Technical University of Munich

Lecture 4: CUDA execution and memory models

Informatik elective: GPU Computing

Pratik Nayak

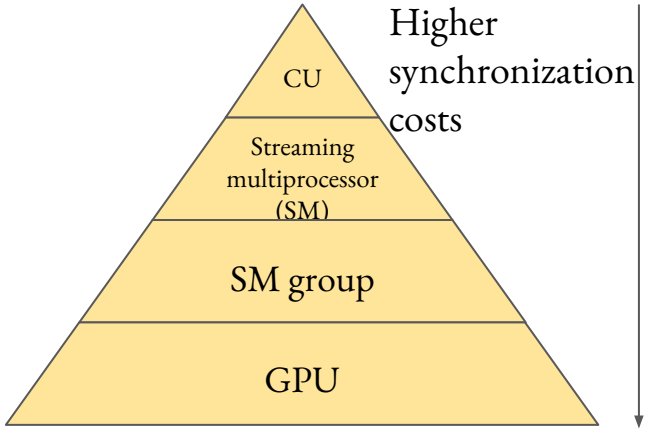
In this session

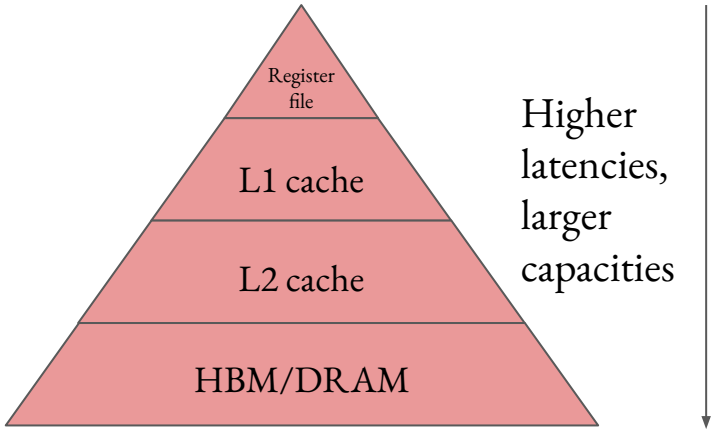
- Recap: CUDA execution model basics
  - SMs, warps; thread blocks
- CUDA memory model
  - Shared memory/L1 cache.
  - Registers.
- Examples:
  - Extending the convolution example.
  - Parallel reduction

Recap: GPUs: High throughput shared memory architectures

- Scalable array of multi-threaded Streaming Multiprocessors (SMs)
- Each SM contains multiple functional units (SIMD units):
  - Compute: FP64, FP32, INT32, Tensor core, special function units
  - Memory: Load/Store units
- Each SM also contains:
  - Local shared memory (L1 cache)
  - Register file: Thread-local registers of fixed size per SM.
  - A warp (SIMD lane) scheduler that issues warps to the available functional units.
- Multiple of such SMs in one GPU: 128 (108) in A100, 80 in V100 etc.

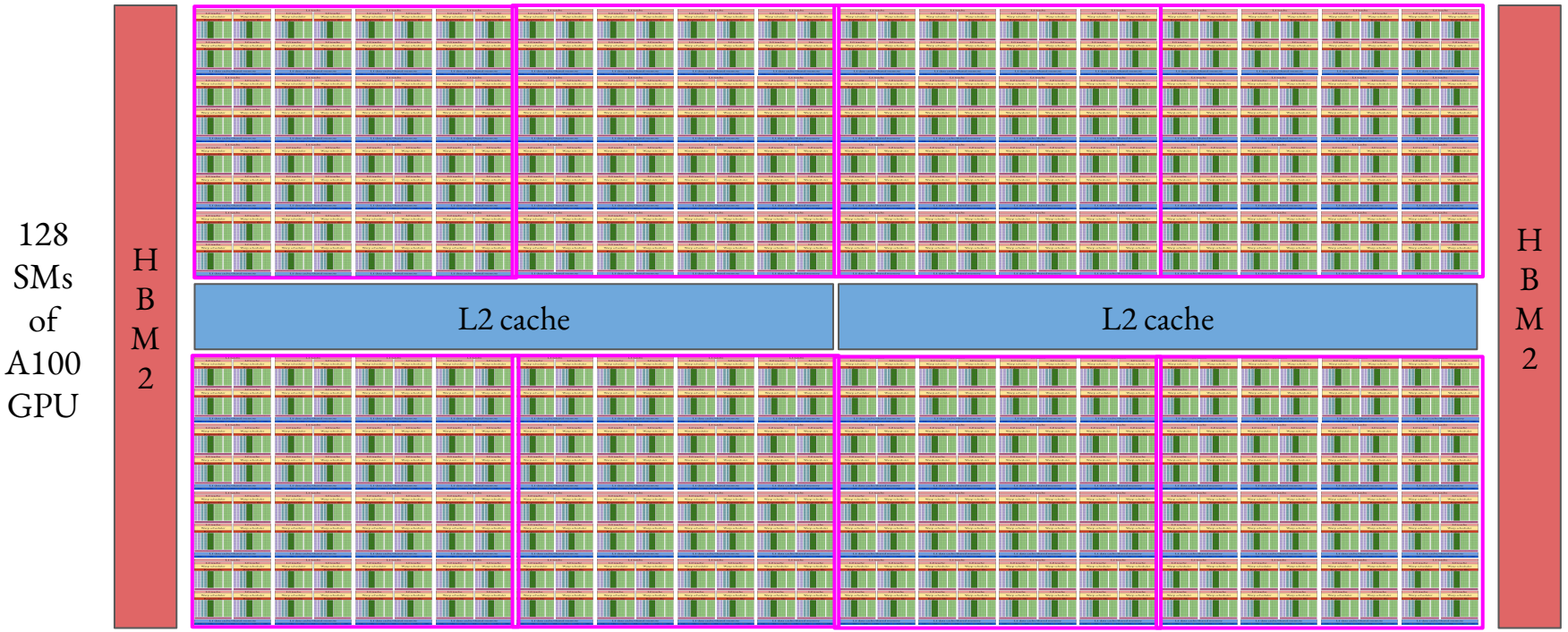
Recap: GPUs: Highly hierarchical memory and compute



