Technical University of Munich

Lecture 3: Parallel programming models

Informatics elective: GPU Computing

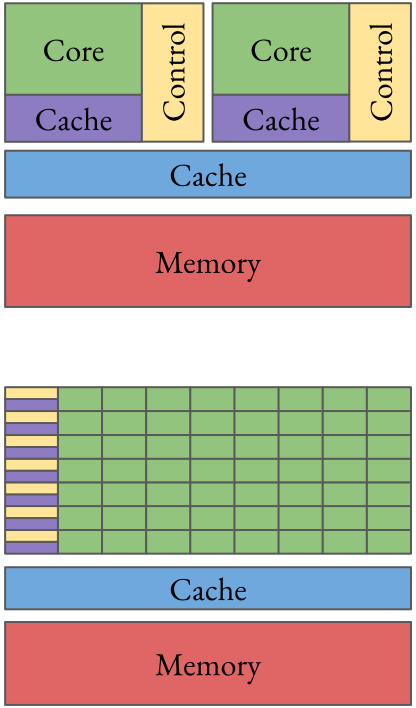
Pratik Nayak

In this session

- Programming models: Types
  - Shared memory models
  - Distributed memory models
  - Distributed-shared memory models
  - Examples: OpenMP, MPI, CUDA, HIP, SYCL, OpenSHMEM
- GPU architecture in detail
- CUDA programming basics: Execution and scheduling
  - Example: Convolution example

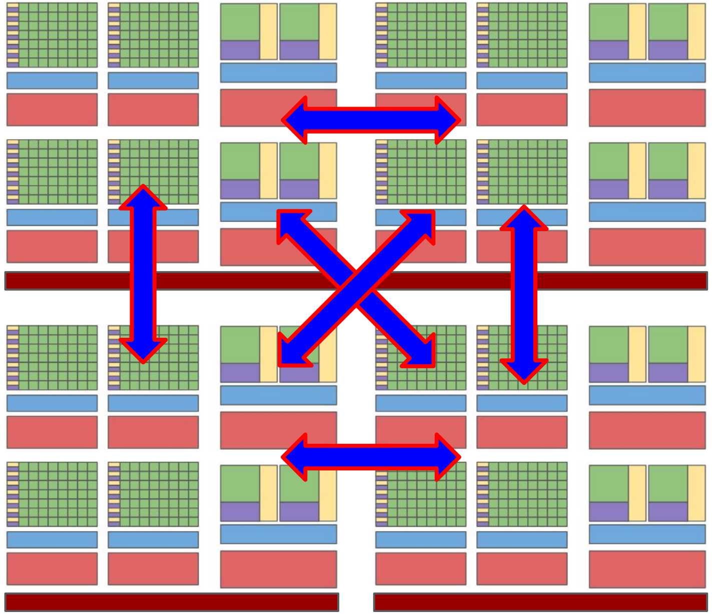
Shared memory architectures

- Shared Memory: Many compute units sharing the same contiguous memory: Modern CPUs and GPUs.
- Limited, but fast main memory accesses.
- Programmable using shared memory programming models.
  - OpenMP, CUDA, SYCL etc.



Distributed memory architectures

- Large scale computing requires collection of shared memory architectures.
- Large but slow (BW and latency) memory accesses
- Programmed through distributed memory programming models
  - MPI, PGAS etc
- Multiple nodes interconnected through a network.



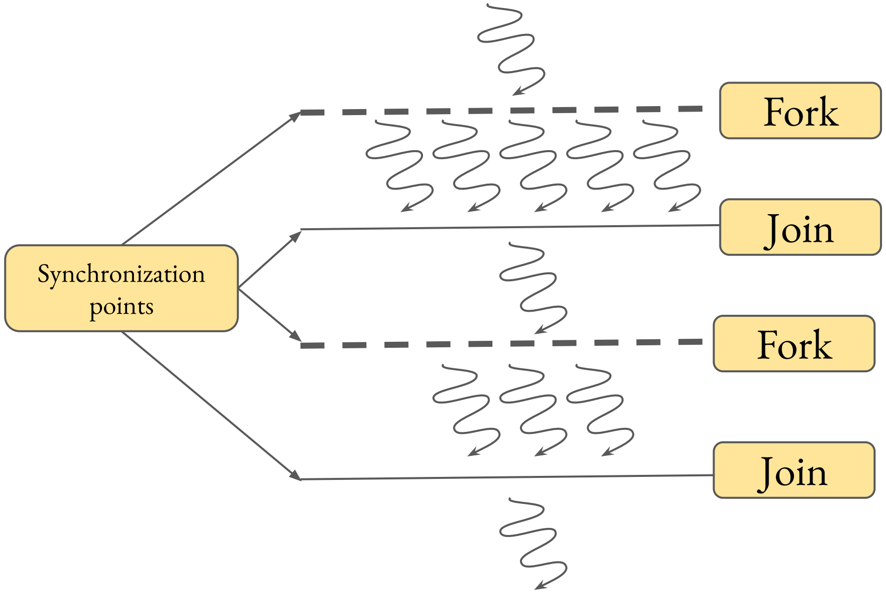
Distributed system: 4 nodes, each with 4 GPUs and 2 CPUs

Parallel programming models

	Parallel programming models		
	Shared memory	Distributed memory	Distributed shared memory
Memory unit	Physically shared memory units	Memory units connected through a network	Virtually shared memory units
Examples	CUDA, SYCL, OpenMP	Message passing interface (MPI)	PGAS
Latency	Low latency	Higher latency	Higher latency
Communication	Implicit	Explicit	Implicit

Bulk-synchronous/fork-join models

- Compute on multiple parallel threads
- Synchronize/join threads when necessary
    - To prevent data races/data hazards
    - Called barriers
- Threads may be mapped to different cores or to same core.
- Example: OpenMP, pthread



Shared memory: OpenMP example and basic syntax

- A low-barrier (pun not intended) way to parallelize existing code.
- Specify parallel regions with `#pragma` directives
- Available for C/C++/Fortran
- Control program level parallelism by setting environment variable:
    - `OMP_NUM_THREADS=x`

```
void add(int length, double* arr1, double* arr2){
        for(int i=0; i<length;++i){
                arr1[i] += arr2[i];
        }
}
```

Shared memory: OpenMP example and basic syntax

- A low-barrier (pun not intended) way to parallelize existing code.
- Specify parallel regions with `#pragma` directives
- Available for C/C++/Fortran
- Control program level parallelism by setting environment variable:
    - `OMP_NUM_THREADS=x`

```
#include “omp.h”

void add(int length, double* arr1, double* arr2){
      #pragma omp parallel for
      for(int i=0; i<length;++i){
            arr1[i] += arr2[i];
      }
}
```

More pragma s

- `#pragma omp parallel for` : Loop parallelization
- `#pragma omp parallel {...}` : Region parallelization
- Data sharing clauses: `#pragma omp parallel <clause>`
  - `private :` All threads access the exact same variable
  - `shared :` All threads access copies of the variable (uninitialized)
  - `firstprivate :` All threads access copies of the variable (initialized before clause)
  - `lastprivate :` All threads access copies of the variable (but after last iteration of loop, variable is set to the value in the last iteration)
- `#pragma omp barrier :` All threads wait until other threads reach the barrier

Parallelizing reductions

```
#include <omp.h>

int main() {
.
.
double res = 0.0;

#pragma omp parallel for
for(int i=0; i<vec.size(); ++i) {
     res += vec[i];
}
}
```

Sum elements of an array

- Does this do what you would expect ?

