Do something with shared_var[i]
       }
   }
}
int main(void) {
   ...
   // The kernel launch is as usual
   my_kernel<<gridDim,blockDim>>(result);
   ...
   return 0;
}
```

Static shared memory

- Shared memory is allocated within the kernel
- If the size is known at compile time, it is declared with that size directly in the kernel
- Call to __syncthreads() is usually needed if results computed with other threads are needed

```
__global__ void my_kernel(float *result) {
   // The size of the shared variable is known at compile time :
   __shared__ float shared_var[N];
   for (int i = threadIdx.x; I < N; i++) {
       shared_var[i] = ...;
   }
   __syncthreads();
   for (int i = threadIdx.x; I < N; i++) {
          result = Do something with shared_var[i]
       }
   }
}
int main(void) {
   ...
   // The kernel launch is as usual
   my_kernel<<gridDim,blockDim>>(result);
   ...
   return 0;
}
```

Dynamic shared memory

- If the size is only known at run time shared memory can be allocated dynamically :
  - Declared within the kernel
  - Declaration requires the keyword **extern**

```
__global__ void my_kernel(float *result) {
   // The size of the shared variable is not known at compile time :
   extern __shared__ float var_sh[];
   for (int i = threadIdx.x; I < N; i++) {
       shared_var[i] = ...;
   }
   __syncthreads();
   for (int i = threadIdx.x; I < N; i++) {
       result = Do something with shared_var[i]
   }
}

int main(void) {
   ...
   // The kernel launch has an additional parameter
   my_kernel<<gridDim,blockDim,N*sizeof(float)>>(result);
   ...
   return 0;
}
```

Dynamic shared memory

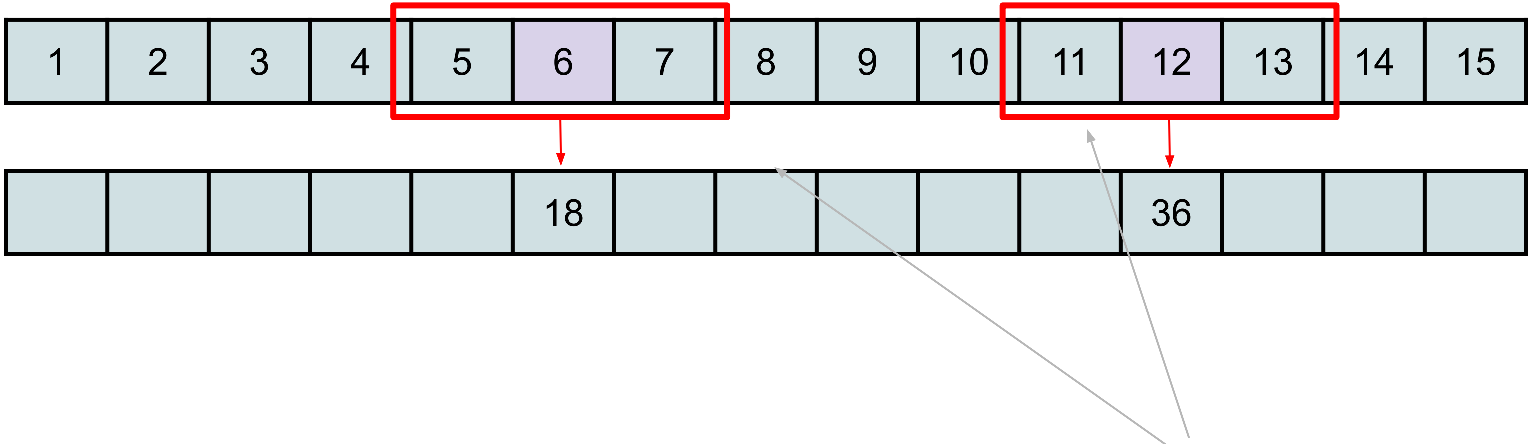
- If the size is only known at run time shared memory can be allocated dynamically :
  - Declared within the kernel
  - Declaration requires the keyword **extern**
- Size must be known on the host and should be passed as an additional kernel call argument