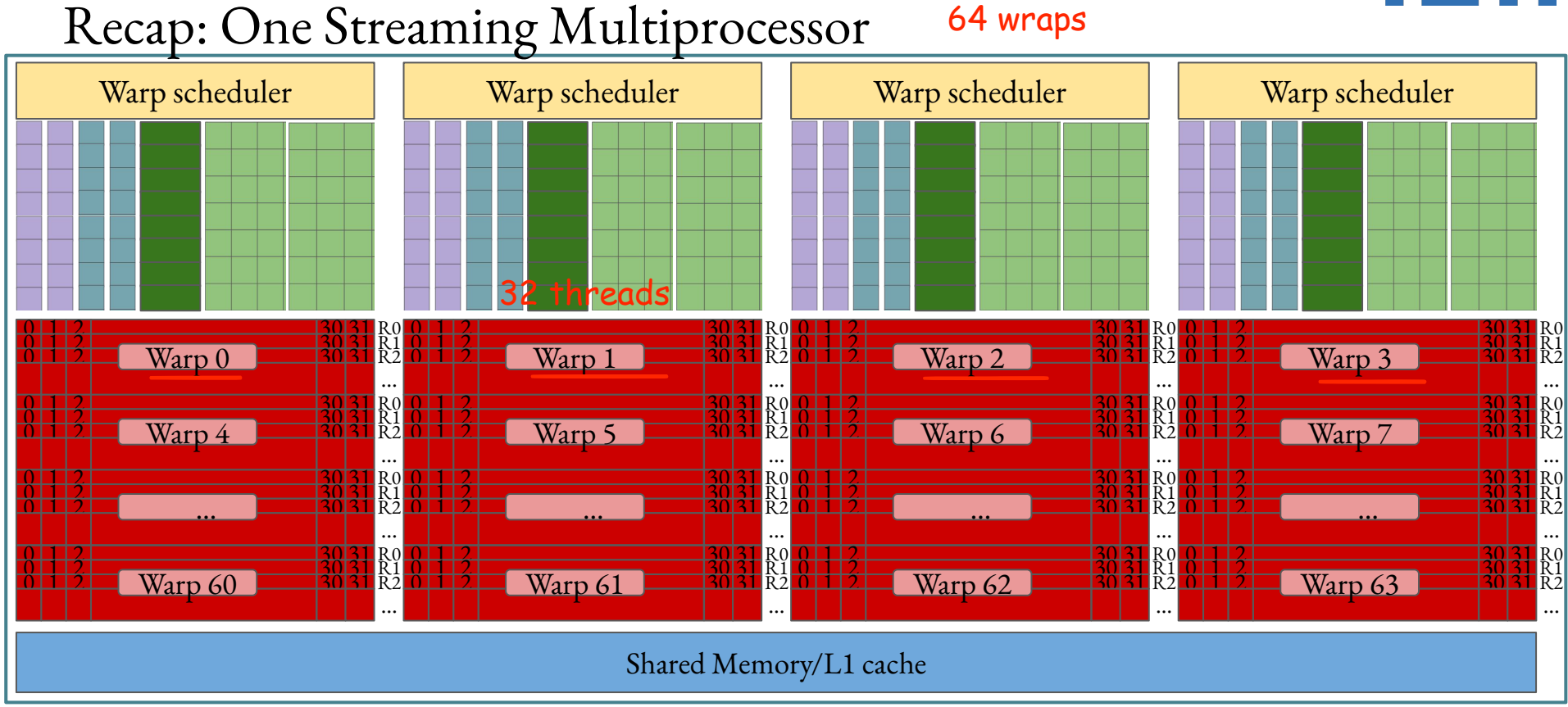


- Highly hierarchical memory and compute.
- Larger capacities/more parallelism → higher latencies/higher synchronization costs
- Reframe algorithms to reduce serial components → Maximize available parallelism

Recap: GPU: Array of Streaming Multiprocessors

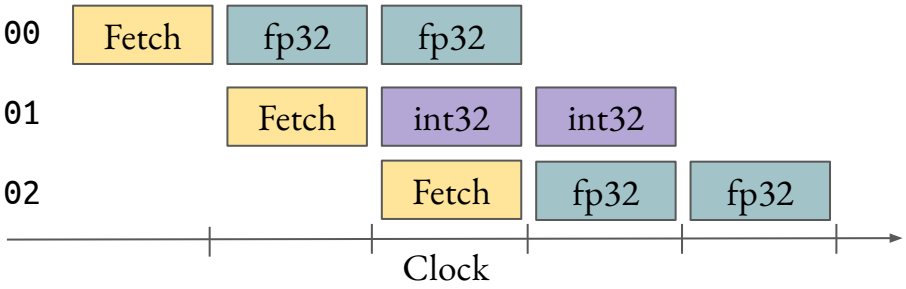


Recap: One Streaming Multiprocessor

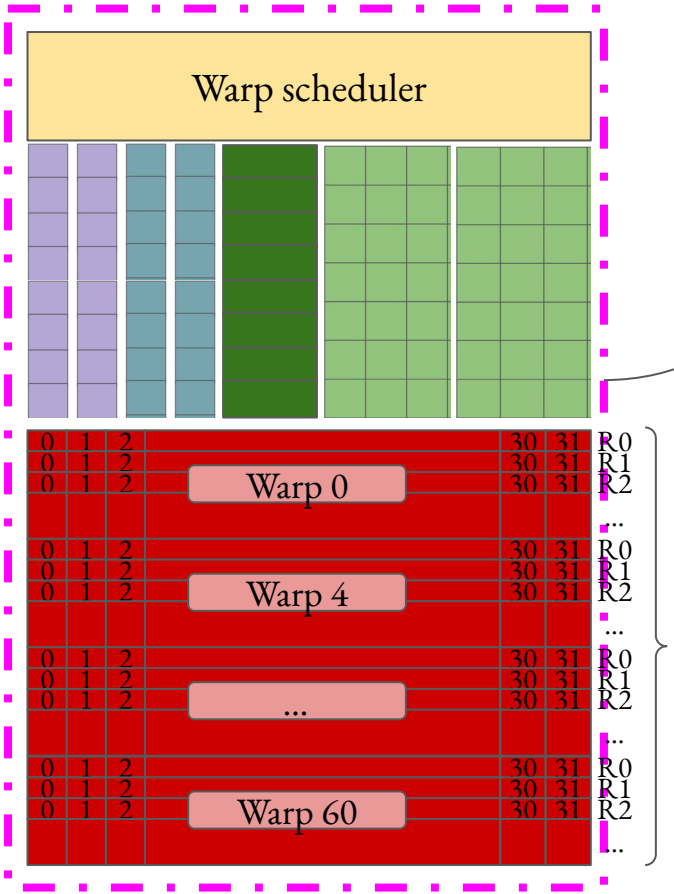


Recap: Warp scheduling

```
00  fp32   mul r0 r1 r2
01  int32  add r3 r4 r5
02  fp32   add r6 r7 r8
```