Parallelizing reductions

```
#include <omp.h>

int main() {
.
.
double res = 0.0;

#pragma omp parallel for
for(int i=0; i<vec.size(); ++i) {
     res += vec[i];
}
}
```

- Shared variable
- No ordering specified on access
- A data race

DEMO

Parallelizing reductions

```
#include <omp.h>

int main() {
.
.
double res = 0.0;

#pragma omp parallel for reduction(+:res)
for(int i=0; i<vec.size(); ++i) {
     res += vec[i];
}
}
```

- Use a reduction clause, specify shared variables
  - `reduction(<op>:<variable>)`
- `<op>: +, - or *, max, min, |, &, ^` etc
- Declare your own custom operator
  - `declare reduction(...)`

Distributed memory : Message Passing Interface (MPI)

- Works on a SPMD model.
- Create N instances (ranks) of the program.
- Explicitly exchange data between ranks as needed with MPI functions.
    - Data is distributed and each rank calls functions on its local data
- Point-to-point communication or collective communication.

```
#include “mpi.h”

void vec_add(vector<int>& vec1, vector<int>& vec2){
for(int i=0; i<vec1.size();++i){
          vec1[i] += vec2[i];
     }
}

int main(int argc, char**argv){
MPI_Init(&argc, &argv);

int len=10;
vector<int> vec1(len/num_ranks, 2.5);
vector<int> vec2(len/num_ranks, 3.5);

vec_add(vec1, vec2);

MPI_Finalize();
}
```

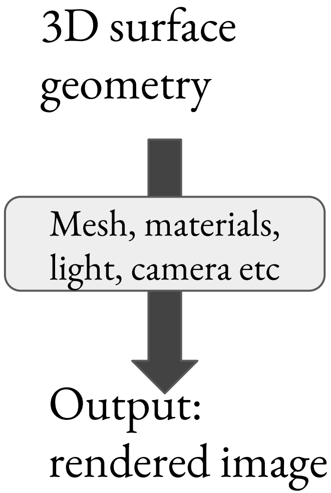
Parallel region

Distributed shared memory : PGAS: UPC

- View the distributed memory units as a single address space.
- Communication hidden to user and done implicitly.
- Easier to program when viewing arrays as global.
- Can be inefficient (resulting in more communications) if proper care is not taken.

```
#include “upc.h”

#define len 10;
shared int vec1[len];
shared int vec2[len];
shared int res[len];

int main(){

int i;
upc_forall (i=0; i<len; i++; &res[len])
res[i]=vec1[i]+vec2[i];

}
```

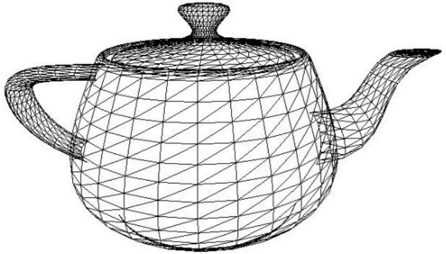
GPUs ? What are they good for ?



3D surface geometry

Mesh, materials, light, camera etc

Output: rendered image

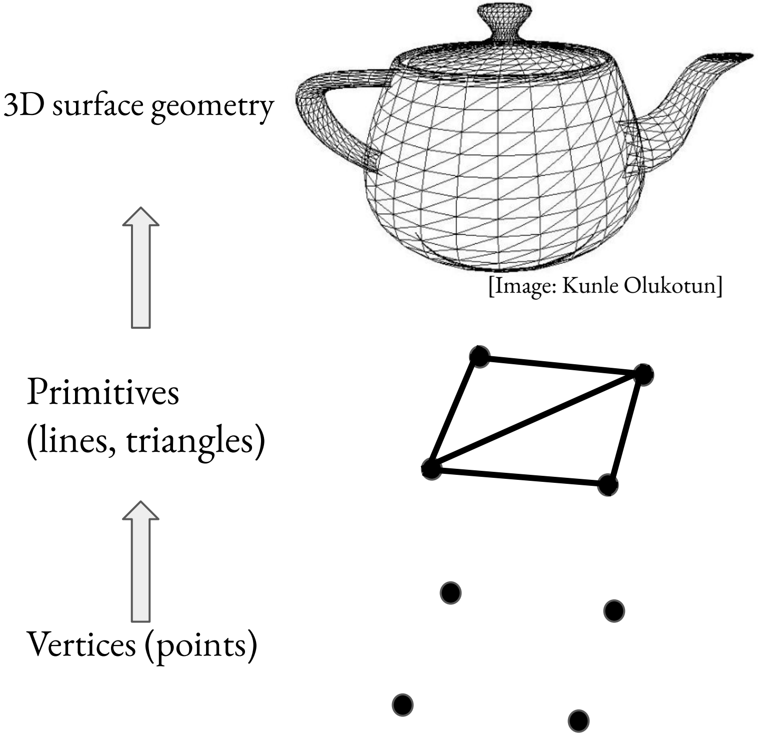


[Image: Kunle Olukotun]

[Image: Henrik Wann Jensen]

Graphics pipelines: workload

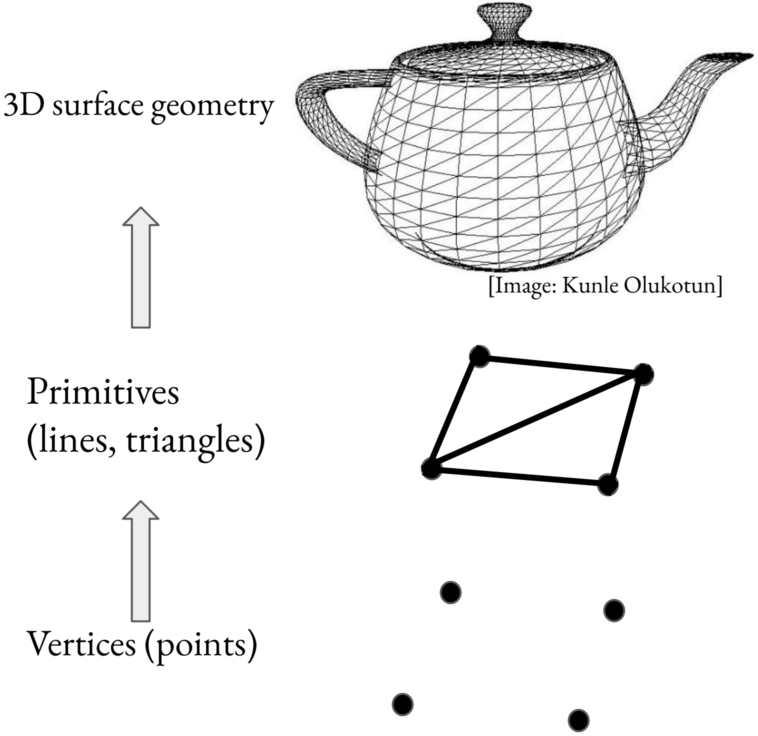
- Given a mesh, and a primitive (for example a triangle in the mesh), a virtual camera, determine its position on the screen.
- For all the pixels covered by the triangle, compute the color of the surface at that pixel.
- Rendering objects: Computing each primitive’s pixel contribution to an image



[Image: Kunle Olukotun]

Graphics pipelines: workload

- Shader programs did exactly that:
  - For each fragment (pixel covered by the triangle), there is a need to calculate a quantity.
- GPUs therefore were used to compute these for very large collections of data (vertices, pixels etc)



3D surface geometry

[Image: Kunle Olukotun]

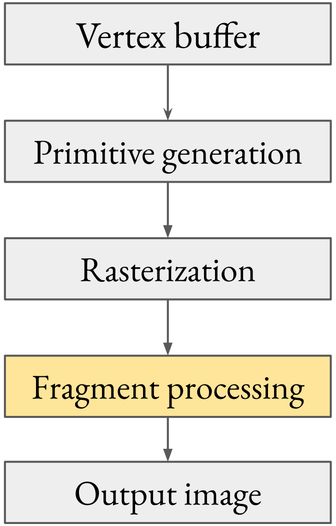
Primitives
(lines, triangles)

Vertices (points)

Graphics pipelines on a GPU

- To draw an image:
    - Using graphics driver API, write shader programs
    - Provide input vertex buffers along with parameters
    - "Draw" the image on the input vertices

This was mostly the case for early GPUs. Only graphics pipelines were allowed.



Generic, non-graphics computations on a GPU

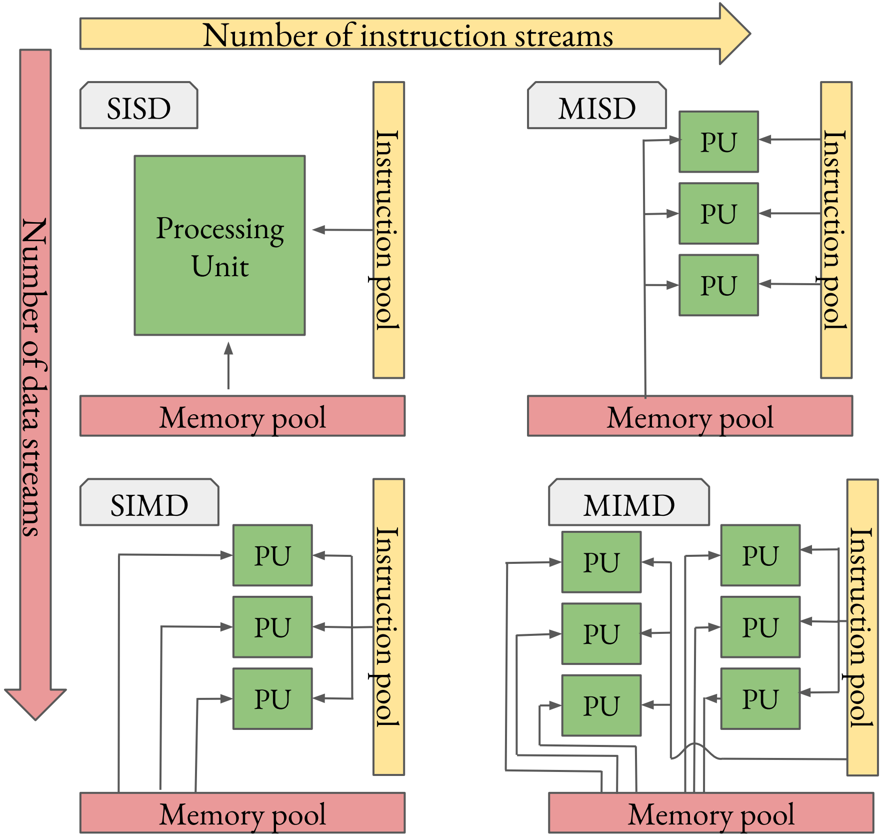
- People figured out that this can be extended for general computations as well:
  - Allocate buffers in GPU memory
  - Provide some kernel (using driver API) that you want to run
  - Run N instances of this kernel in a SPMD (**S**ingle **P**rogram **M**ultiple **D**ata) fashion.

```
run_kernel(myKernel, N)
```

- Surprisingly simpler than graphics computing.
  - Different programming model, but eventually needs to map to the same hardware.

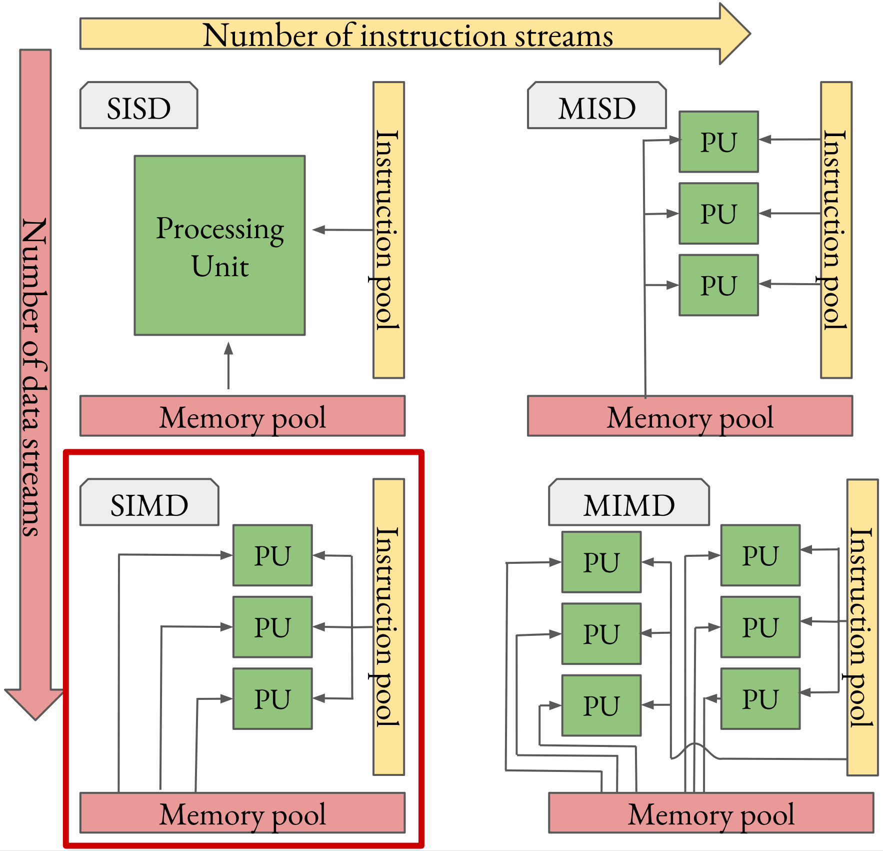
Recall Flynn’s taxonomy

- SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata stream
- MISD: **M**ultiple **I**nstruction streams, **S**ingle **D**ata stream
- SIMD: **S**ingle **I**nstruction stream, **M**ultiple **D**ata streams.
- MIMD: **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams



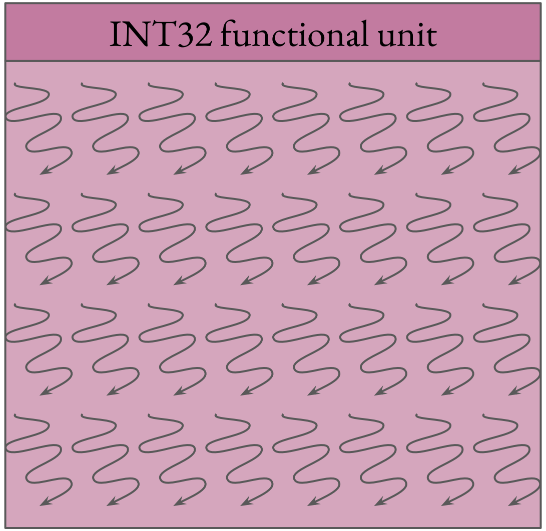
Recall Flynn's taxonomy

- SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata stream
- MISD: **M**ultiple **I**nstruction streams, **S**ingle **D**ata stream
- SIMD: **S**ingle **I**nstruction stream, **M**ultiple **D**ata streams.
- MIMD: **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams



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



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

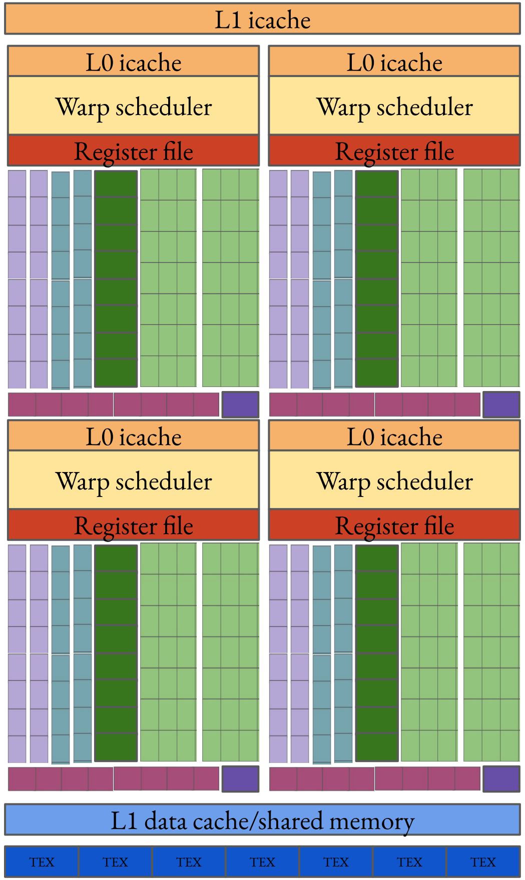
FP32 functional unit → 16 FMA per clock-cycle