```
__global__ void my_kernel(float *result) {
   // The size of the shared variable is not known at compile time :
   extern __shared__ float var_sh[];
   for (int i = threadIdx.x; I < N; i++) {
       shared_var[i] = ...;
   }
   __syncthreads();
   for (int i = threadIdx.x; I < N; i++) {
       result = Do something with shared_var[i]
   }
}

int main(void) {
   ...
   // The kernel launch has an additional parameter
   my_kernel<<gridDim,blockDim,N*sizeof(float)>>(result);
   ...
   return 0;
}
```

Shared memory : A practical example

Lets understand how shared memory works by trying to apply a **1-D stencil** to a 1-D array!

- Each output element will be the sum of input elements within a predefined radius :

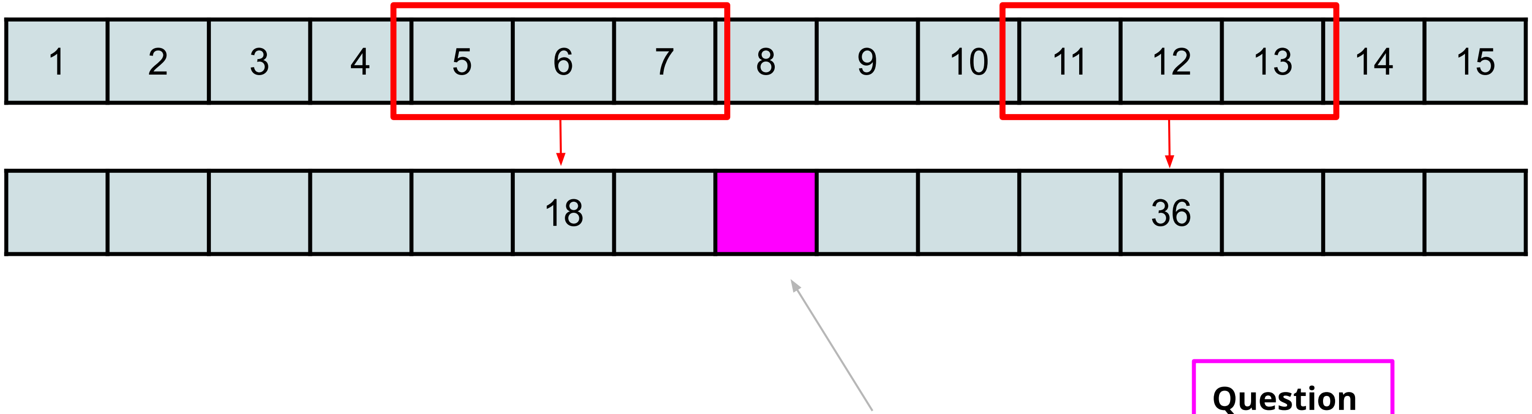


We assume that **radius = 1** for this example

Shared memory : A practical example

Lets understand how shared memory works by trying to apply a **1-D stencil** to a 1-D array!

- Each output element will be the sum of input elements within a predefined radius :



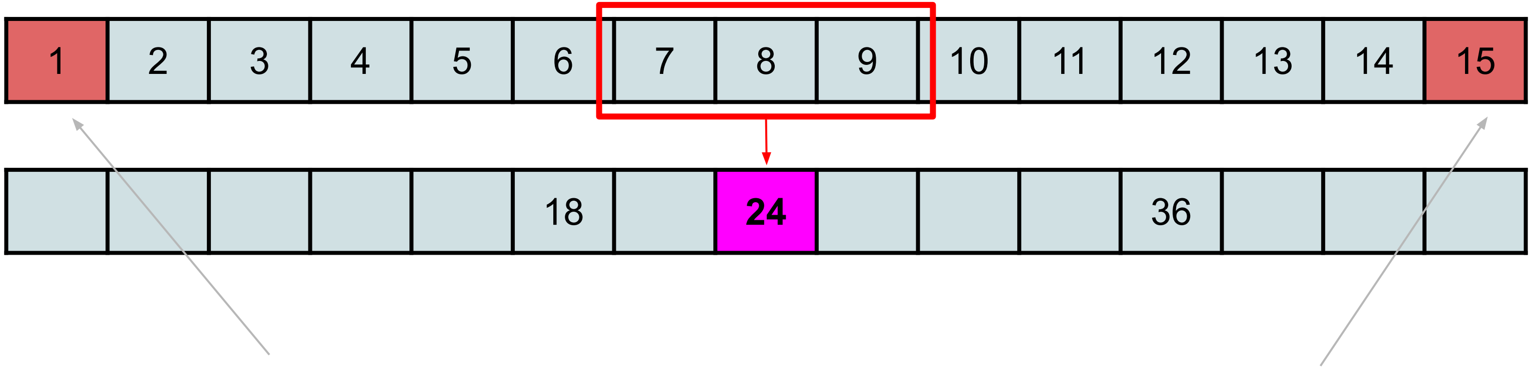
Question

What will be the value of this element if we apply the stencil?

Shared memory : A practical example

Lets understand how shared memory works by trying to apply a **1-D stencil** to a 1-D array!

- Each output element will be the sum of input elements within a predefined radius :

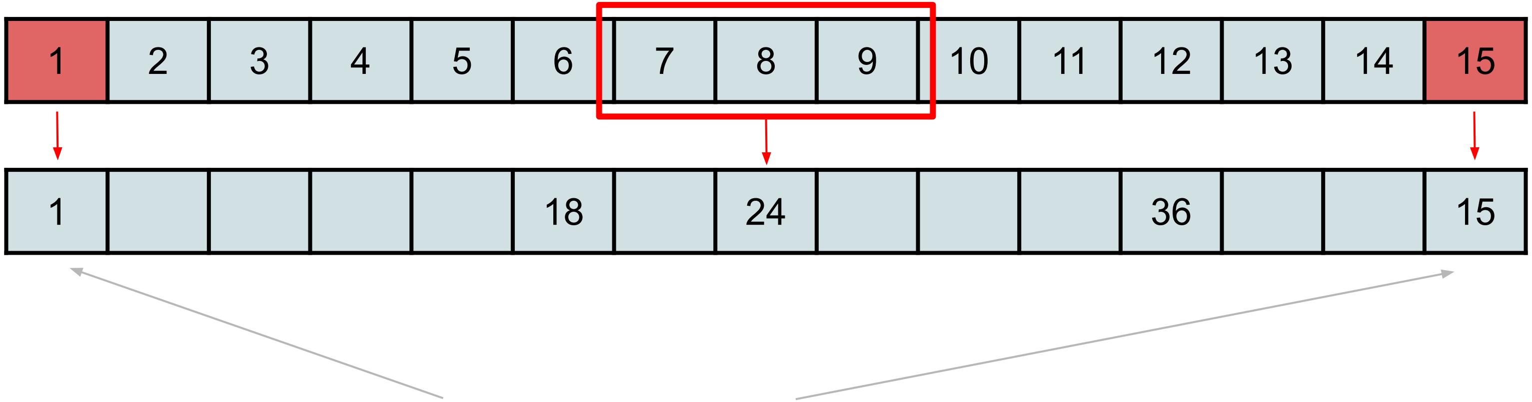


- What happens to these "boundary" elements? (these are equal to the stencil radius)

Shared memory : A practical example

Lets understand how shared memory works by trying to apply a **1-D stencil** to a 1-D array!

- Each output element will be the sum of input elements within a predefined radius :

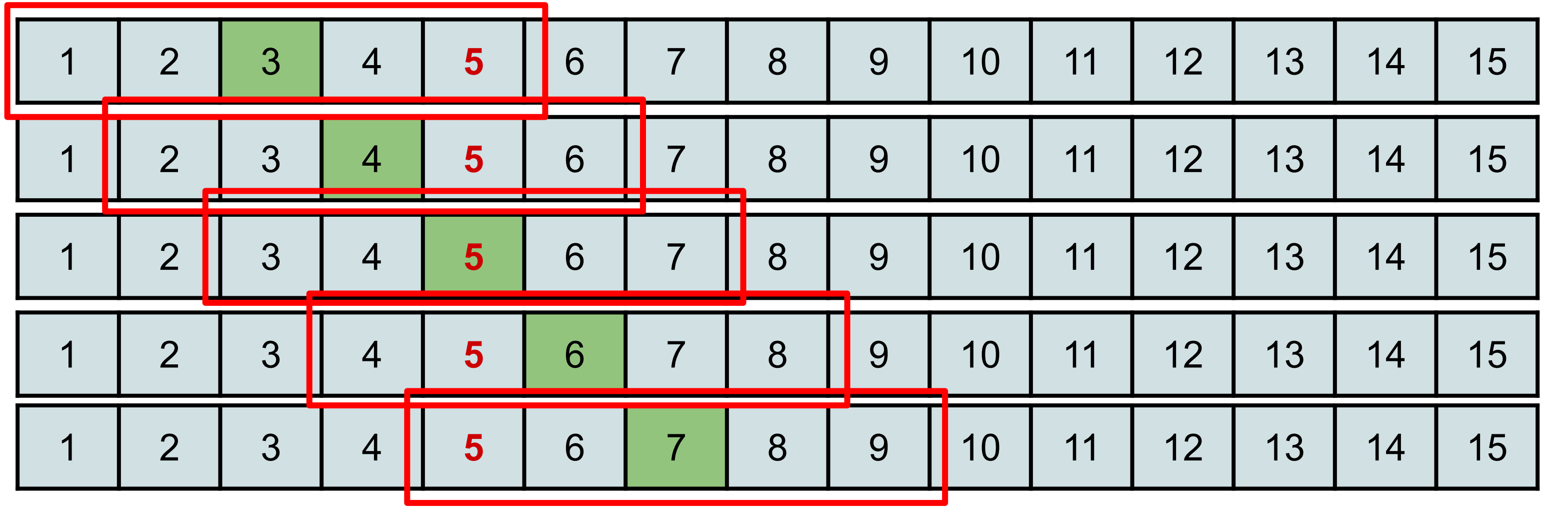


- They do not change when applying the stencil

Shared memory : A practical example

Why is this a problem that benefits from using shared memory?

- **Input elements are read several times!!**
  - This depends on the radius size
  - e.g for radius = 2, element 5 is read 5 times



1-D stencil kernel

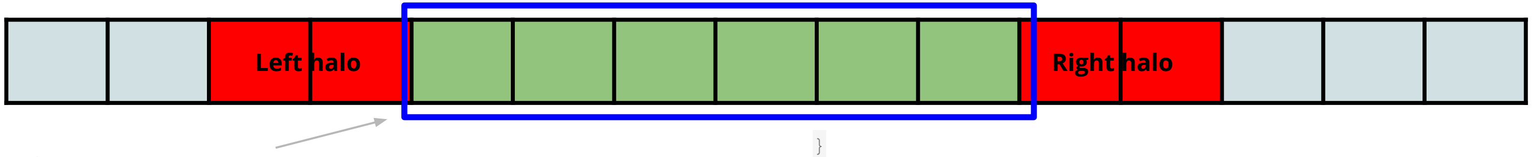
- Shared memory is allocated per-block :
  - Threads in the same block can access the shared variable
  - Threads from different blocks cannot