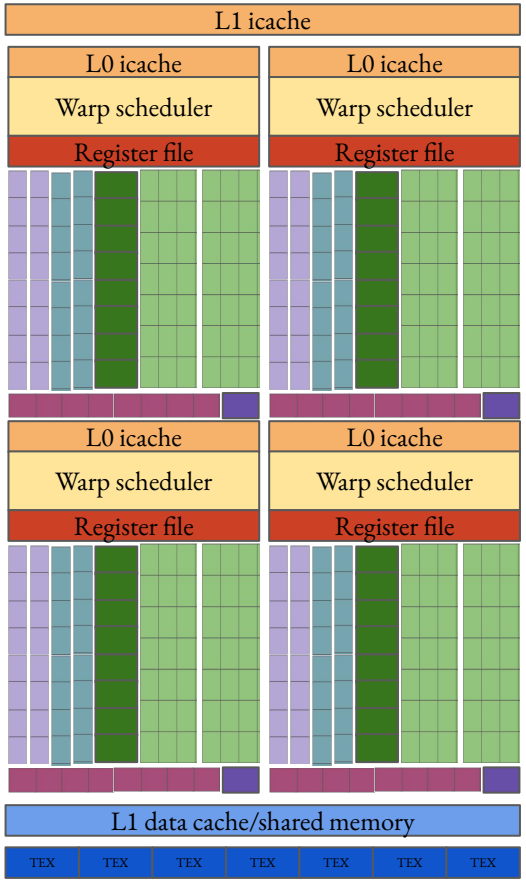
- Each instruction is run by the whole warp
- All 32 threads of the warp run the same instruction
- Thread divergence → performance hit



Recap: Streaming Multiprocessor

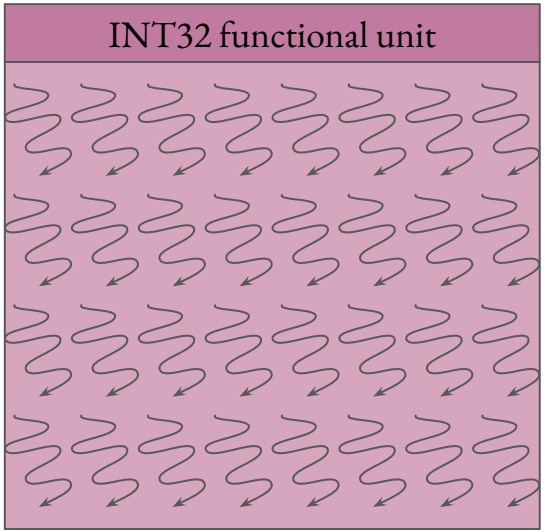
- Many functional units of SIMD length 32 that perform an operation every x clock cycle.
- A scheduler that schedules work to the appropriate functional unit
- Each SM shares an L1 cache
- Number of SIMD units that can be scheduled at once limited by the register file size.

INT32 functional unit	16 FMA per clock-cycle
FP32 functional unit	16 FMA per clock-cycle
FP64 functional unit	8 FMA per clock-cycle
Tensor core functional unit	
Load/store units	
Special function unit	



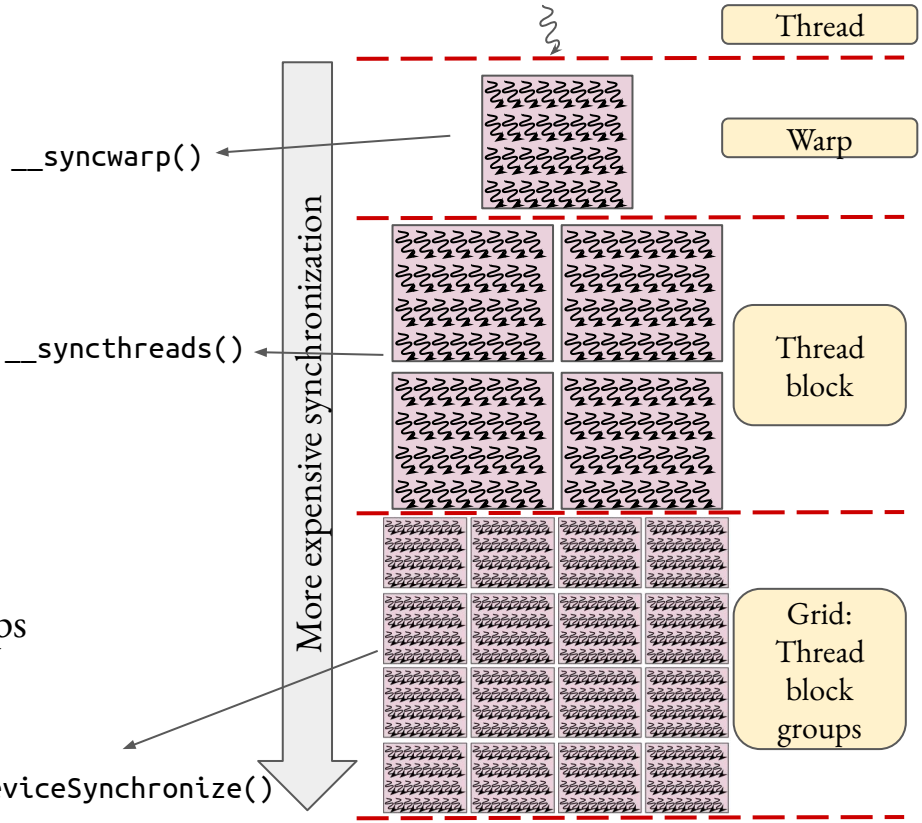
Recap: GPU SIMD units

- SIMD units processing a certain number of threads (in a lock-step fashion).
  - 32 for NVIDIA GPUs: called a ***warp*** *(only a software concept)*.
  - Operate on same instruction, but on different data
- Different types of these SIMD units for different instruction types.
- One 32-length SIMD operation per x clock cycle.
  - x depends on the functional unit



CUDA execution hierarchy

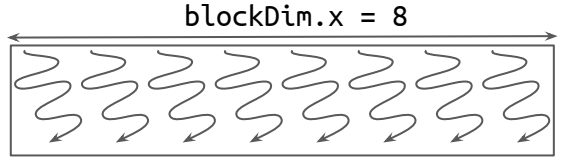
- Finest level: thread
- Warps: Groups of 32 threads → Maps to one functional unit.
- Thread block: Groups of threads (maximum size fixed for a architecture) → Maps to one SM.
- Grid: Groups of thread blocks.
- Advanced (in later lectures):
  - Finer control available with cooperative groups
  - Independent thread scheduling (after Volta)



CUDA: Grid and thread block organization

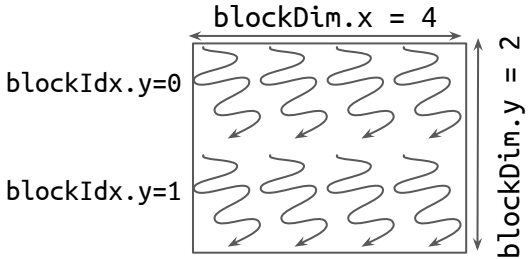
- CUDA logically organizes the threads in a thread-block in a 3D fashion.
- 3D grid of thread blocks, 3D grid of threads in thread blocks.
- Can access global thread id with block strides.

```
B = dim3(8,1,1)
G = dim3(Nb,1,1)
```



1D: global thread id: `blockIdx.x*blockDim.x + threadIdx.x`

```
B = dim3(4,2,1)
G = dim3(Nb,1,1)
```

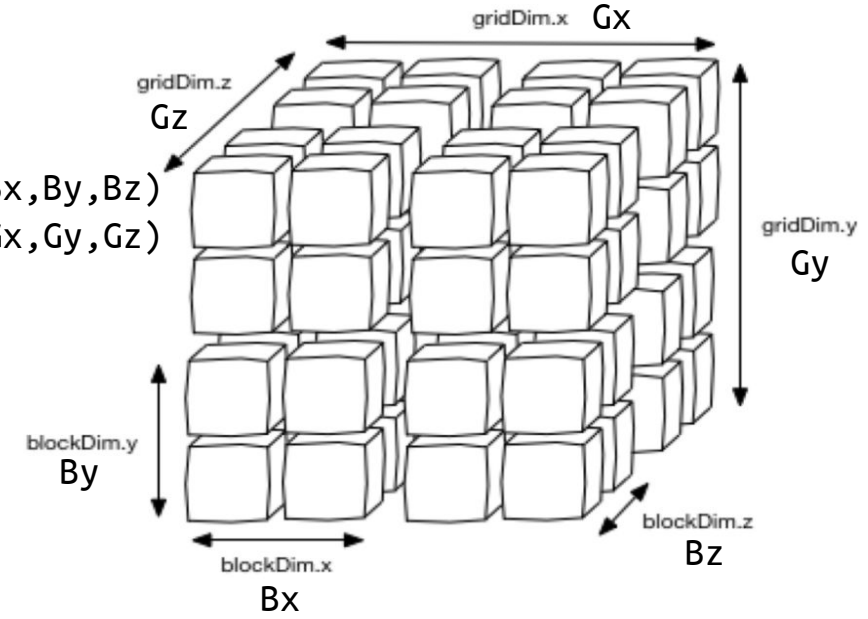


2D: global thread id: `(blockIdx.y * gridDim.x + blockIdx.y) * blockDim.x + threadIdx.x`

CUDA: Grid and thread block organization

- CUDA logically organizes the threads in a thread-block in a 3D fashion.
- 3D grid of thread blocks, 3D grid of threads in thread blocks.
- Can access global thread id with block strides.
- Can be a useful abstraction when working with multi-dimensional objects.

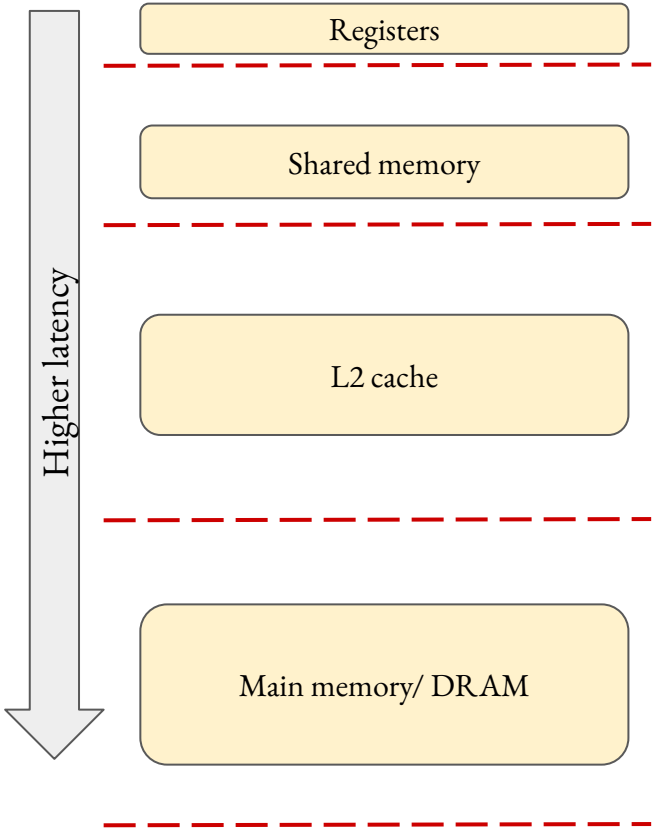
```
B = dim3(Bx,By,Bz)
G = dim3(Gx,Gy,Gz)
```



3D: global id:
```
blockId = (gridDim.x * gridDim.y * blockIdx.z + gridDim.x *
    blockIdx.y + blockIdx.x)
threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
  + (threadIdx.z * (blockDim.x * blockDim.y)) +
    (threadIdx.y * blockDim.x) + threadIdx.x
```

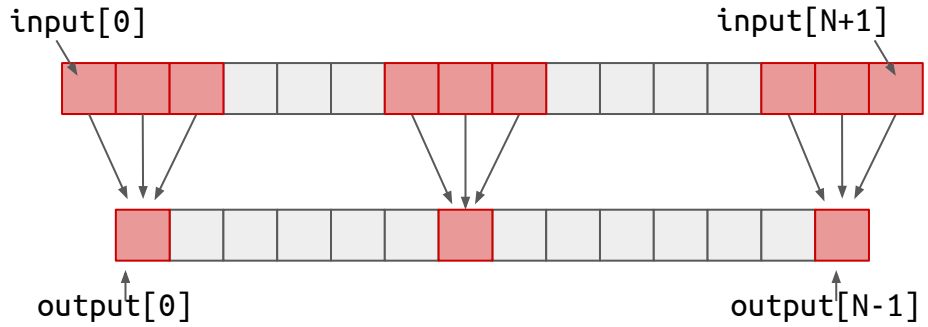
CUDA memory hierarchy

- Finest level: registers, shared within one warp (32 threads)
- Shared memory/L1 cache: shared by all threads in a thread-block.
- L2 cache: Smaller capacity memory shared between all SMs.
- Main memory: Large capacity memory shared between all SMs.