FP64 functional unit → 8 FMA per clock-cycle

Tensor core functional unit

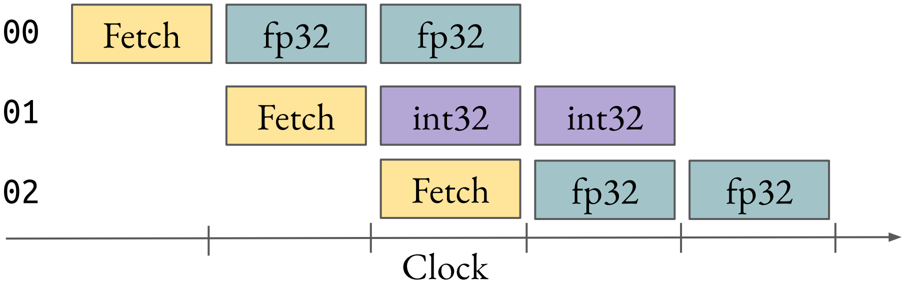
Load/store units

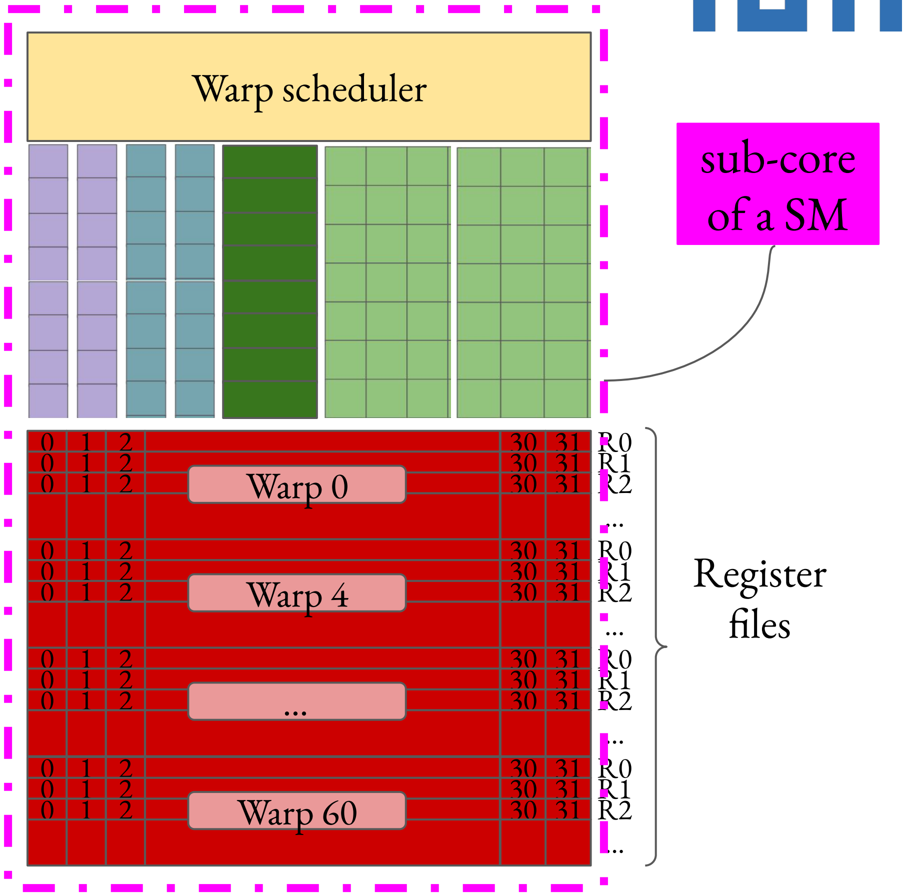
Special function unit



Warp scheduling

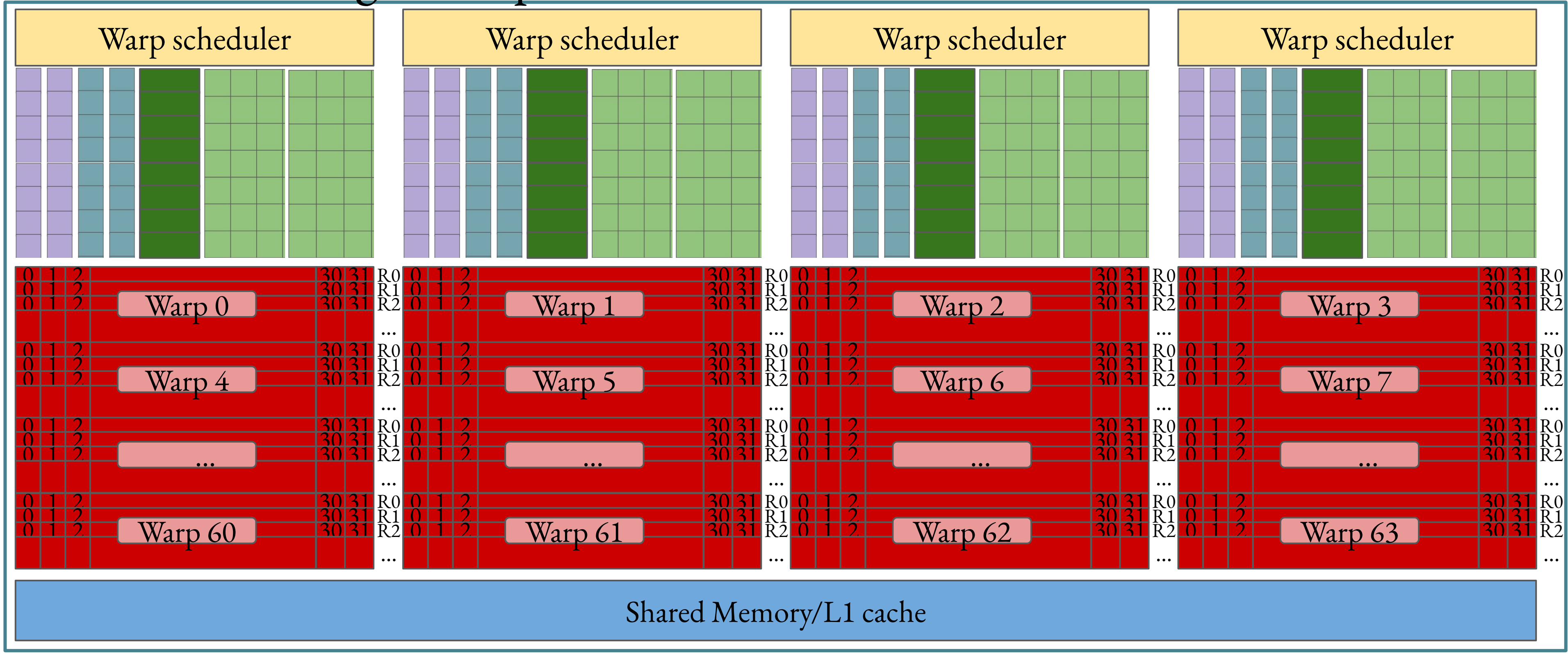
```
00  fp32   mul r0 r1 r2
01  int32  add r3 r4 r5
02  fp32   add r6 r7 r8
```





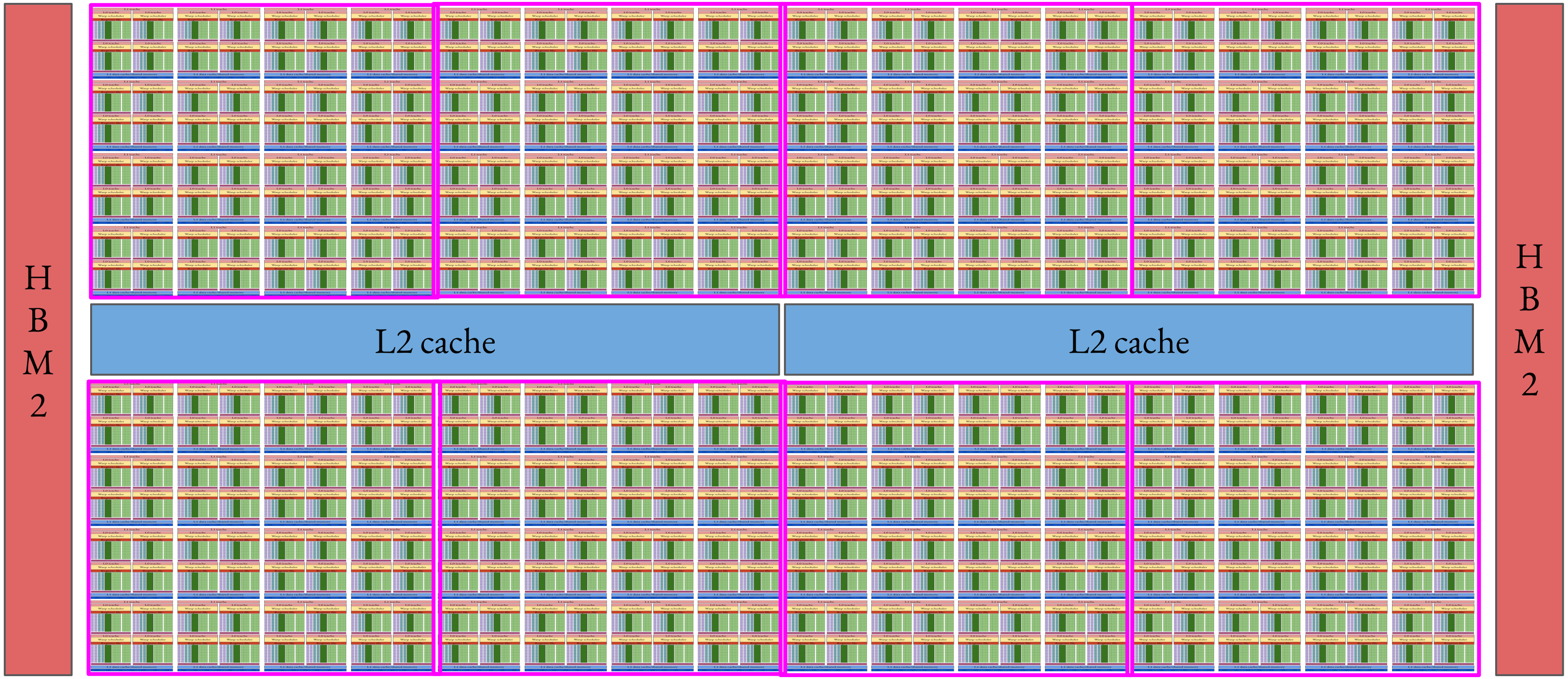
- Each instruction is run by the whole warp
- All 32 threads of the warp run the same instruction
- Thread divergence → performance hit