```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;

}
```

1-D stencil kernel

- In order to properly apply the stencil to the boundary elements we have to have access to some additional edge elements :
  - These are equal to the radius of the stencil

```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

}
```

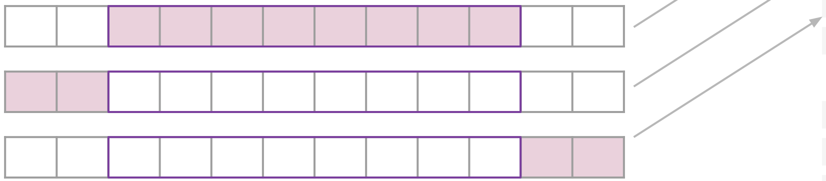


If these are the elements that the threads in a block are going to apply the stencil on, the threads should also have access to a **"halo" of elements** left and right equal to the stencil radius

1-D stencil kernel

- Input elements are read into shared memory

e.g. if BLOCK_SIZE = 8 & stencil radius = 2

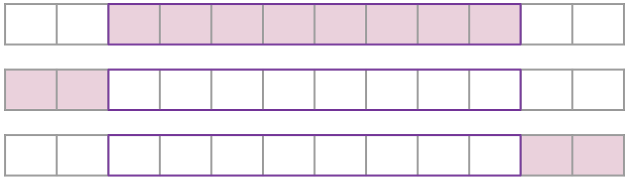


```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;

}
```

1-D stencil kernel

- Input elements are read into shared memory

e.g. if BLOCK_SIZE = 8 & stencil radius = 2



```
__global__ void stencil_1d(int *in, int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
      temp[lindex - RADIUS] = in[gindex - RADIUS];
      temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS; offset <= RADIUS; offset++)
      result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;

