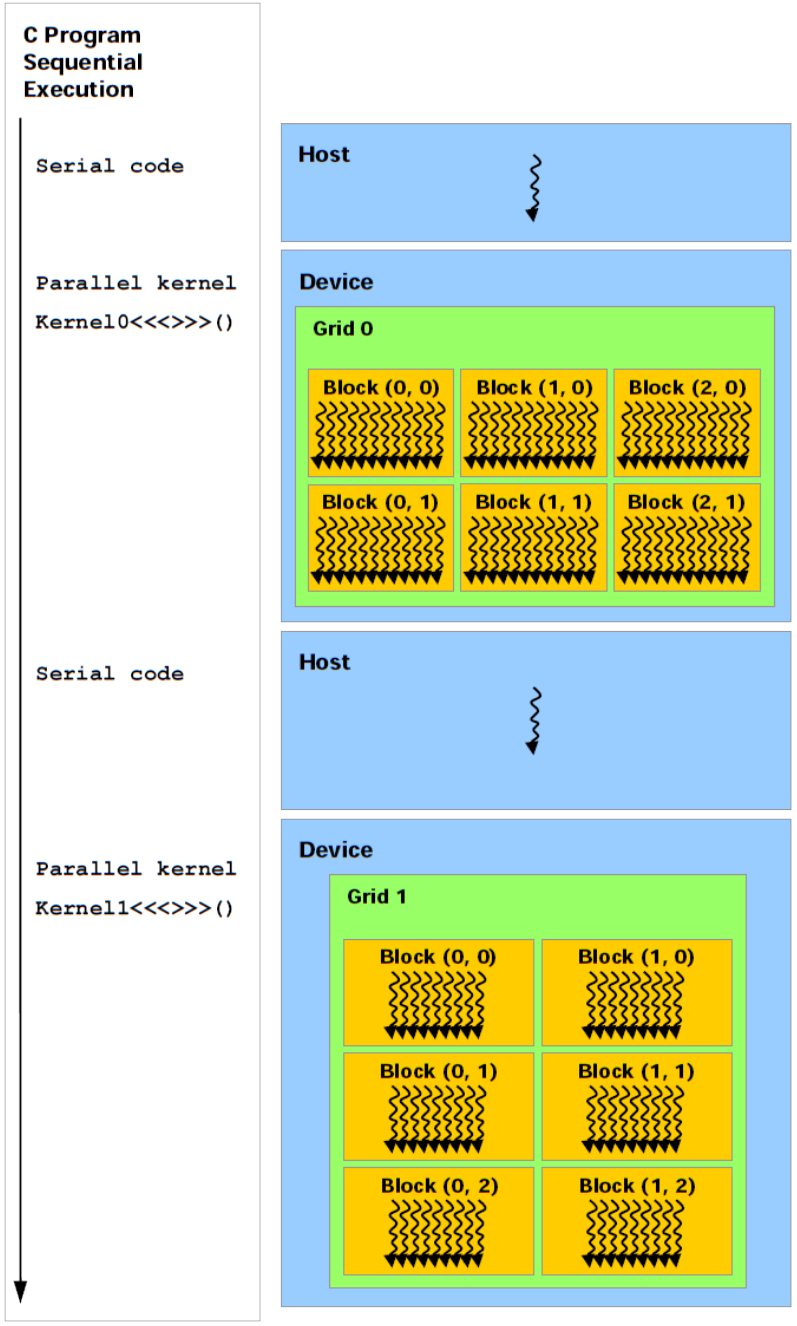
Latency hiding by Multicontext Reconfiguration

Grzegorz Jabłoński

NVIDIA CUDA

- NVIDIA proprietary
- Formerly known as “Compute Unified Device Architecture”