One Streaming Multiprocessor



GPU: Array of Streaming Multiprocessors

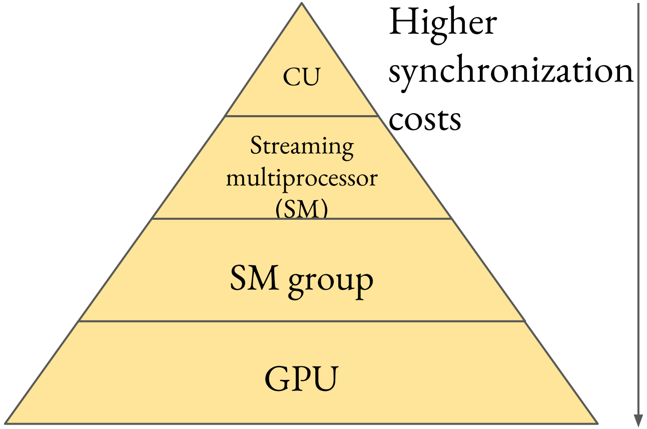
128 SMs of A100 GPU

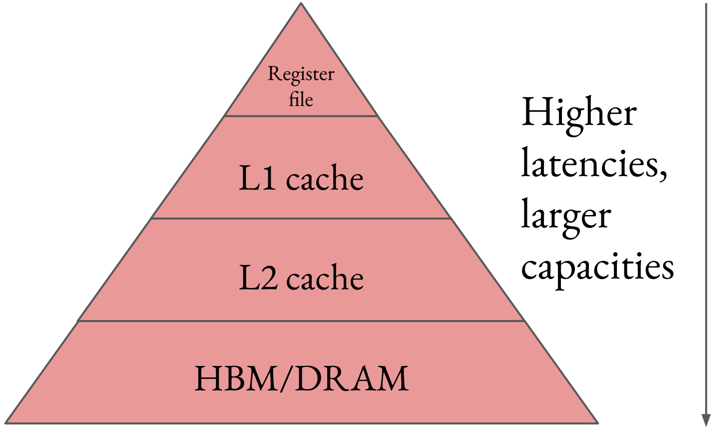


GPUs: High throughput shared memory architectures

- 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.

GPUs: Highly hierarchical memory and compute





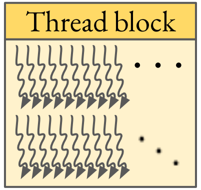
- Highly hierarchical memory and compute.
- Larger capacities/more parallelism → higher latencies/higher synchronization costs
- Reframe algorithms to reduce serial components → Maximize available parallelism

Computing on a GPU: CUDA

- CUDA is a “C-like” language enabling massively parallel computations on GPUs.
- Three key abstractions:
  - Hierarchy of thread groups
  - Shared memory
  - Barrier synchronization
- Program using data-parallelism and thread parallelism.
- Fairly well-documented: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Computing on a GPU: CUDA

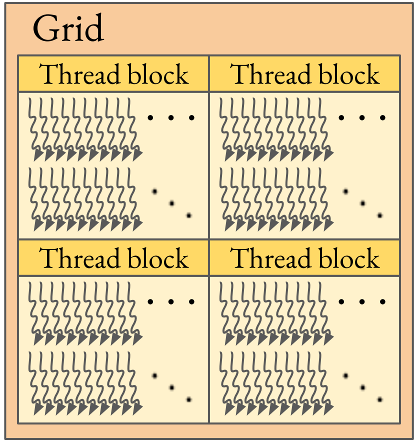
- Threads organized into “thread-blocks”.



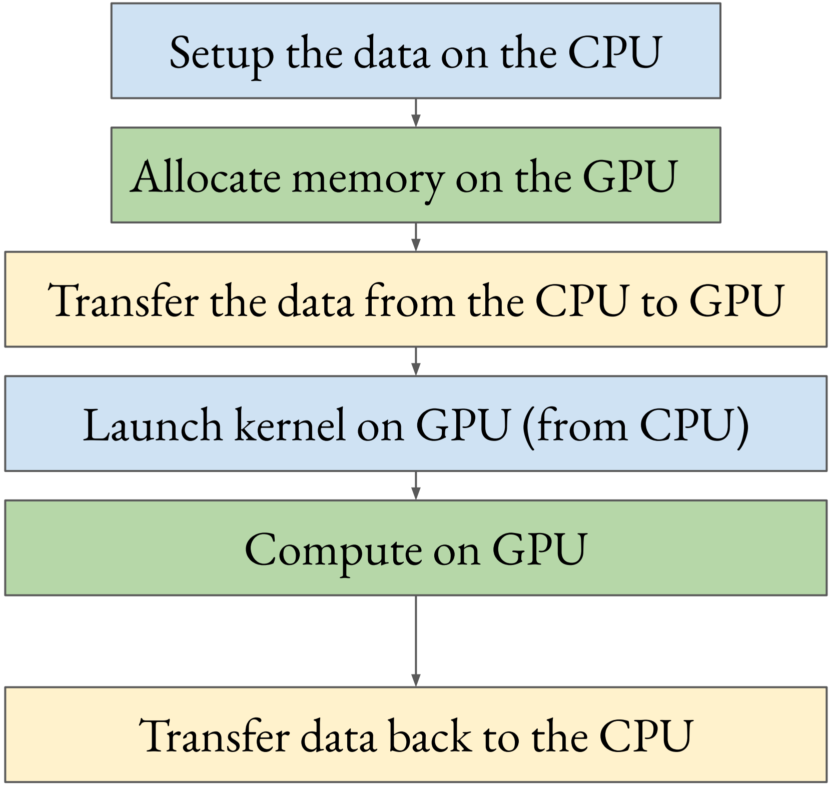
Computing on a GPU: CUDA

- Threads organized into “thread-blocks”.
- A kernel consists of launching a “grid” of these “thread-blocks”.
- Launching a kernel is done with the syntax:

  `kernel<<<num_blocks, block_size>>>(...)`
- `block_size` : number of threads in a block.
- `num_blocks` : The number of thread-blocks to launch.



Computing on a GPU: A general workflow



Setup the data on the CPU

Allocate memory on the GPU

Transfer the data from the CPU to GPU

Launch kernel on GPU (from CPU)

Compute on GPU

Transfer data back to the CPU

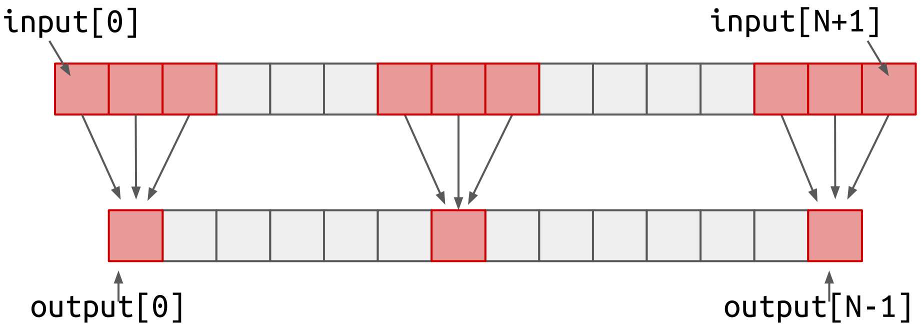
```
malloc(...);initialize_data(...)

cudaMalloc(...)

cudaMemcpy(..., HostToDevice)

kernel<<<nblocks, bsize>>>(...)
```

Control transferred to the GPU

```
cudaMemcpy(..., DeviceToHost)
```

Computing on a GPU

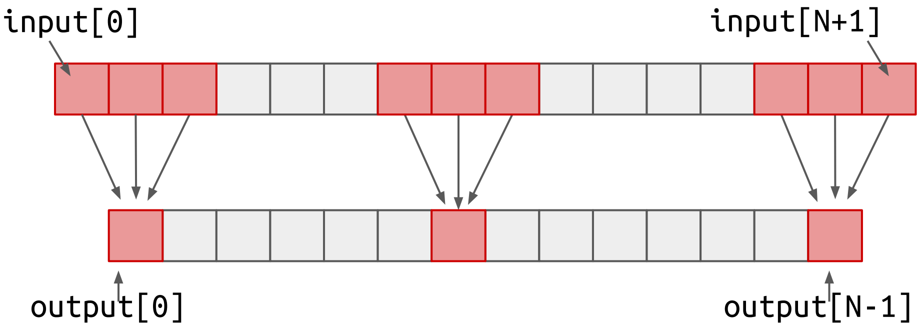
- Consider a 1D convolution operation
- Convolve input array and sum the values in an output array
- What are the data dependencies ?
- What parallelism is available ?