}
```

Exercise

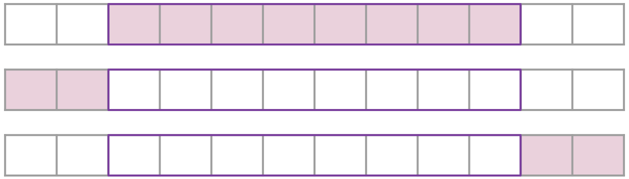
Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

What do you observe??

1-D stencil kernel

- Input elements are read into shared memory

e.g. if BLOCK_SIZE = 8 & stencil radius = 2



```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;

}
```

Exercise

Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

Data race !!!

Shared memory and synchronization

- Threads in a block don't necessarily execute the same instruction simultaneously!
  - Only threads in the same warp execute instructions simultaneously
- The program does not know a priori the desired way of how threads should execute instructions
  - Outcome depends on timing of the different threads
  - In our example there were cases where the stencil was applied before the values were loaded into shared memory
- To address this race condition we can use the **__syncthreads()** primitive:
  - synchronizes all threads within a block

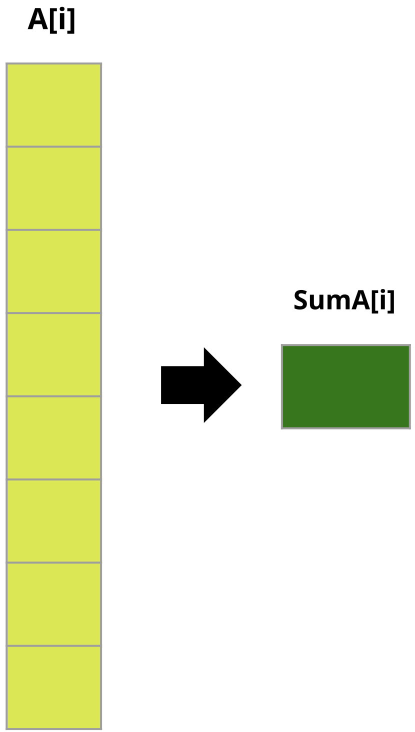
Exercise

Let's try and add the **_syncthreads()** primitive and see what we get!

Atomic operations

Atomic operations

- Useful when modifying the same value in memory from different threads :
  - Are used to prevent race conditions in multithreaded applications
  - Read-modify-write cannot be interrupted
    - Appear to be one operation
- Atomics are special hardware instruction on NVIDIA GPUs e.g.:
  - atomicAdd/Sub (Add or subtract)
    - e.g. syntax : atomicAdd(int* address, int val);
  - atomicMax/Min (Find max or min)
  - atomicExch/CAS (Swap or conditionally swap variables)
    - e.g. syntax : atomicCAS (&addr, compare, value)
  - atomicAnd/Or/Xor (bitwise operations)
  - ...

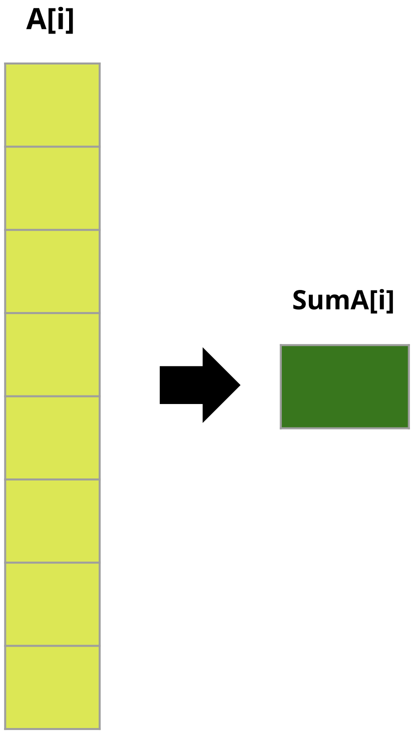


Adding elements in a vector

Let's start by writing a CUDA kernel that calculated the sum of the elements of a vector :