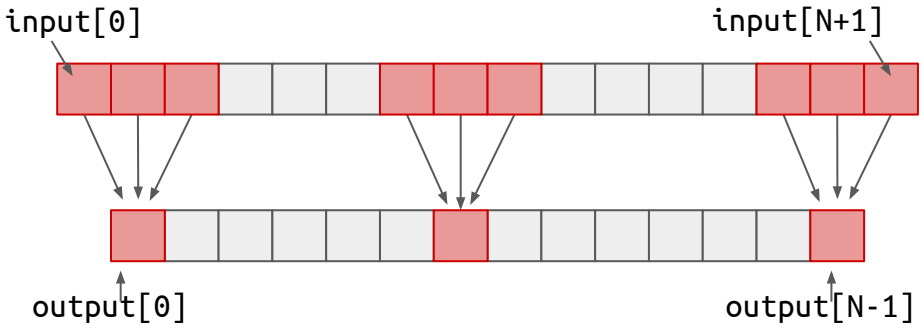
Convolution example: Thread to element map

- Each thread computes one element.
- No dependencies between threads.
- Launch `N/block_size` number of thread blocks, each of size `block_size`.




```
output[i] = (input[i] + input[i+1] + input[i+2])/3.0
```

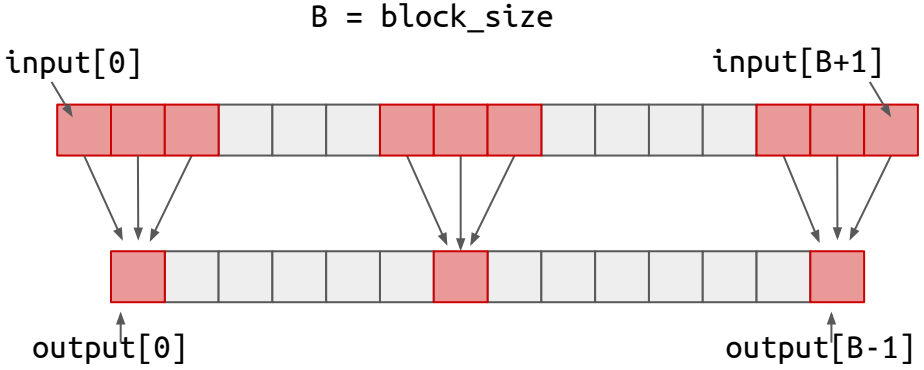
Convolution example: shared memory




```
output[i] = (input[i] + input[i+1] + input[i+2])/3.0
```

```
...


template <typename ValueType>
__global__ void convolution(const ValueType* input,
ValueType* output)
{
	int tid = blockIdx.x * blockDim.x + threadIdx.x;

	ValueType res = 0.0;
	for (int i = 0; i < 3; ++i) {
		res += input[tid + i];
	}
	output[tid] = res / 3.0;
}
```

Convolution example: shared memory



- Explicitly load thread-block data into shared memory
- Need to synchronize within thread-block to prevent read-after-write hazards
- All computations now only need shared memory data. No global memory reads.

```
template <typename ValueType>
__global__ void convolution(const ValueType* input,
ValueType* output)
{
	int tid = blockIdx.x * blockDim.x + threadIdx.x;
	__shared__ ValueType stencil[B+2];
	if (threadIdx.x < 2) {
	stencil[default_block_size + threadIdx.x] =
		input[tid + default_block_size];
	}
	__syncthreads();

	ValueType res = 0.0;
	for (int i = 0; i < 3; ++i) {
		res += stencil[threadIdx.x + i];
	}
	output[tid] = res / 3.0;
}
```

Shared memory: Static and dynamic

- Static shared memory: If the size of the shared memory needed is known at compilation time.

```
// Inside kernel
__shared__ ValueType var[size];
```

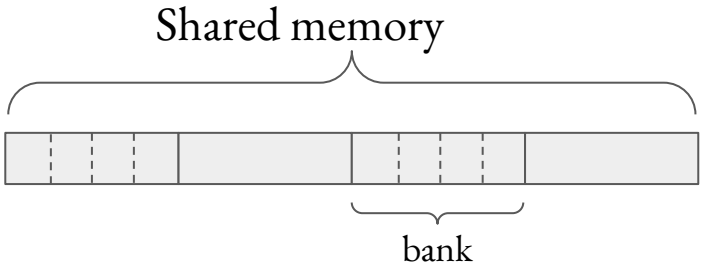
- Dynamic shared memory: If the shared memory that needs to be allocated depends on runtime parameters (example: input size)

```
// Inside kernel
extern __shared__ char var[];

ValueType* shared_var = reinterpret_cast<ValueType*>(var);
...

// kernel launch
kernel<<<num_blocks, block_size, shmem_size_in_bytes>>>(...)
```

DEMO

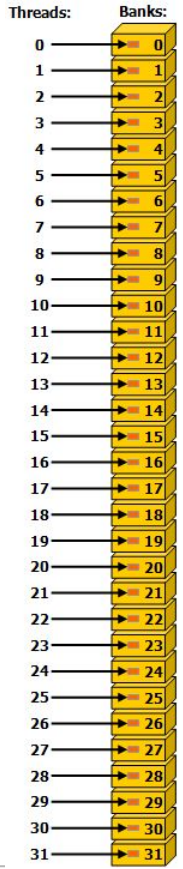
Avoiding shared memory bank conflicts

- Shared memory is divided into equal size chunks (banks), each of which can be accessed concurrently.
- A load or store of n addresses in n distinct memory banks has an effective speedup of n for bandwidth compared to one bank.
- Each bank has bandwidth: 32 bits per clock-cycle



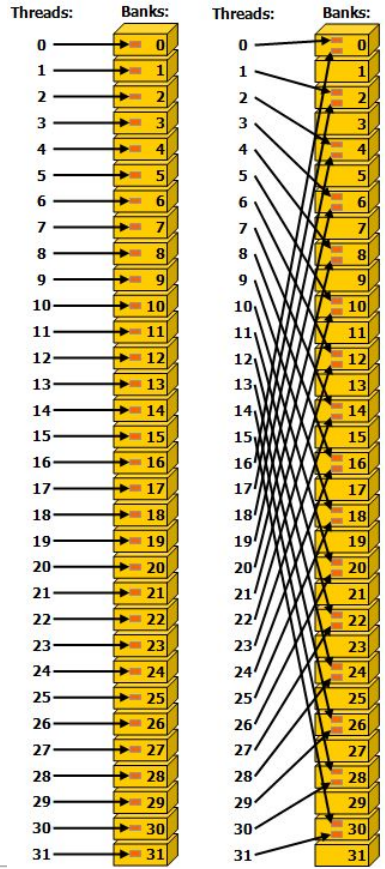
Avoiding shared memory bank conflicts

- (No bank-conflict) Left: Linear addresses with stride 1.



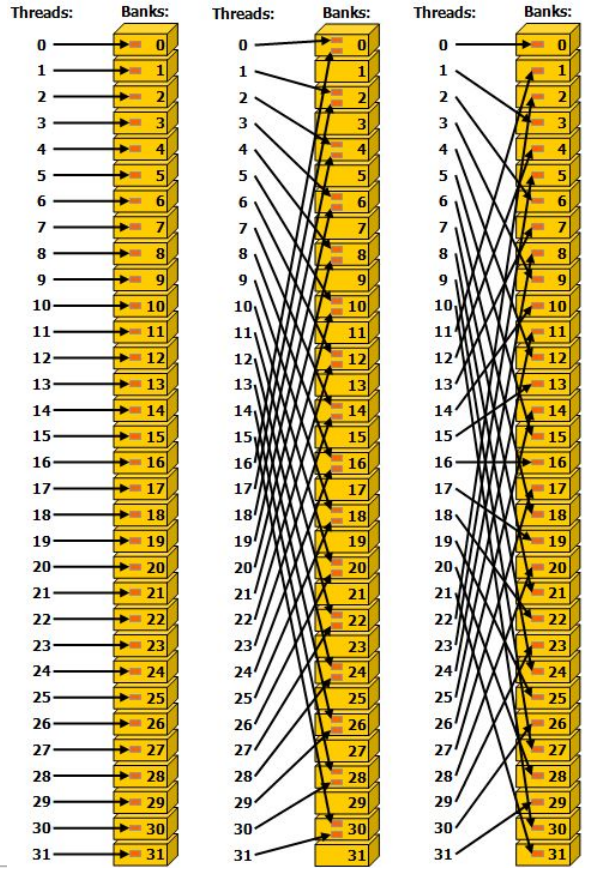
Avoiding shared memory bank conflicts

- (No bank-conflict) Left: Linear addresses with stride 1.
- (Two-way bank conflict) Middle: Linear address with stride 2.



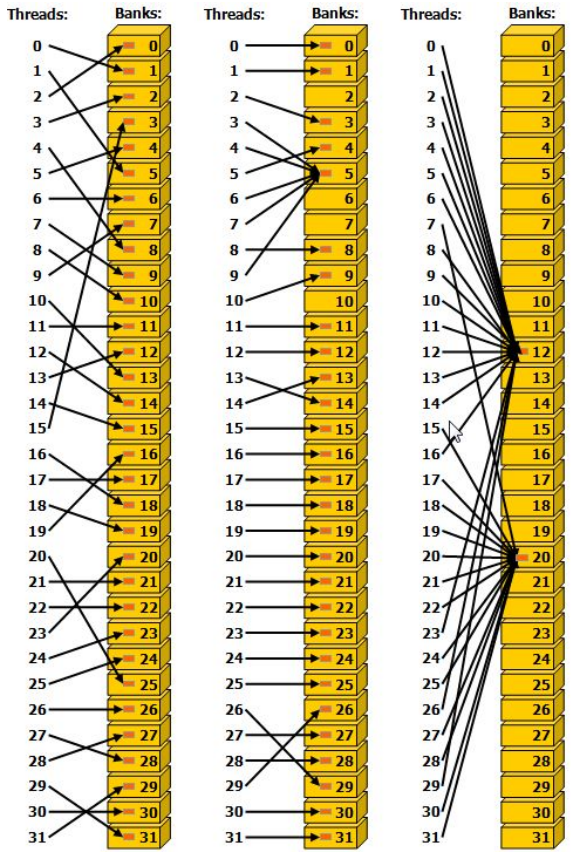
Avoiding shared memory bank conflicts

- (No bank-conflict) Left: Linear addresses with stride 1.
- (Two-way bank conflict) Middle: Linear address with stride 2.
- (No bank-conflict) Right: Linear address with stride 3.



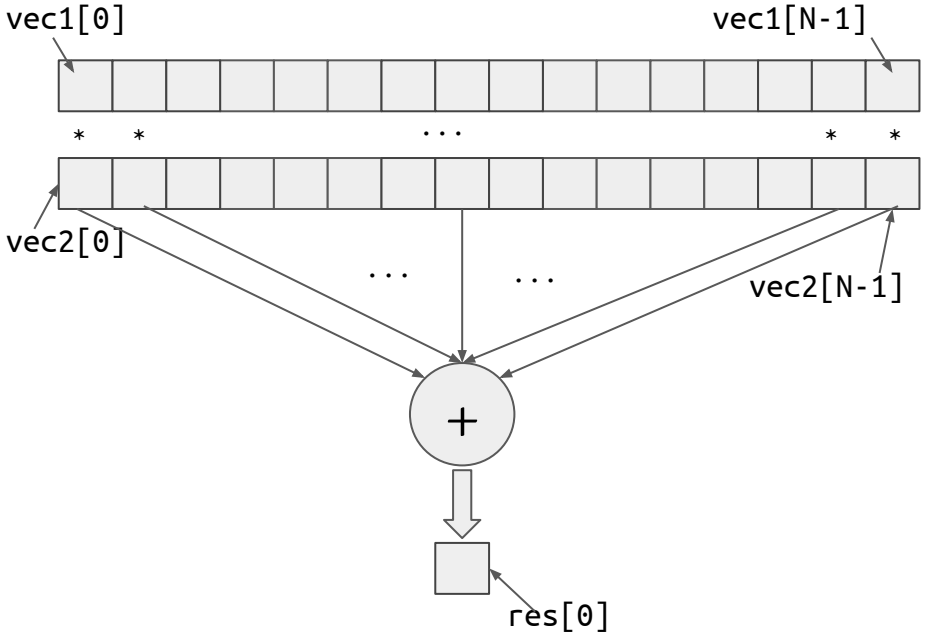
Avoiding shared memory bank conflicts

- (No bank-conflict) Left: Random permutation
- (No bank-conflict) Middle: 3,4,6,7,9 access some word in bank 5.
- (No bank-conflict) Right: Same word accessed → broadcast.



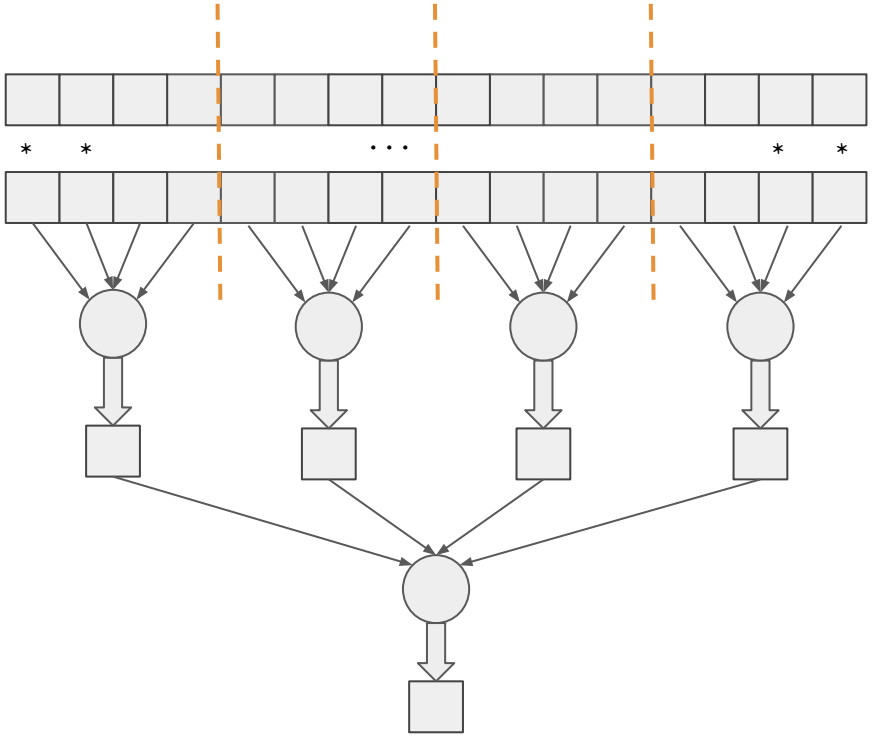
Reduction example: Dot product

- Element-wise multiplication, and then summation of two vectors.
- “Reduce” values in the vectors to one single value.

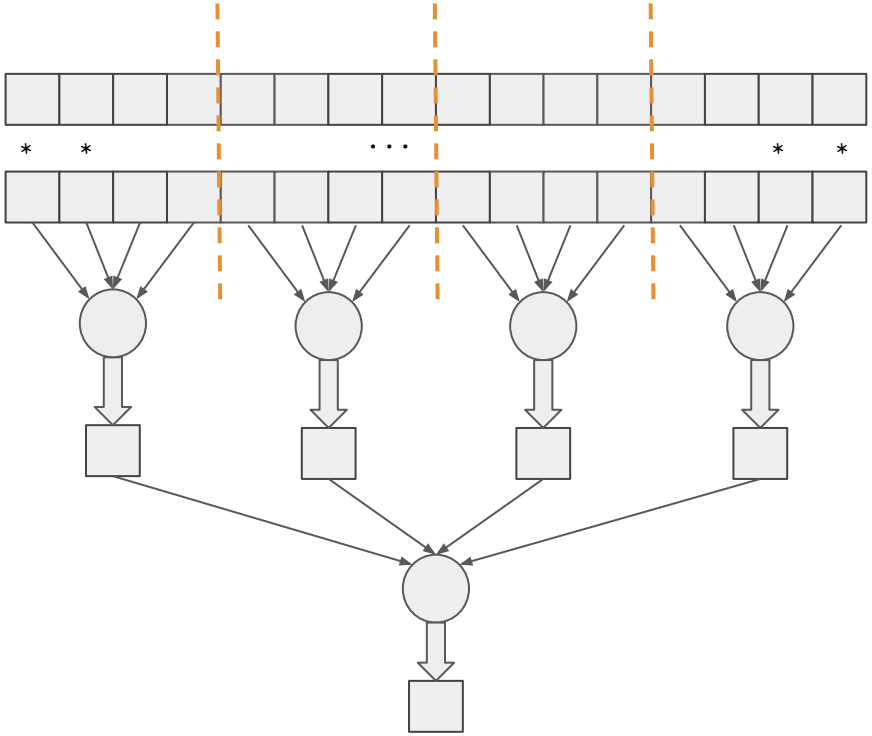


Reduction example: Dot product

- Parallel reduction (multi-stage reduction):
  - Observation: Order of summation is not important (upto rounding errors)
    - a+b+c = b+c+a = c+b+a
  - Distribute vector to multiple blocks and perform recursive summation.
  - Final write to result (global memory) needs to be consistent



Reduction example: Dot product



```
template <typename ValueType>
__global__ void dot(const ValueType* vec1, ValueType* vec2,
ValueType* out)
{
	int tid = threadIdx.x;
	int idx = blockIdx.x * blockDim.x + threadIdx.x;

	extern __shared__ char local_shmem[];
	ValueType* temp = reinterpret_cast<ValueType*>(local_shmem);
	temp[tid] = vec1[idx] * vec2[idx];
	__syncthreads();

	for (int i = blockDim.x / 2; i > 0; i = i / 2) {
		if (tid < i) {
			temp[tid] += temp[tid + i];
		}
		__syncthreads();
	}
	if (tid == 0) {
		atomicAdd(out, temp[tid]);
	}


}
```

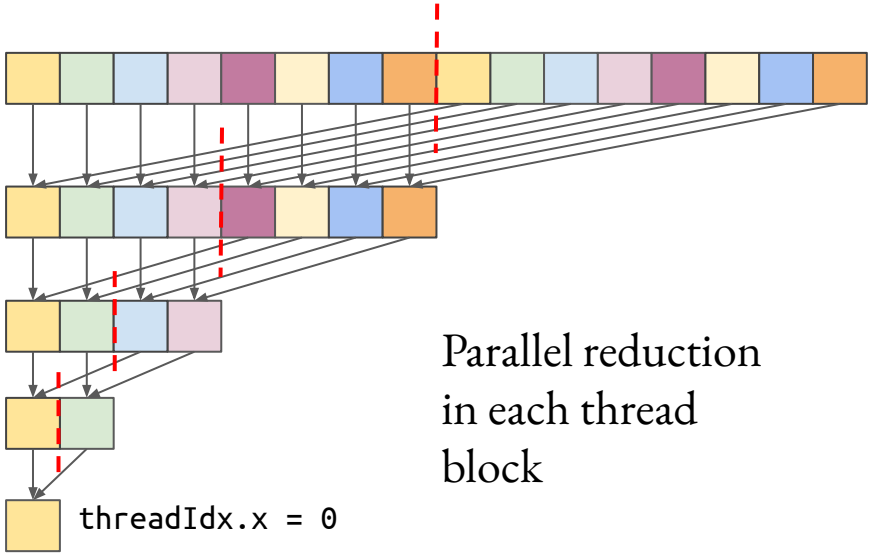
Reduction example: Dot product

- Set shared memory of `block_size` elements .
- Load and element-wise multiply vectors into shared memory.
- Synchronize to prevent RAW hazards.
- Tree-based reduction.
- Atomic add to result in global memory. Necessary to ensure correctness as all thread-blocks write in parallel.

```
template <typename ValueType>
__global__ void dot(const ValueType* vec1, ValueType* vec2,
ValueType* out)
{
        int tid = threadIdx.x;
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        __shared__ ValueType temp[block_size];
        temp[tid] = vec1[idx] * vec2[idx];
        __syncthreads();

        for (int i = blockDim.x / 2; i > 0; i = i / 2) {
                if (tid < i) {
                        temp[tid] += temp[tid + i];
                }
                __syncthreads();
        }

        if (tid == 0) {
                atomicAdd(out, temp[tid]);
        }

}
```

Reduction example: Dot product

- Set shared memory of `block_size` elements .
- Load and element-wise multiply vectors into shared memory.
- Synchronize to prevent RAW hazards.
- Tree-based reduction.
- Atomic add to result in global memory. Necessary to ensure correctness as all thread-blocks write in parallel.

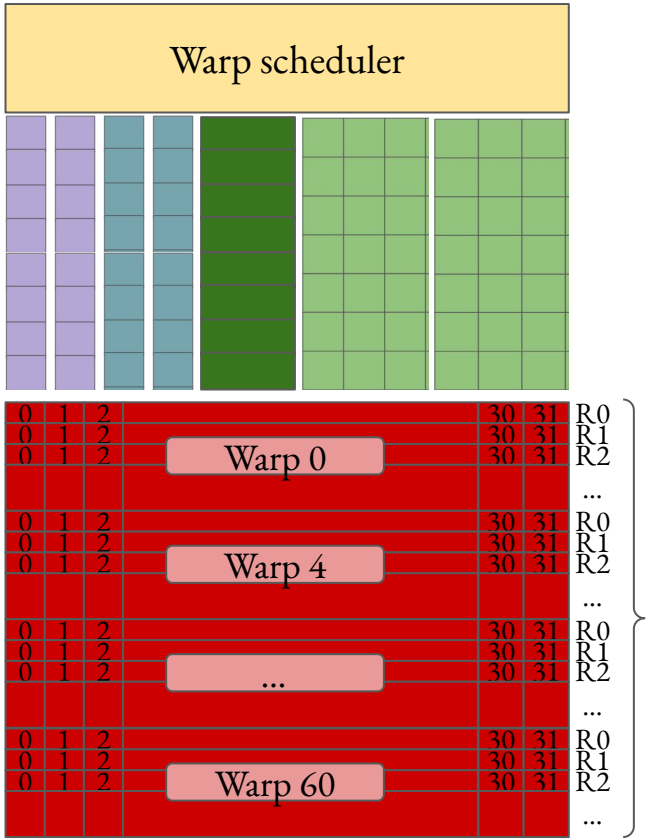


Parallel reduction in each thread block

DEMO

Registers

- Store thread local variables. Fastest access to compute units (Least latency).
- Limited number available in one SM (thread-block) and per thread
  - A100: 65536 32-bit registers per SM
  - CC > 5.0, per thread: 255
- Threads eventually load data from registers to compute.
- Register spilling: When enough registers are not available.



Registers, spilling and control

- Register spilling: When enough registers are not available. Data is then “spilled” into local memory (LMEM), which is cached in the L1 cache → L2 cache → DRAM (successively)
- High register usage limits the number of threads that can run concurrently (Occupancy).
- Maximum number of registers can be controlled with
  - Source: `__launch_bounds__(maxThreadsPerBlock,...)`

```
__global__ void __launch_bounds__() void dot(...)
```

  - Compilation: with `-maxregcount <X>` flag to limit the number of registers per thread to `X` for all kernels within that source file.
- Spilling into LMEM is not always bad! Especially if it can be cached.
- Fewer registers used per thread means that more parallelism is available.

Performance: A balancing game

Quantity	Recommended minimum	Increase in quantity, reduces	Increase in quantity, increases
grid_size = Number of thread blocks	SM count	1. Block size (assuming constant number of threads) 2. Latency (scheduling)	1. Occupancy 2. Available parallelism
block_size = Number of threads in one thread block	Warp size (32)	1. Shared memory available per block	1. Register pressure 2. Functional unit parallelism
Registers per thread	-	1. Warps per SM 2. Block size	1. Effective BW
Shared memory	-	1. Number of thread blocks per SM 2. Occupancy	1. Effective BW

Summary

- Grid and thread block configurations.
- Usage of shared memory: static and dynamic.
- Shared memory banks and avoiding bank conflicts.
- Register usage, register spilling and their effects.

Next lecture

- Device kernels and C++ functionality in CUDA
- Co-operative groups: Harnessing fine-grained level parallelism
- A deeper look into synchronization at various hierarchy levels
- Efficient data exchange at different compute hierarchy levels