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

Computing on a GPU

- Consider a 1D convolution operation
- Convolve input array and sum the values in an output array
- What are the data dependencies ?
  - None: Each element can be computed in parallel
- What parallelism is available ?
  - All `N` elements(of `output`) can be computed in parallel



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

Can have `N` threads in parallel

Computing on a GPU: A general workflow

DEMO

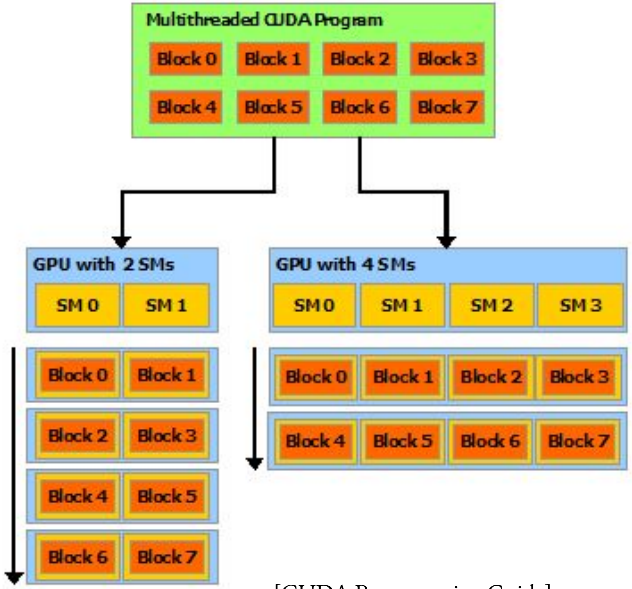
```
malloc(...);initialize_data(...)

cudaMalloc(...)

cudaMemcpy(..., HostToDevice)

convolution<<<nblocks, bsize>>>(...)
```

Control transferred to the GPU

```
cudaMemcpy(..., DeviceToHost)
```

Scheduling on a GPU

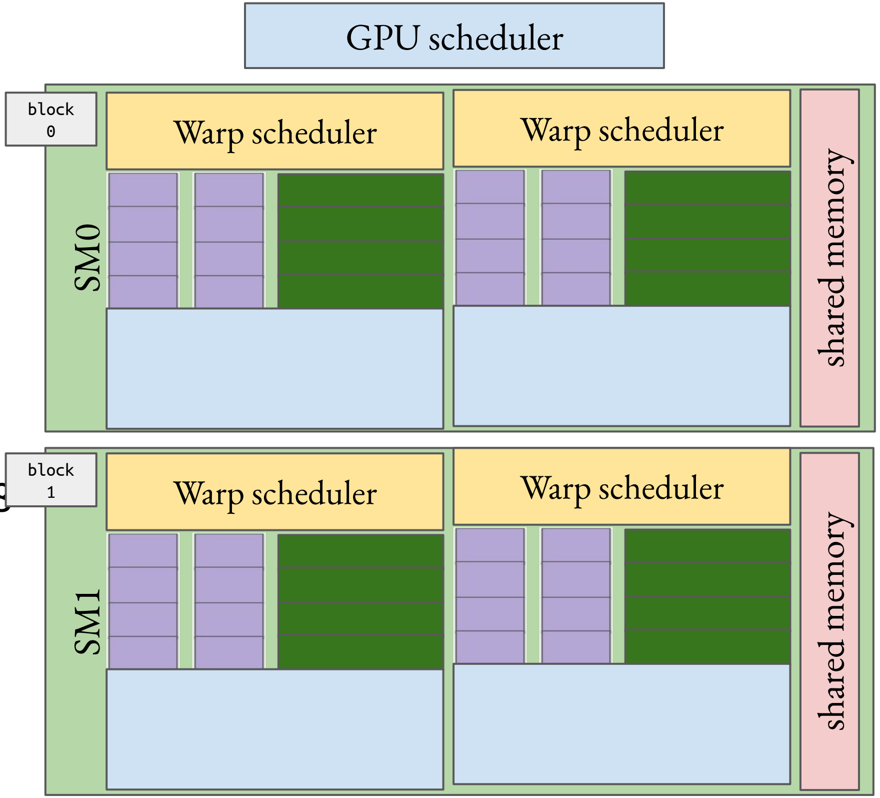
- Understanding how thread-blocks are mapped to SMs crucial to maximize performance.
- CUDA handles the scheduling transparently to the user.
- Scales automatically to different GPU generations.



[CUDA Programming Guide]

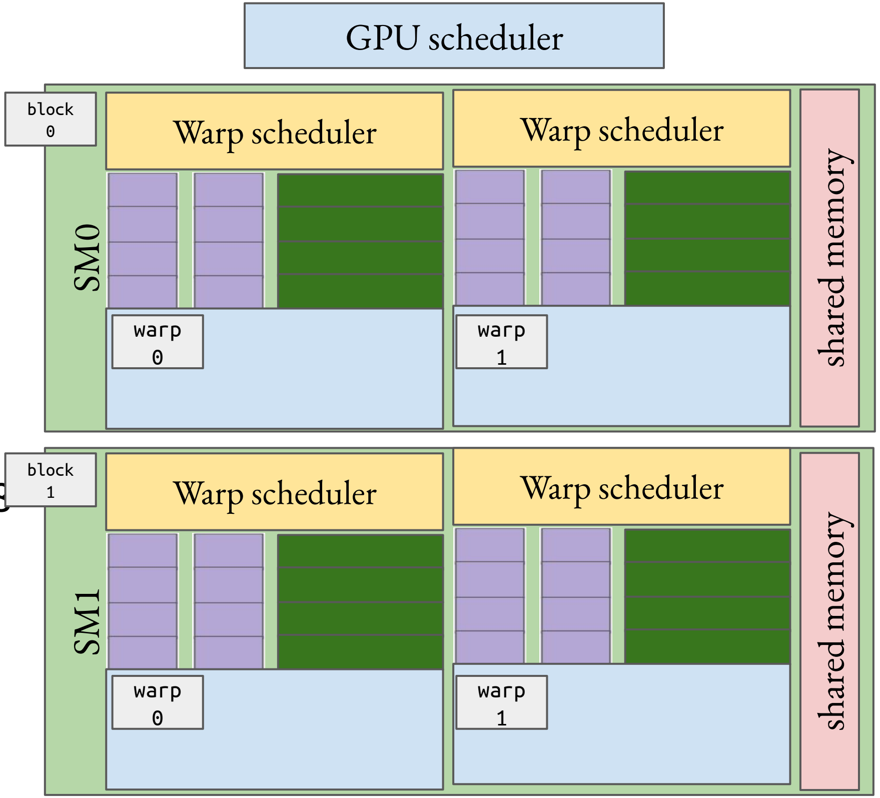
Scheduling on a GPU

- Consider an imaginary CUDA GPU with 2 SMs, each with 2 “sub-cores” and shared memory/L1 cache.
- Consider our convolution kernel.
  - Launched with `(N/block_size = 8)` thread blocks, with `N = 1024, block_size = 128`
  - 1 thread block always mapped to 1 SM
    - Warps are then mapped to “sub-cores”
    - 128 block size → 4 warps (32 x 4)



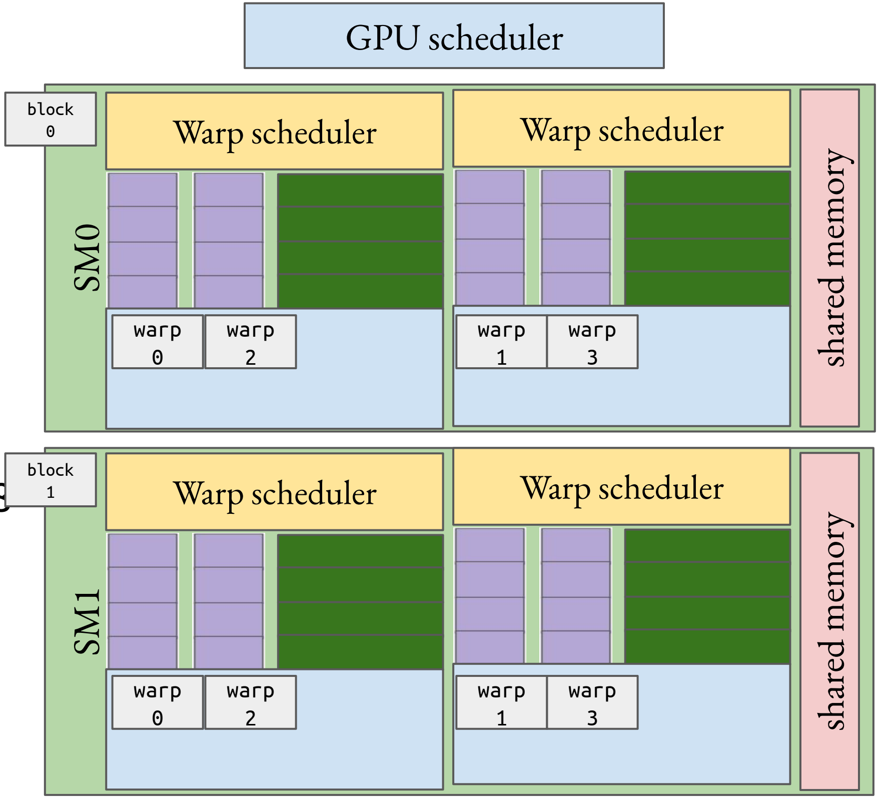
Scheduling on a GPU

- Consider an imaginary CUDA GPU with 2 SMs, each with 2 “sub-cores” and shared memory/L1 cache.
- Consider our convolution kernel.
  - Launched with (`N/block_size = 8`) thread blocks, with `N = 1024, block_size = 128`
  - 1 thread block always mapped to 1 SM
    - Warps are then mapped to “sub-cores”
    - 128 block size → 4 warps (32 x 4)



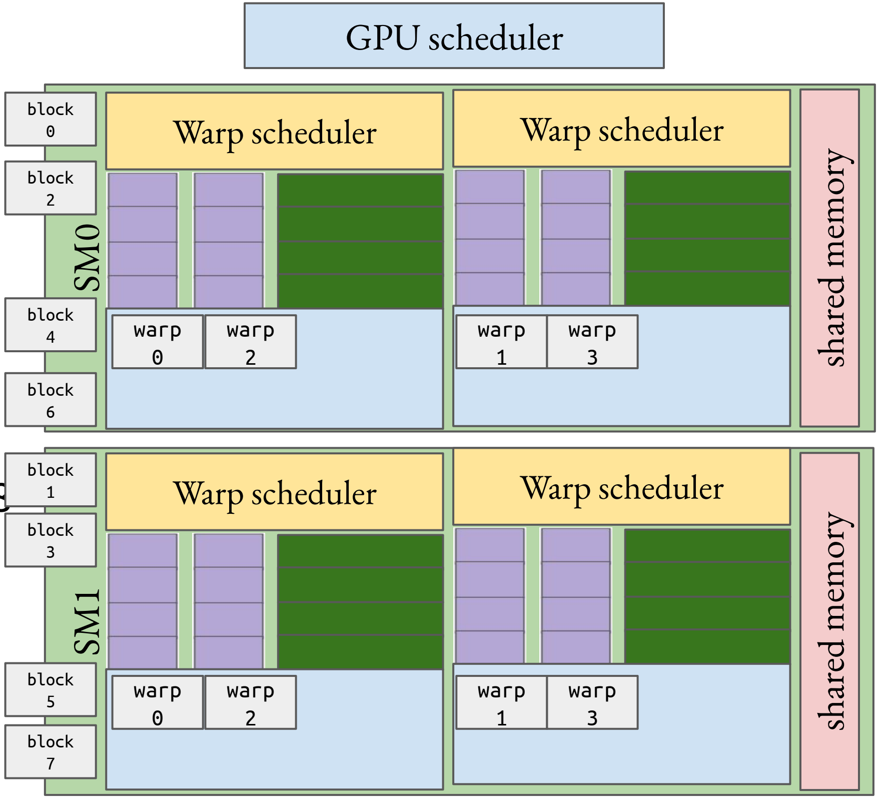
Scheduling on a GPU

- Consider an imaginary CUDA GPU with 2 SMs, each with 2 "sub-cores" and shared memory/L1 cache.
- Consider our convolution kernel.
  - Launched with (`N/block_size = 8`) thread blocks, with `N = 1024, block_size = 128`
  - 1 thread block always mapped to 1 SM
    - Warps are then mapped to "sub-cores"
    - 128 block size → 4 warps (32 x 4)



Scheduling on a GPU

- Consider an imaginary CUDA GPU with 2 SMs, each with 2 "sub-cores" and shared memory/L1 cache.
- Consider our convolution kernel.
  - Launched with (`N/block_size = 8`) thread blocks, with `N = 1024, block_size = 128`
  - 1 thread block always mapped to 1 SM
    - Warps are then mapped to "sub-cores"
    - 128 block size → 4 warps (32 x 4)



Summary

- Different types of parallel programming models: Shared, distributed and distributed shared
- CUDA: The programming model for NVIDIA GPUs.
- Understanding GPU architecture is crucial.
- Maximizing GPU performance requires attention to memory hierarchy and the execution hierarchy

Next week

- A more detailed look at CUDA programming syntax and semantics
- CUDA Memory model: Global memory, shared memory, texture memory
- Device kernels and device programming: Better code architecture with CUDA C++