```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
       *sum +=A[idx];
   }
}
```

- There are 3 instructions that will be executed :
  - Load the value of A for each thread
  - Read the value of c
  - Modify the value of c

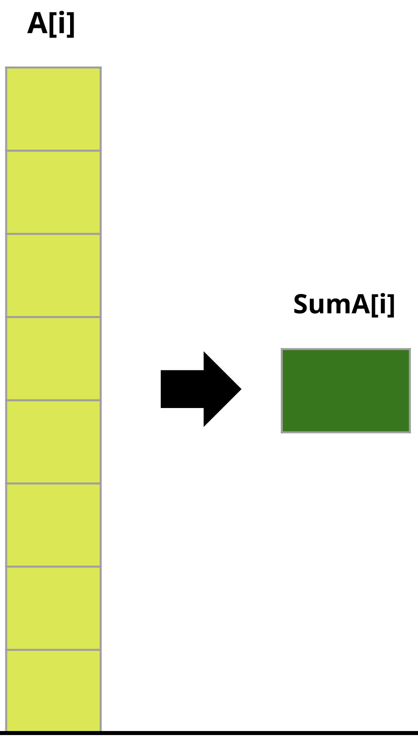


Adding elements in a vector

Let's start by writing a CUDA kernel that calculated the sum of the elements of a vector :

```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
       *sum +=A[idx];
   }
}
```

- There are 3 instructions that will be executed :
  - Load the value of A for each thread
  - Read the value of c
  - Modify the value of c



Exercise

Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

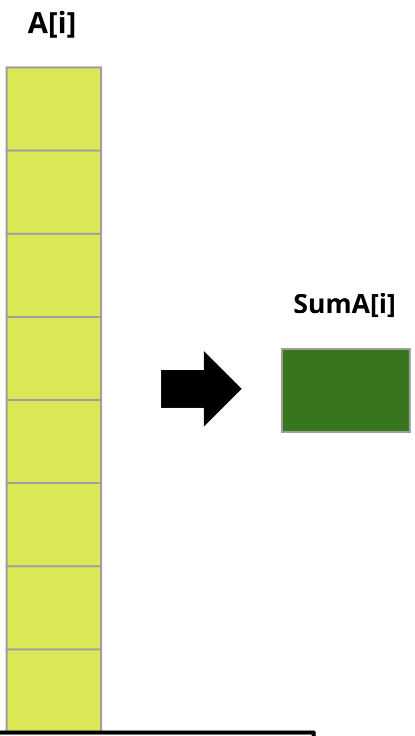
Can you guess what will happen?

Adding elements in a vector

Let's start by writing a CUDA kernel that calculated the sum of the elements of a vector :

```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
       *sum +=A[idx];
   }
}
```

- There are 3 instructions that will be executed :
  - Load the value of A for each thread
  - Read the value of c
  - Modify the value of c



Exercise

Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

The behaviour of the kernel is unpredictable - the read/writes can happen in random orders

The sum is incorrect!!!

Adding elements in a vector

Let's try to use atomicAdd to sum the vector elements :

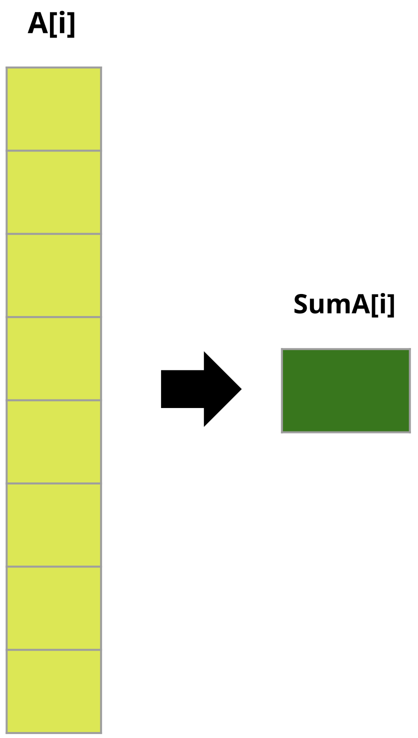
```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
      atomicAdd(sum,A[idx]);
   }
}
```

Lets use atomicAdd

Each read-modify-write access cannot be interrupted

Exercise

Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

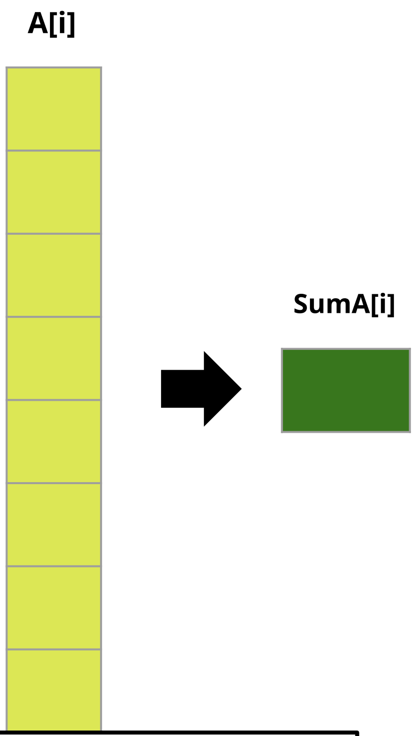


Adding elements in a vector

Let's try to use atomicAdd to sum the vector elements :