- Extensions to C to allow better control of GPU capabilities
- Modest extensions but major rewriting of the code
- No Fortran version available



CUDA Example

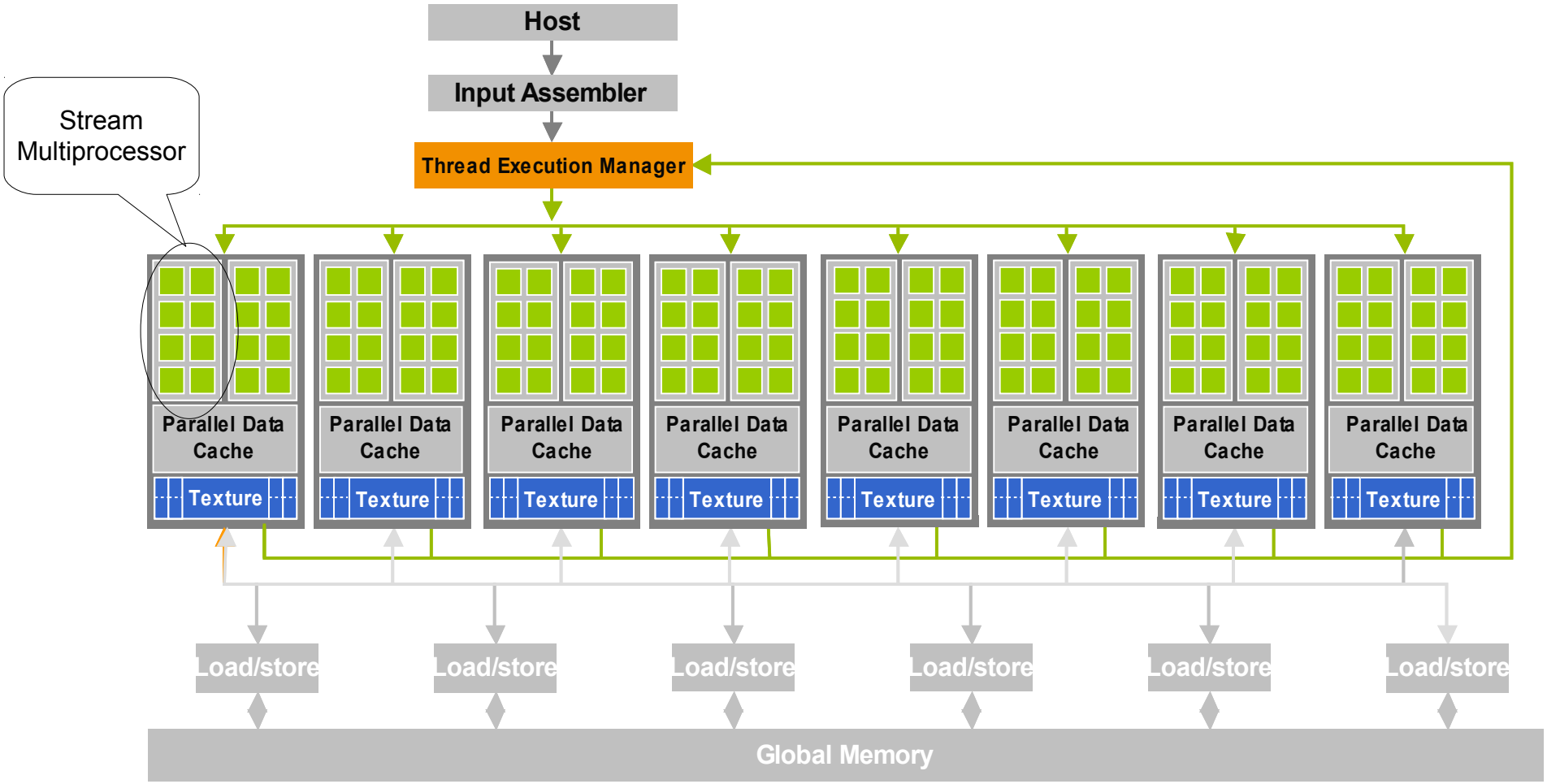
```
