GPU programming lab

CUDA programming on the Jetson TX2

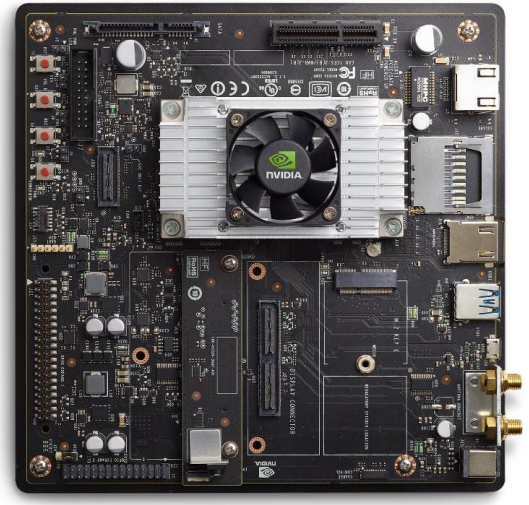
Introduction

In this lab, you will learn to:

- Get familiar with the NVIDIA TX2
- Allocate and transfer memory from/to the GPU device
- Create simple GPU kernels
- Compile and run your kernels on the GPU
- Use the Unified Memory Address
- Use shared memory to optimize your code

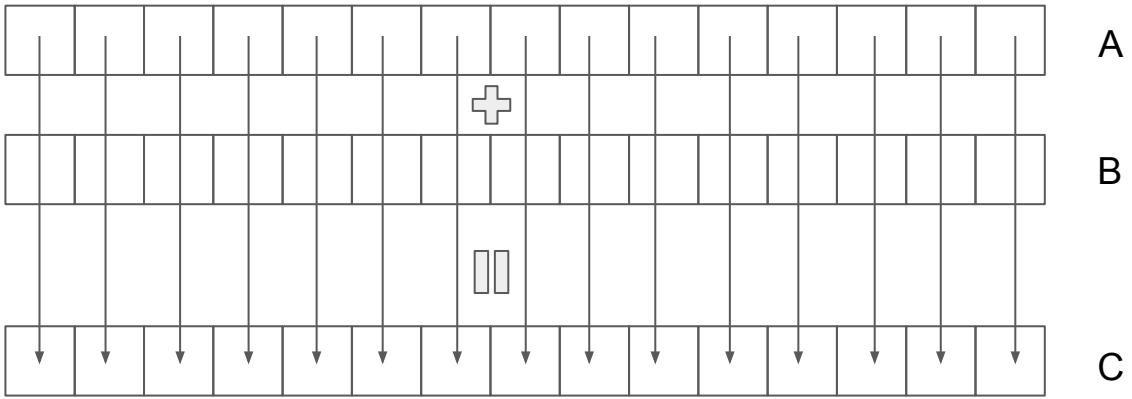
NVIDIA Jetson TX2

- **GPU**: NVIDIA Pascal architecture, 256 CUDA cores
- **CPU**: NVIDIA Denver 2 cores + ARM A57 4 cores
- CPU and GPU share 8 GB 128 bit LPDDR4
- CUDA v8.0, compute capability 6.2
- Comes with optimized OpenCV and multiple CUDA libraries for machine learning



Part 1: vector add

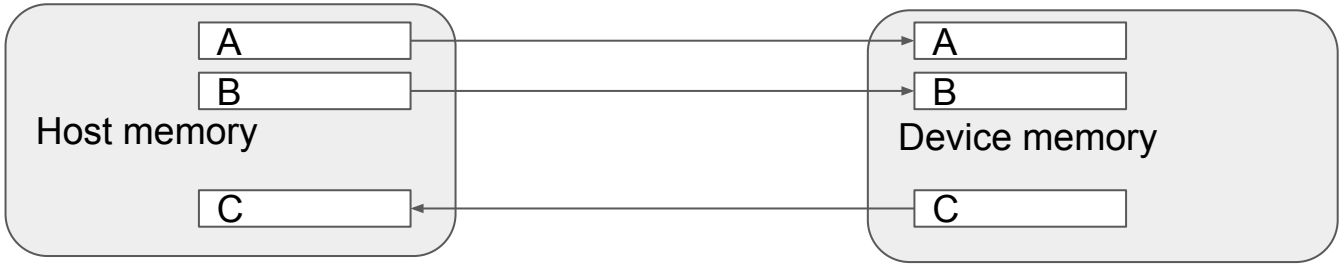
C = A + B



Part 1: vector add

First you need to manage memory transfers:

