HIGH PERFORMANCE PARALLEL PROGRAMMING (CS61064)

Soumyajit Dey
CSE, IIT Kharagpur

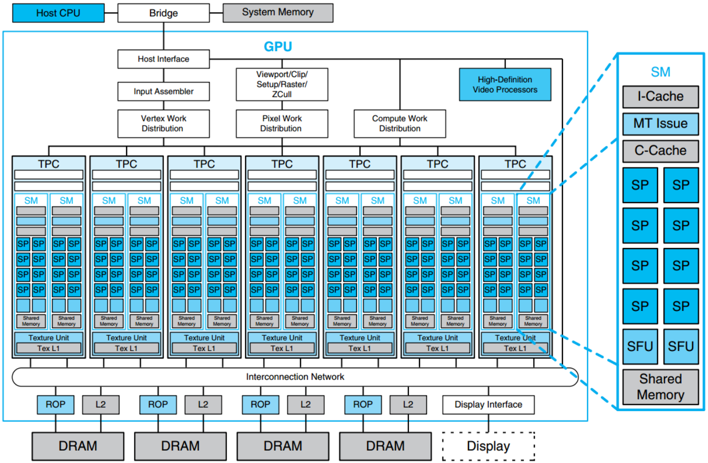


Figure: GPU Architecture - Hennessy Patterson

GPU can be viewed as an array of Streaming Multiprocessors (SMs) Each SM has the following elements

- Registers that can be partitioned among threads of execution
- Several Caches: Shared memory, Constant, Texture, L1 etc
- Warp Schedulers (More on this later)
- Scalar Processors(SPs) for integer and floating-point operations
- Special Function Units (SFUs) for single-precision floating-point transcendental functions

Memory Model Overview

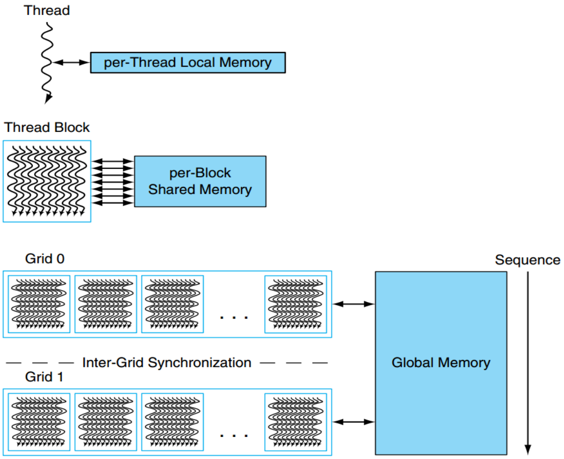


Figure: Hennessy Patterson GPU Memory Model

Table: CUDA Device Memory Types and Scopes

Variables Declaration	Memory	Scope	Lifetime
Automatic Variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar	Shared	Block	Kernel
__device__ int GlobalVar	Global	Grid	Application
__device__ __constant__ int ConstVar	Constant	Grid	Application

Recap

$$\text{NumCols} = \text{blockDim.x} * \text{gridDim.x}$$
$$\text{NumRows} = \text{blockDim.y} * \text{gridDim.y}$$

$gridDim = \langle 3, 2 \rangle$ $blockDim = \langle 5, 4 \rangle$

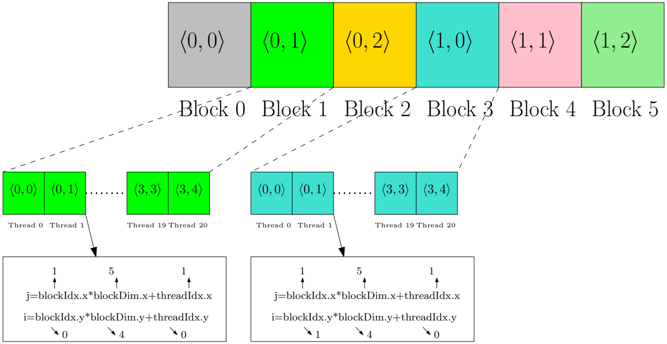


Figure: Kernel Grid

Mapping to Hardware

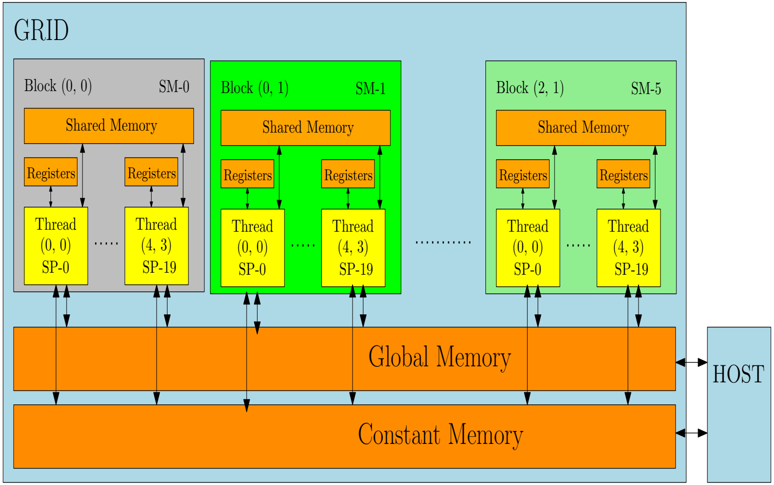


Figure: Mapping Kernel Grids to Architecture

Mapping to Hardware

- Consider a scenario where the number of blocks is more than the number of SMs in the hardware
- *gridDim* $=< \mathbf{6}, 2 >$ and *blockDim* $=< 5, 4 >$
- Thread Blocks are launched in batches sequentially.
- Execution is serialized to some extent.

Mapping to Hardware

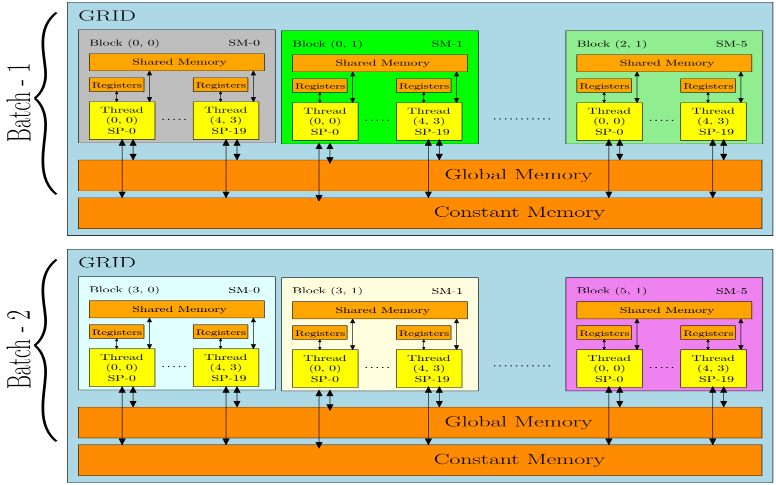


Figure: Mapping Kernel Grids

GPU HW scheduler

- If we really use the simplistic arrangement as discussed, we cannot launch more threads than number of physical hardware
- Problem: if a thread is stalled due to long latency access, the SP is under-utilized.
- So, threads should share SPs
- To use the full possible power of a GPU you need much more threads per SM than the SM has SPs.

Threads getting mapped to physical SPs is managed by the scheduler

SM, SP, Block n thread

- thread block max size : 1024 (modern archs 2048)
- SM can store max 1024 "thread contexts"
- can have much less than 1024 SPs
- GTX 970 : 13 SMs : 13 X 1024 thread contexts in parallel
- GTX 970 : 128 SP per SM

SM, SP, Block n thread

- One block in one SM
- One SM can have multiple blocks

If SM can store max 1024 "thread contexts", and block size is 256, we have 4 blocks per SM.

GPU HW scheduler

- The hw scheduler decided which threads to map to a collection of SPs in SIMD fashion :: SIMT model of execution
- This collection is physically guaranteed to execute in parallel
- The unit of such collections is ”warp”