#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/

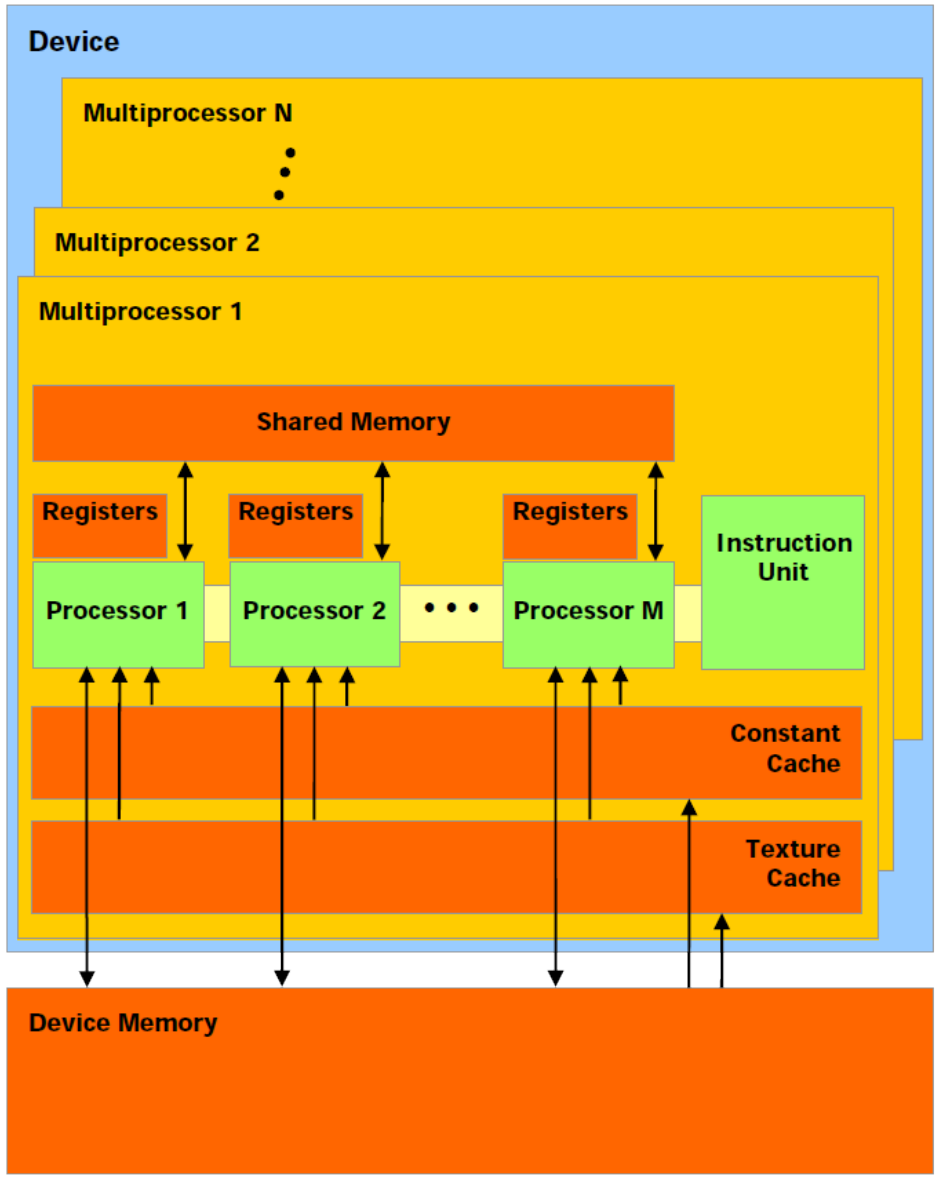
G80 Architecture – Computation Mode

- Processors execute computing threads
- New operating mode/HW interface for computing