```
__global__ void add_array(float* A, float* sum) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   if (idx < N) {
      atomicAdd(sum,A[idx]);
   }
}
```

Each read-modify-write access cannot be interrupted



Exercise

Lets try this out!
You can copy this code into a .cu file and try to run it.
Remember: To compile first set up your environment and then :
nvcc myscript.cu -o myscript
./myscript

Now the sum is correct!!

Default CUDA stream

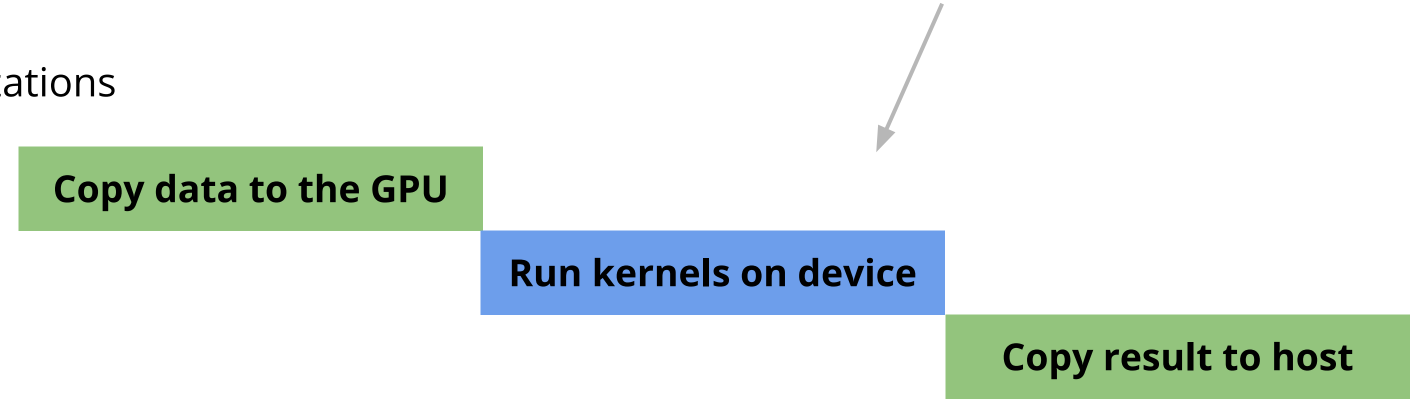
What is a Stream?

- Sequence of commands that execute in order
  - Executed on the device in the order in which they are issued by the host code
- A Stream can execute various types of commands.
  - Kernel invocations
  - Memory transmissions
  - Memory (de)allocations
  - Memsets
  - Synchronizations

What is a Stream?

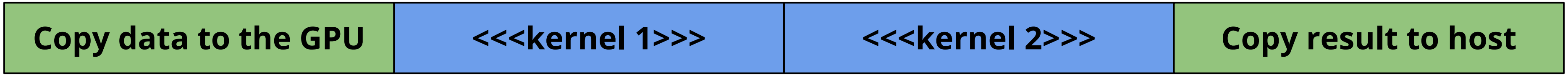
- Sequence of commands that execute in order.
  - Executed on the device in the order in which they are issued by the host code
- A Stream can execute various types of commands.
  - Kernel invocations
  - Memory transmissions
  - Memory (de)allocations
  - Memsets
  - Synchronizations

Any instruction that runs in a stream must complete before the next can be issued



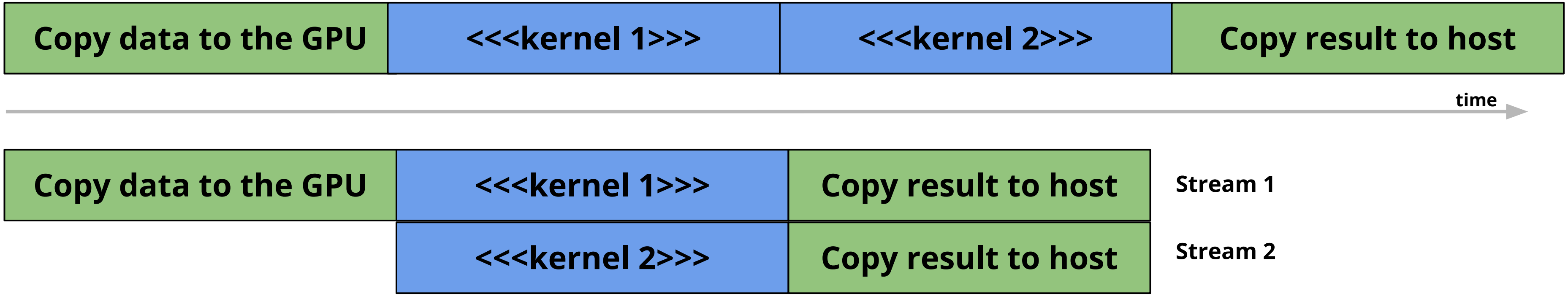
CUDA default stream

- CUDA has what we call a **default stream**
  - By default all CUDA kernels run in this default stream
- The default stream is blocking :
  - Other commands are not executed in parallel on the device