SM: A closer look

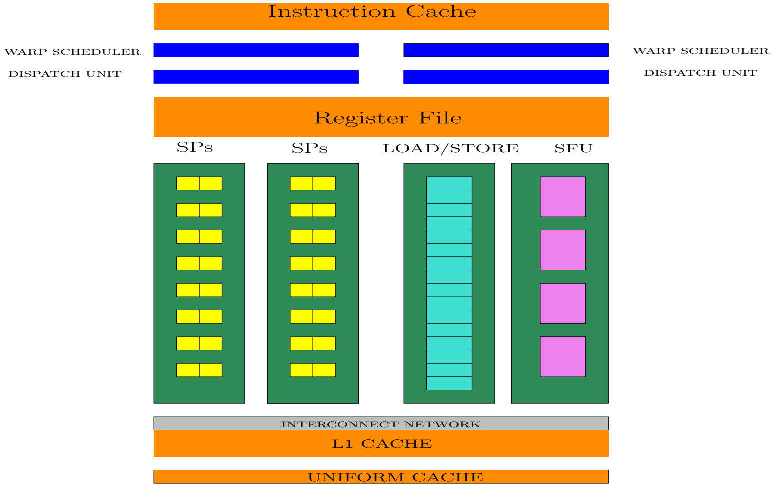


Figure: Streaming Multiprocessor

Warps

- Warp is a unit of thread Scheduling in SMs.
- Warp size is implementation specific (typically 32 threads)
- Warps are executed in an SIMD fashion i.e. the warp scheduler launches warps of threads and each warp typically executes one instruction across parallel threads.

Ex : If a SM has 128 SPs, it can execute 4 Warps at a given time (one Warp has 32 Threads)

Warp Scheduling

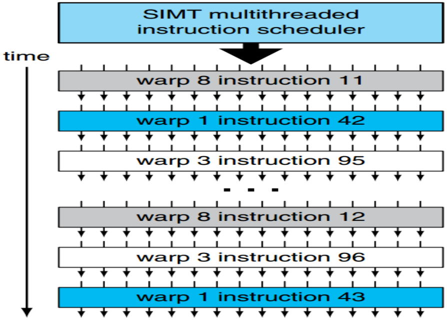


Figure: Scheduling warps on one SM - Hennessy Patterson

Latency Tolerance

- When threads in one warp execute a long-latency operation (read from global memory), the warp scheduler will dispatch and execute other warps until that operation is finished.
- A common practice is to launch thread blocks of a size that is a multiple of the warp size to maximally utilize threads.
- Slow global memory accesses by threads in a warp may be optimized using coalescing (more on this later)

Warp Scheduling

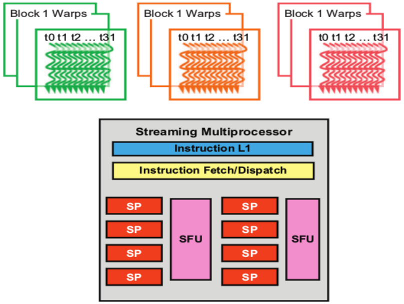


Figure: Scheduling warps on GPU - Kirk Hwu

Efficient use of thread blocks

Target System Constraints

- A maximum of 8 blocks and 1024 threads per SM
- A maximum of 512 threads per block

Table: Solutions for various block scenarios

Input Block Size	**Blocks per SM**	**Threads per Block**	**Remarks**
8 * 8	12	64	SM execution resources will be underutilized
16*16	4	256	Achieves full thread capacity in SMs
32*32	1	1024	Exceeds the limit of 512 threads per block

Querying Device Properties

CUDA API provides constructs for obtaining properties of the target GPU.

- **cudaGetDeviceCount():** Obtains the number of devices in the system.
- **cudaGetDeviceProperties():** Returns the property values of a particular device

Querying Device Properties

```
void printDevProp(cudaDeviceProp devp)
{
 printf("No. of multiprocessors: %d\n"
    , devp.multiProcessorCount); //24
 printf("Size of warp %d\n", devp.
    warpSize ); //32
 return;
}
```

Querying Device Properties

```
int main()
{

    int devCount;
    cudaGetDeviceCount(&devCount);
    for (int i = 0; i < devCount; ++i)
    {
        cudaDeviceProp devp;
        cudaGetDeviceProperties(&devp,
            i);
        printDevProp(devp);
    }
    return 0;
}
```

Complete program

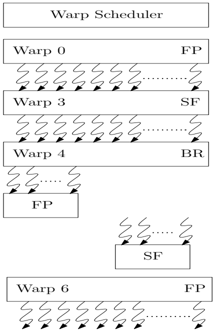
Target Device Characteristics

- Maximum threads per block: 1024
- Maximum dimension 0 of block: 1024
- Maximum dimension 1 of block: 1024
- Maximum dimension 2 of block: 64
- Number of multiprocessors: 24
- Size of warp 32

Control Flow Divergence

- Threads inside a warp execute the same instruction.
- How does a warp handle if statements / branch instructions?
- The GPU is not capable of running both the if else blocks at the same time.

Warp Scheduling



Threads inside a warp executing same instruction - efficient

Threads inside a warp executing different instruction - inefficient

Figure: Warp Divergence

Divergent Code 1

Consider the following kernel code

```
__global__
void divergence(float *M)
{
        int j=blockIdx.x*blockDim.x+
           threadIdx.x;
        if(j%2)
              M[j]+=2;
        else
              M[j]-=2;

}
```

Half the threads of a warp execute the addition instruction while the other half execute the subtraction instruction.

The Hardware's Job

The GPU has two methodologies for handling divergent branch instructions in code.

- Replace branch instructions with predicated instructions.
- Maintain internal masks and branch synchronization stack

To understand the above concepts, lets have a look at the PTX Assembly code.

```
nvcc --keep divergence.cu
cuobjdump a.out -ptx
```

Predicated Instructions

- The setp instruction evaluates the conditional expression of the `if` statement.
- PTX assembler generates only predicated instructions and no branch instruction
- The SIMD lanes with the predicate set to 1 perform the operation and store the result for one conditional block statement while the other performs no operation.

Warp Scheduling with Predicated Instructions

```
// r5 contains thread id
// p1 and p2 are predicate registers
and.b32 %r5, %r4, 1;
setp.eq.b32 %p1, %r5, 1;
not.pred %p2, %p1;
mul.wide.s32 %rd3, %r4, 4;
add.s64 %rd4, %rd2, %rd3;
ld.global.f32 %f1, [%rd4];
selp.f32    %f2, 0fC0000000, 0
   f40000000, %p2;
add.f32 %f3, %f1, %f2;
st.global.f32 [%rd4], %f3;
```

Warp Scheduling

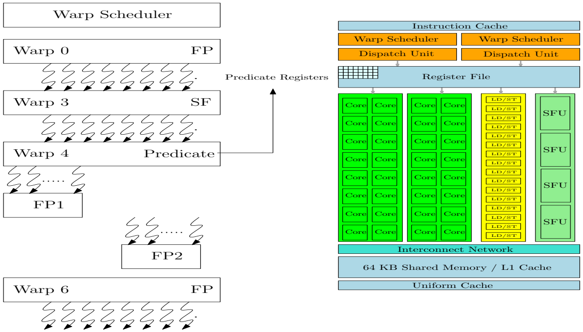


Figure: Solution with predicated instructions

Divergent Code 2

The conditional blocks now have more instructions (load and store)

```
__global__
void divergence(float *M, float *N,
   float *P)
{
        int j=blockIdx.x*blockDim.x+
           threadIdx.x;
        if(j>0)
             M[j]+=M[j]-N[j];

        else
               M[j]=P[j];

}
```

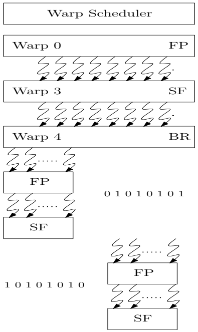
PTX Code has branch instructions

```
setp.gt.s32 %p1, %r1, 0;
mul.wide.s32 %rd6, %r1, 4;
add.s64 %rd1, %rd5, %rd6;
@%p1 bra BB0_2;
bra.uni BB0_1;
BB0_2:
ld.global.f32 %f4, [%rd12];
ld.global.f32 %f5, [%rd1];
sub.f32 %f6, %f5, %f4;
add.f32 %f7, %f5, %f6;
bra.uni BB0_3;
BB0_1:
ld.global.f32 %f7, [%rd9];
BB0_3:
st.global.f32 [%rd1], %f7;
ret;
```

Branching using PTX Instructions

- The PTX Assembler maintains internal masks a branch synchronization stack and special markers
- The PTX Assembler sets a **branch synchronization marker** first for the divergent `if` statement that pushes the active mask on a stack inside each SIMD thread
- Depending on the value of the mask relevant threads execute instructions,
- Once the instructions in the `if` block are finished, the active mask is popped from the stack, flipped and pushed back.

Warp Scheduling



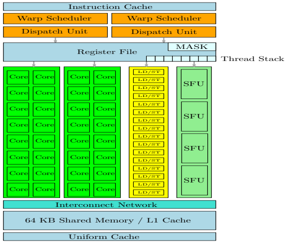


Figure: Solution with branch synchronization stack