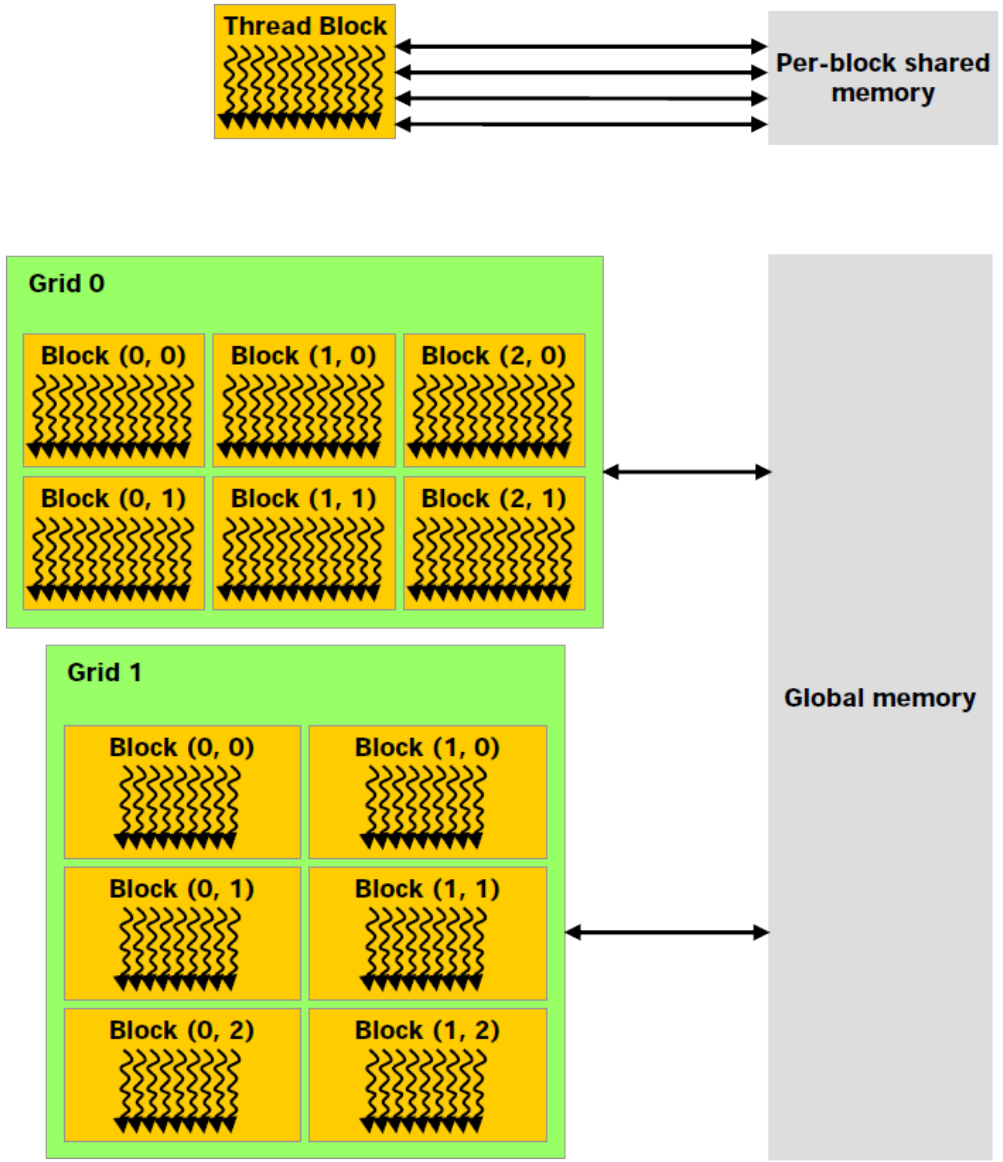
The Multiprocessor

- A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory.
- The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead.