1. Allocate memory needed for GPU computation
2. Transfer the input and output back and forth

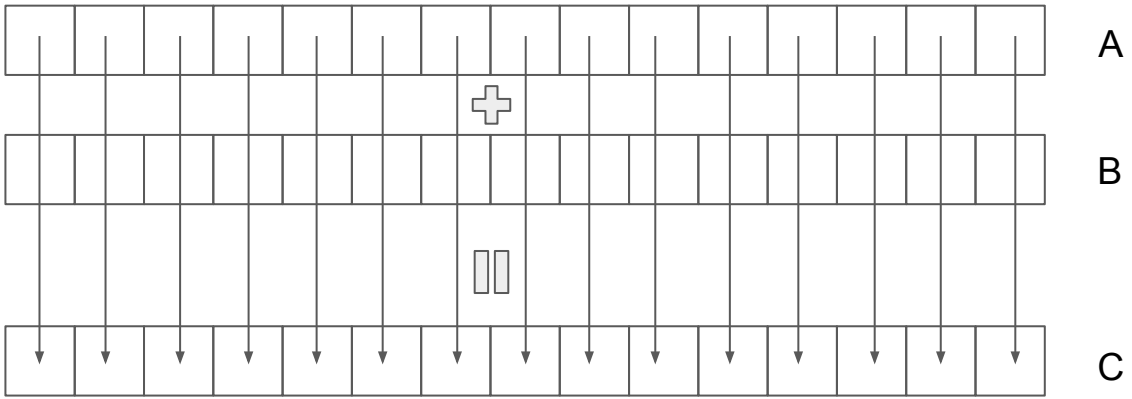


We will see later how to avoid separating host and device memory

Part 1: vector add

Each thread:

- Reads from A
- Reads from B
- Add values
- Store result in C



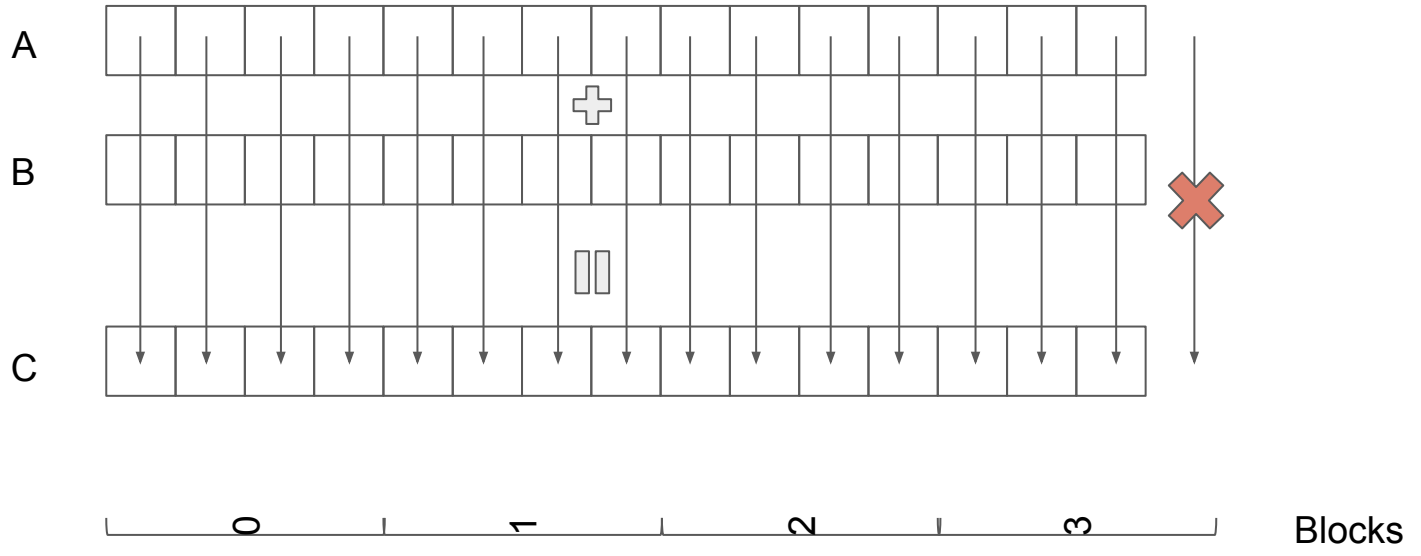
Kernel will run with:

- N blocks
- M threads per block
- What if N*M > vector size?

Part 1: vector add

- Example:
- N = 4 blocks
- M = 4 threads per block

Protect out-of-bounds access with an *if* statement!



Part 1: vector add

1. Write a *vectorAdd* kernel
2. Allocate device memory for A,B,C
3. Transfer A and B from Host to Device
4. Run the kernel with 256 threads per block
5. Transfer C from Host to Device
6. Free the cuda memory

Complete lab5/vectorAdd/vectorAdd.cu

Short reference:

```
__global__ void kernel_name(...){...}

blockDim.x
blockIdx.x
threadIdx.x
```

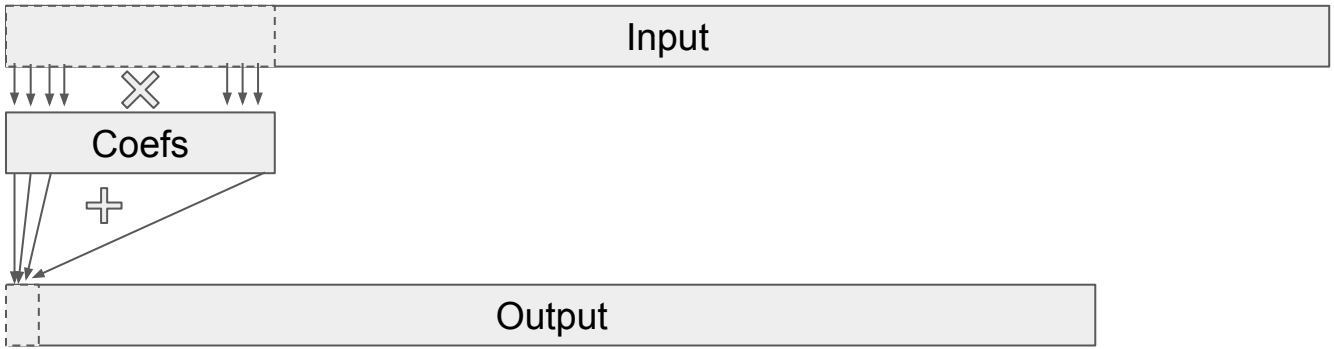
```
cudaError_t cudaMalloc(void ** devPtr, size_t size)

cudaError_t cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost

kernel_name<<<grid_size, block_size>>>(...);

cudaError_t cudaFree(void * devPtr)

cudaSuccess
```

To compile:

```
nvcc vectorAdd.cu -o vectorAdd
```

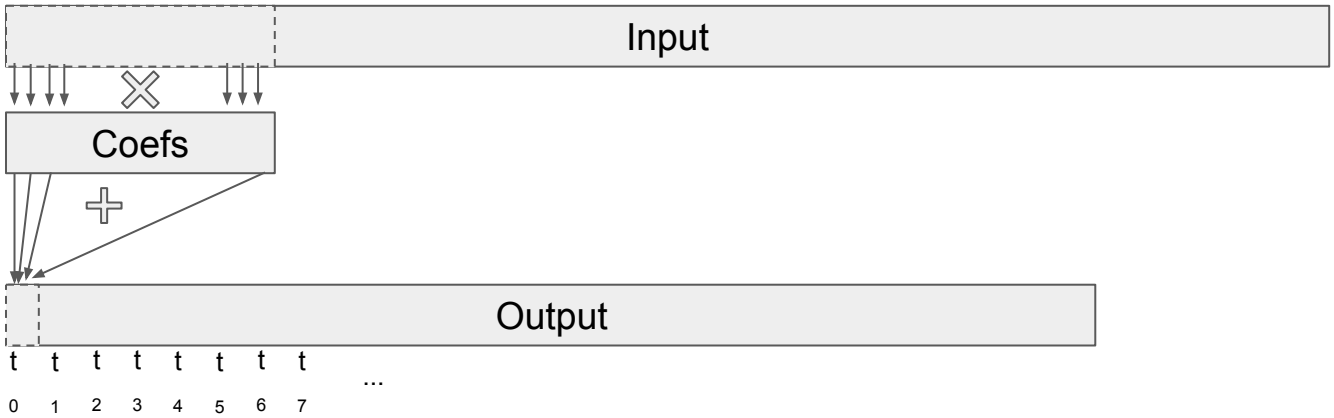
Part 2: FIR filter

FIR filter from lab3

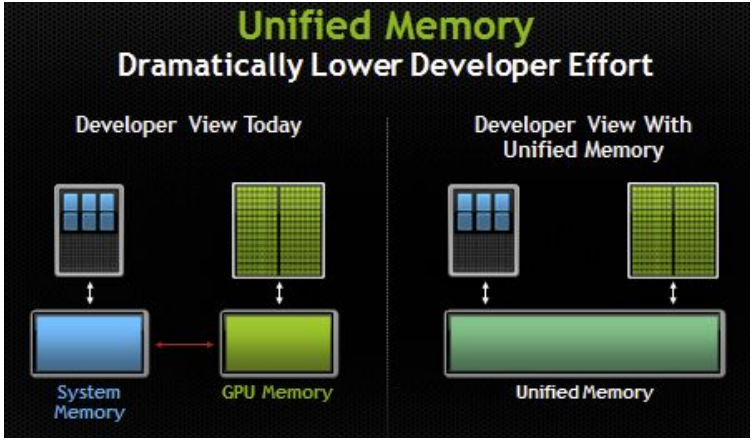


Part 2: FIR filter

On CUDA: 1 thread per output data



Part 2: FIR filter



In this application, we will get rid of cudaMemcpy

CPU and GPU are on the same chip,
share the same memory

Let’s use CUDA Unified Memory:

- Create a unified address space for host and devices
- A pointer is valid on host and device space
- The driver takes care of moving memory as needed

Part 2: FIR filter

1. Replace *malloc* by *cudaMallocManaged* (and *free/cudaFree*)
2. Complete the kernel *fir_kernel1*
3. No need to allocate/copy device memory!
4. Set X dimension of CUDA grid (hint: use *divup*)
5. Launch the kernel
6. Compile and run your code (make)

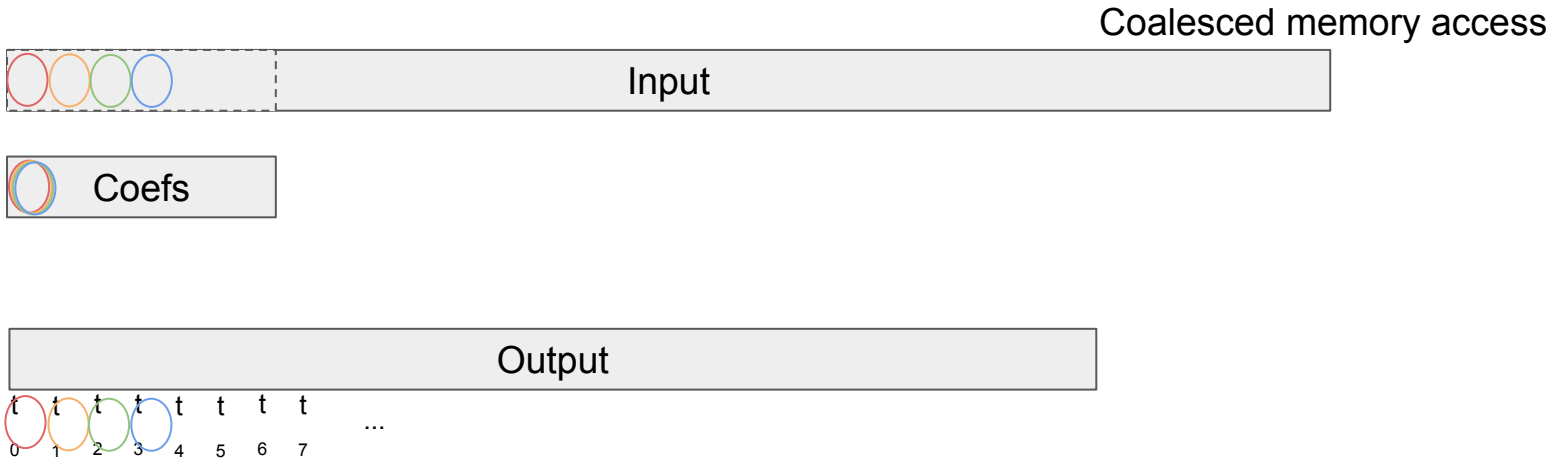
Complete **lab5/fir/src/main.cpp** and **lab5/fir/src/fir_gpu.cu**

Short reference:

```
cudaError_t cudaMallocManaged ( void** devPtr, size_t size, unsigned int flags = cudaMemAttachGlobal )
```

Part 2: Shared memory optimization

FIR filter memory accesses

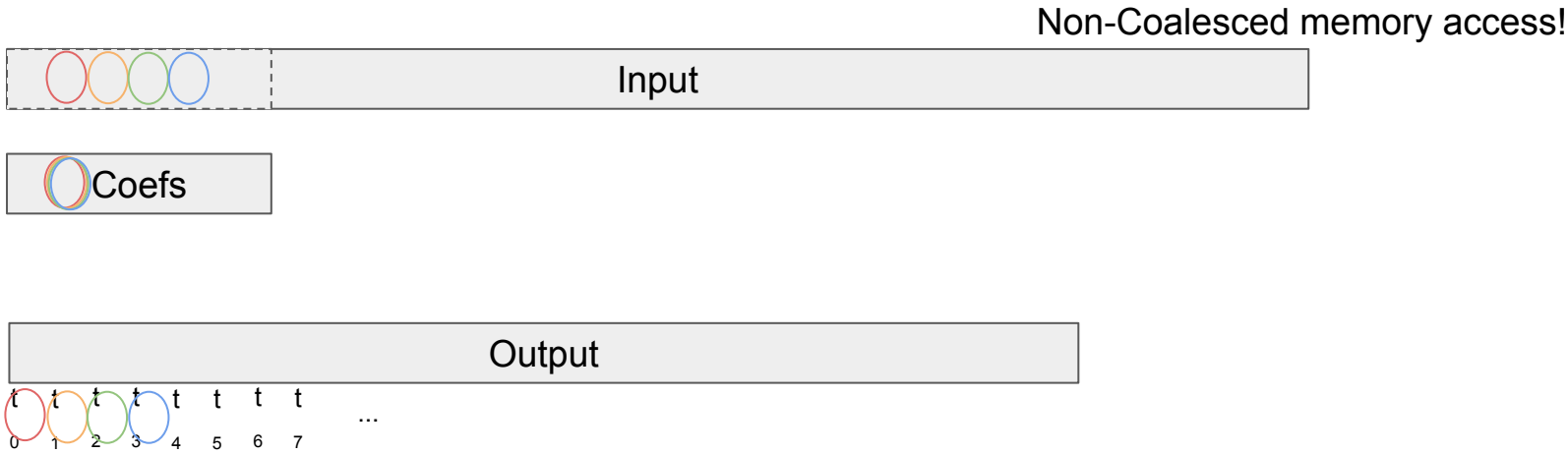


Filter iteration 1

(For illustration, assume warp size = 4)

Part 2: Shared memory optimization

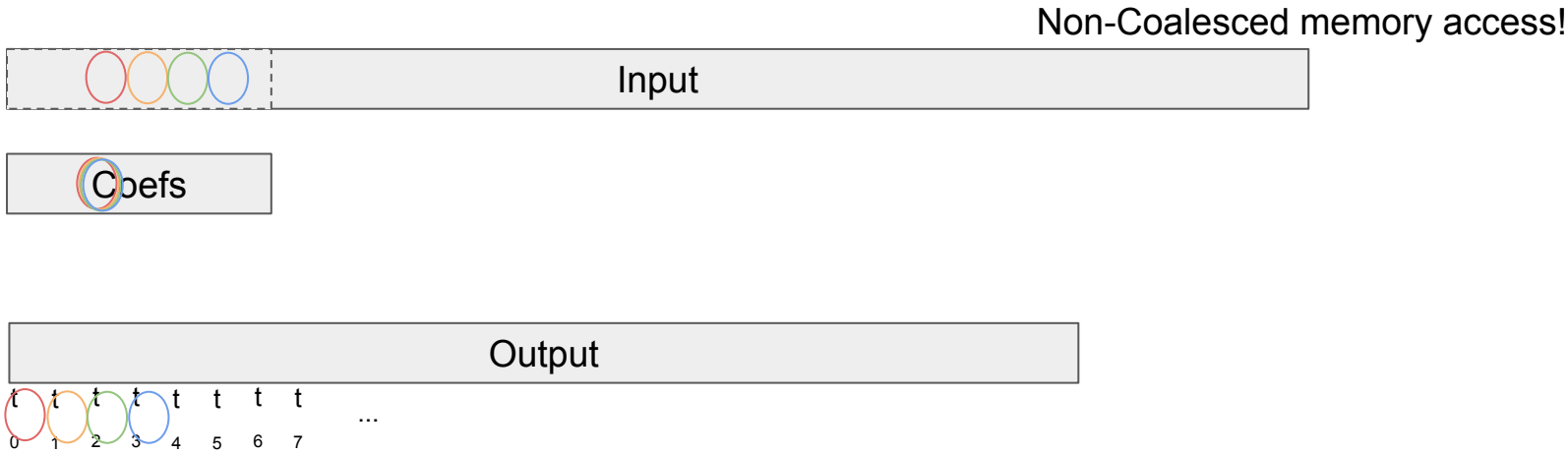
FIR filter memory accesses



Filter iteration 2

Part 2: Shared memory optimization

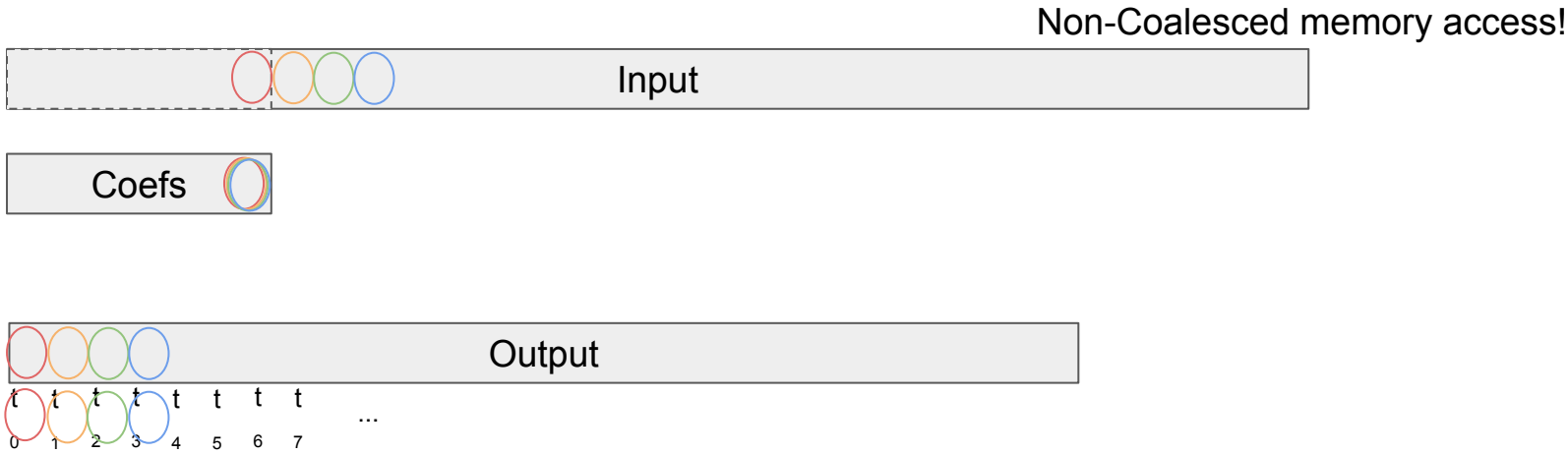
FIR filter memory accesses



Filter iteration 3

Part 2: Shared memory optimization

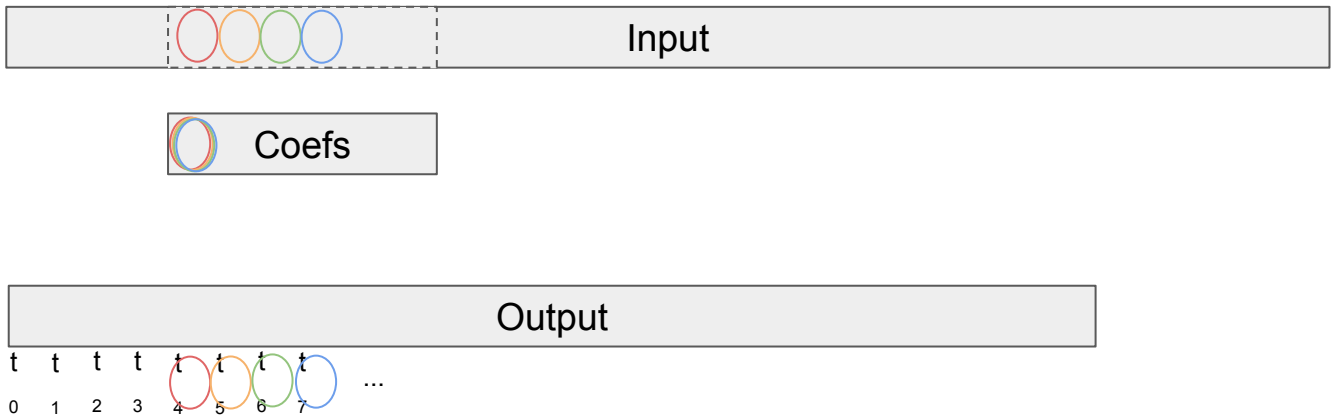
FIR filter memory accesses



Filter iteration 63

Part 2: Shared memory optimization

- Coefs: Next warp of threads will the same values
- Input: Within the same warp, and across multiple warps, reusing the same values
- Let’s use shared memory

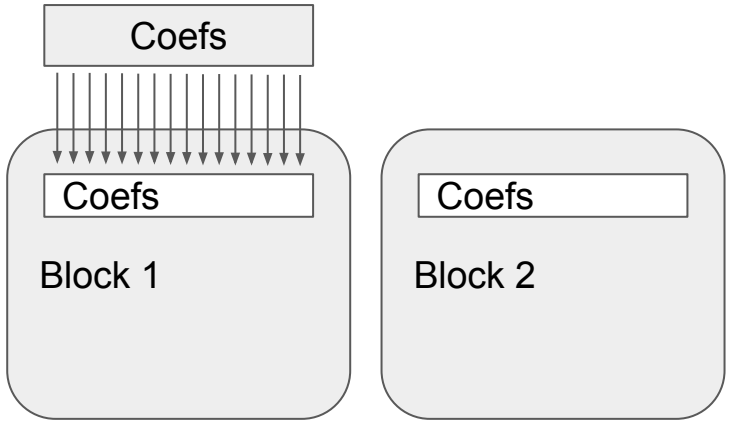


Part 2: Shared memory optimization

Step 1: Use shared memory for coefficients

1. From now on, fix coefficient array size to 64
2. Create blocks of 64 threads
3. Each thread loads 1 coef into shared memory
4. Synchronize all threads
5. Then all threads use the coefs from shared memory

Complete *fir_kernel2*



Short reference:

```
blockDim.x  blockIdx.x  threadIdx.x

__shared__

__syncthreads();
```

Part 2: Shared memory optimization

Step 2: Use shared memory for input

1. Keep loading coefficients into shared mem
2. Also load inputs into shared mem
3. Each thread may need to load more than one input

Complete *fir_kernel3*