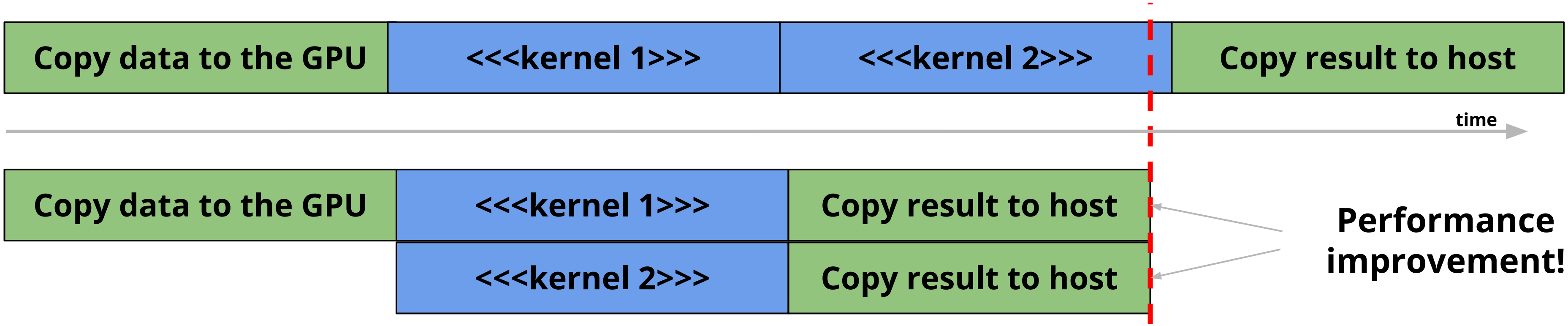
CUDA default stream

- In CUDA, we can also run multiple kernels on different streams concurrently
    - Non-default CUDA streams!



CUDA default stream

- In CUDA, we can also run multiple kernels on different streams concurrently
  - Non-default CUDA streams!



Wrapping-up

Overview of today's lecture

- We learnt about shared memory :
  - Can be static or dynamic
  - Reduces the number of loads from the global memory
  - Important efficiency consideration
- We learnt about atomic operations
  - Useful to avoid race conditions and unpredictable kernel behaviour
- Learnt about the default CUDA stream

Assignment for next week

- Assignment can be found here (**Week 3**) :

  https://github.com/ckoraka/tac-hep-gpus

- To clone :
  - git clone git@github.com:ckoraka/tac-hep-gpus.git
- **Due Friday February 24th**
- Please upload assignment here :
  - https://pages.hep.wisc.edu/~ckoraka/assignments/TAC-HEP/
  - Upload only 1 .pdf file with all exercises
  - If you also have your code on git, please add the link to your repository in the pdf file you upload.

During the next weeks

- We will hear a lot more about CUDA streams
- We will learn how to profile CPU & GPU
- We will learn about managed memory in CUDA
- We will get familiar with Alpaka

Back-up

Resources

1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link
7. CUDA/C++ best practices guide link
8. NVidia DLI teaching kit link