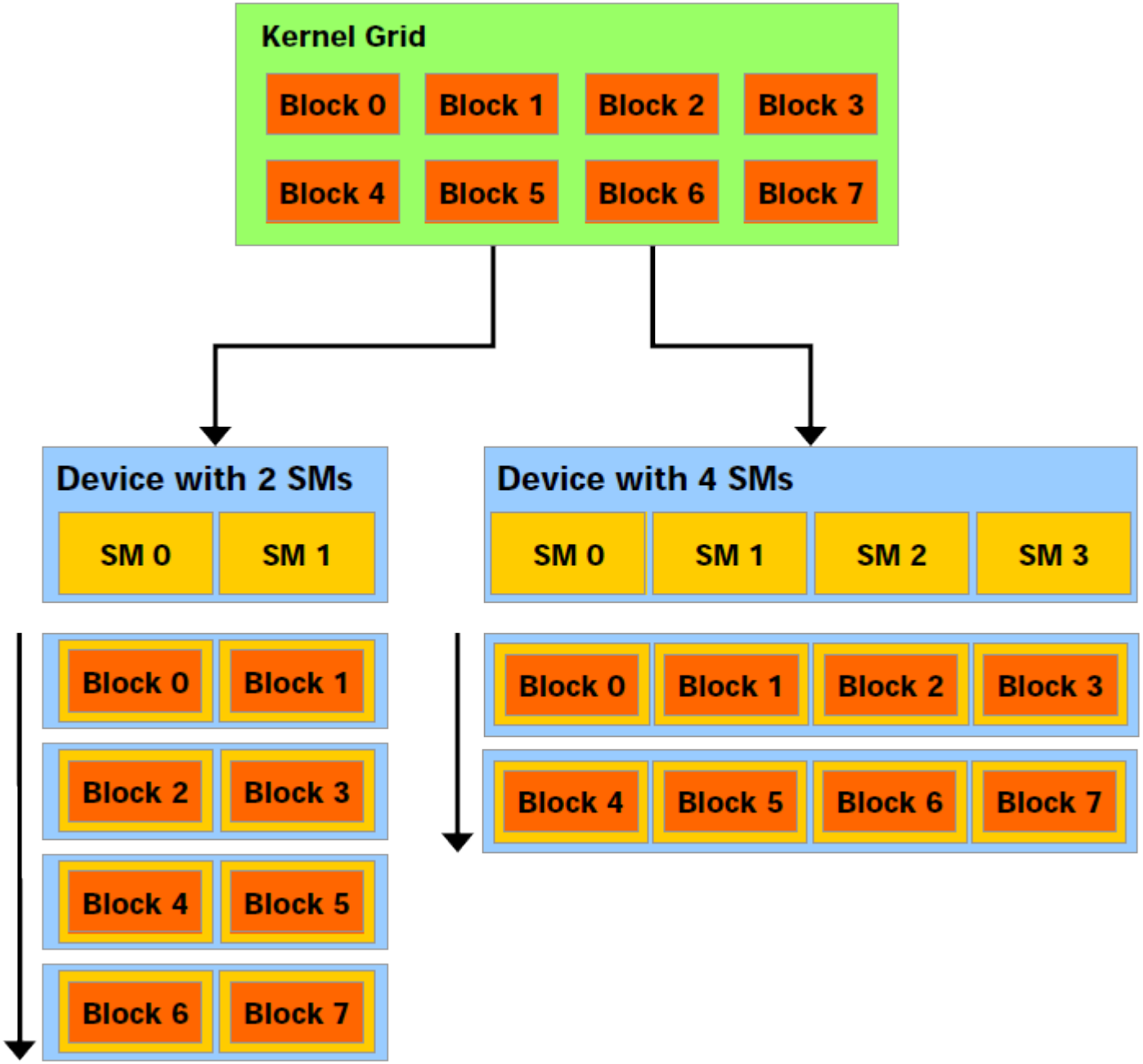


The Warps

- Running threads are divided into blocks, which share the multiprocessor and can exchange data through the local memory
- Multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread).
  - The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.
  - The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.



The Mapping Between Blocks and SMs



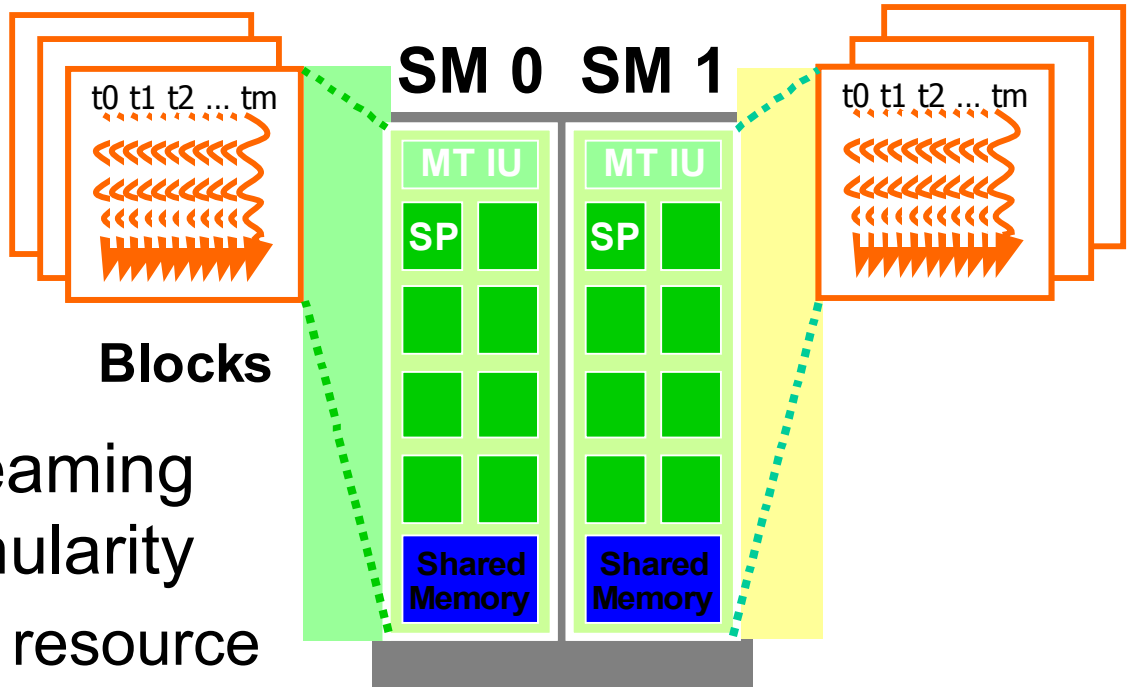
The Warps

- Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.
- When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit.
- Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp.
  - A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.
  - If threads of a warp diverge via a datadependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.
  - Branch divergence occurs only within a warp; different warps execute independently regardless

Compute Capability

- NVIDIA defines the computing capabilities of their devices by a number
- G80 has a Compute Capability of 1.1:
  - The maximum number of threads per block is 512;
  - The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;
  - The maximum size of each dimension of a grid of thread blocks is 65535;
  - The warp size is 32 threads;
  - The number of registers per multiprocessor is 8192;
  - The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks;
  - The total amount of constant memory is 64 KB;
  - The total amount of local memory per thread is 16 KB;
  - The cache working set for constant memory is 8 KB per multiprocessor;
  - The cache working set for texture memory varies between 6 and 8 KB per multiprocessor;
  - The maximum number of active blocks per multiprocessor is 8;
  - The maximum number of active warps per multiprocessor is 24;
  - The maximum number of active threads per multiprocessor is 768;
  - The limit on kernel size is 2 millions of microcode instructions;

G80 Example: Executing Thread Blocks



- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to 8 blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps

G80 Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions, a minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency