M02: High Performance Computing with CUDA

Parallel Programming with CUDA

Ian Buck

Outline

- CUDA model
- CUDA programming basics
- Tools
- GPU architecture for computing
- Q&A

What is CUDA?

- C with minimal extensions
- CUDA goals:
  - Scale code to 100s of cores
  - Scale code to 1000s of parallel threads
  - Allow heterogeneous computing:
    - For example: CPU + GPU
- CUDA defines:
  - Programming model
  - Memory model

CUDA Programming Model

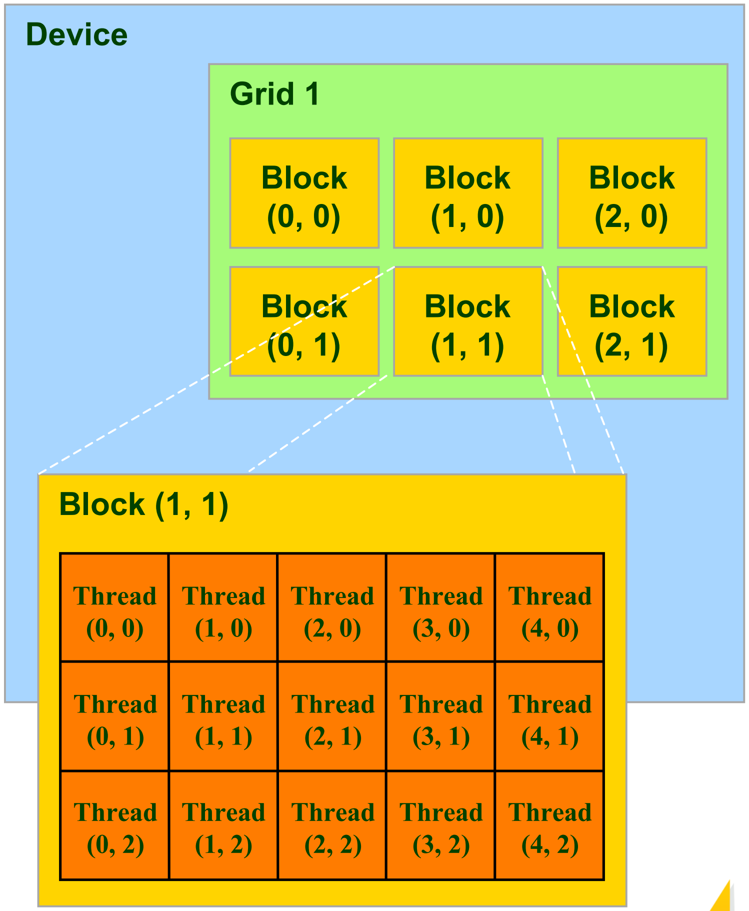
- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

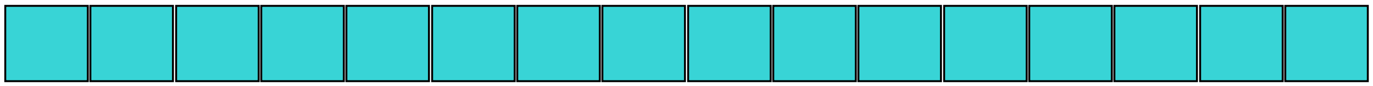
IDs and Dimensions

- **Threads:**
  - **3D IDs, unique within a block**
- **Blocks:**
  - **2D IDs, unique within a grid**
- **Dimensions set at launch time**
  - **Can be unique for each section**
- **Built-in variables:**
  - **threadIdx, blockIdx**
  - **blockDim, gridDim**



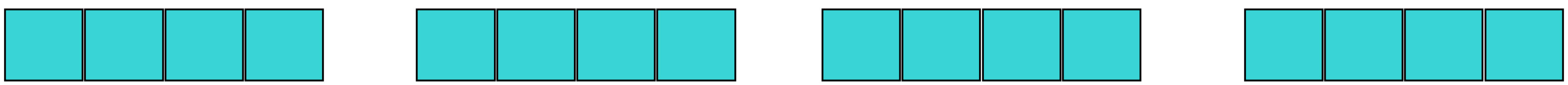
Example: Increment Array Elements

Increment N-element vector a by scalar b



Let’s assume N=16, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```



blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
                a[idx] = a[idx] + b;
}


void main()
{
    .....
        increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if( idx < N)
                a[idx] = a[idx] + b;
}


void main()
{
    .....
        dim3 dimBlock (blocksize);
        dim3 dimGrid( ceil( N / (float)blocksize)  );
        increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Minimal Kernel for 2D data

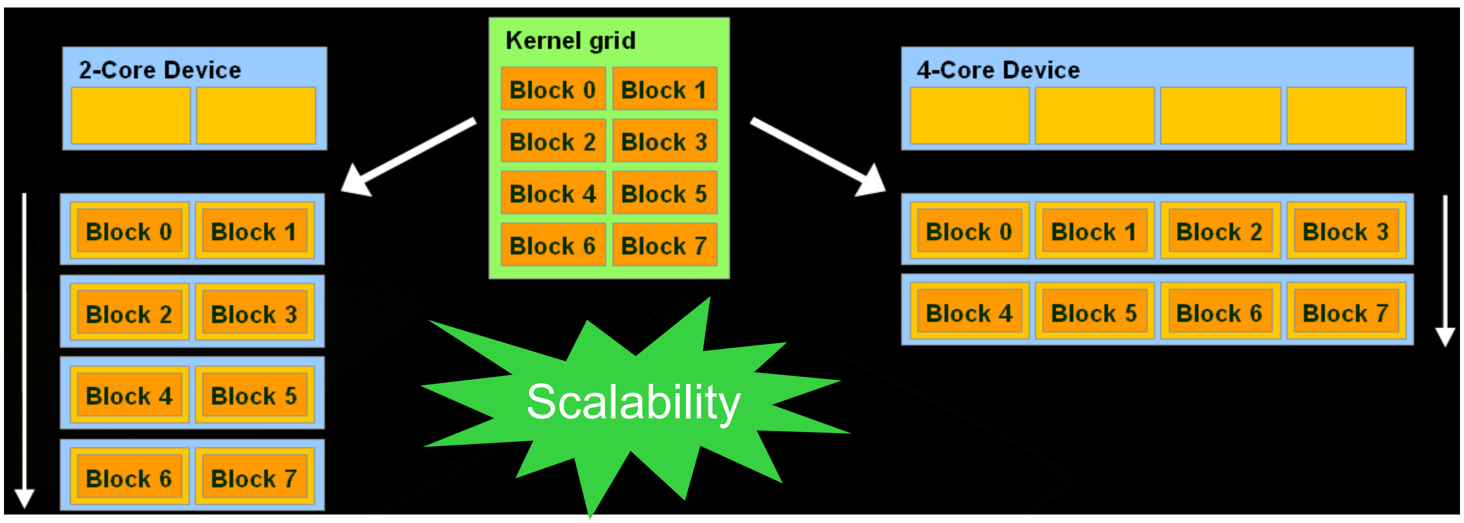
```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Blocks must be independent

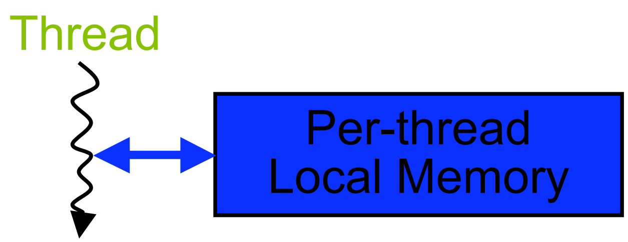
- Thread blocks can run in any order
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices

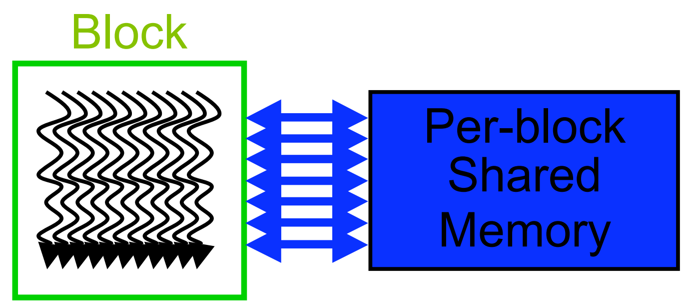


Memory Model

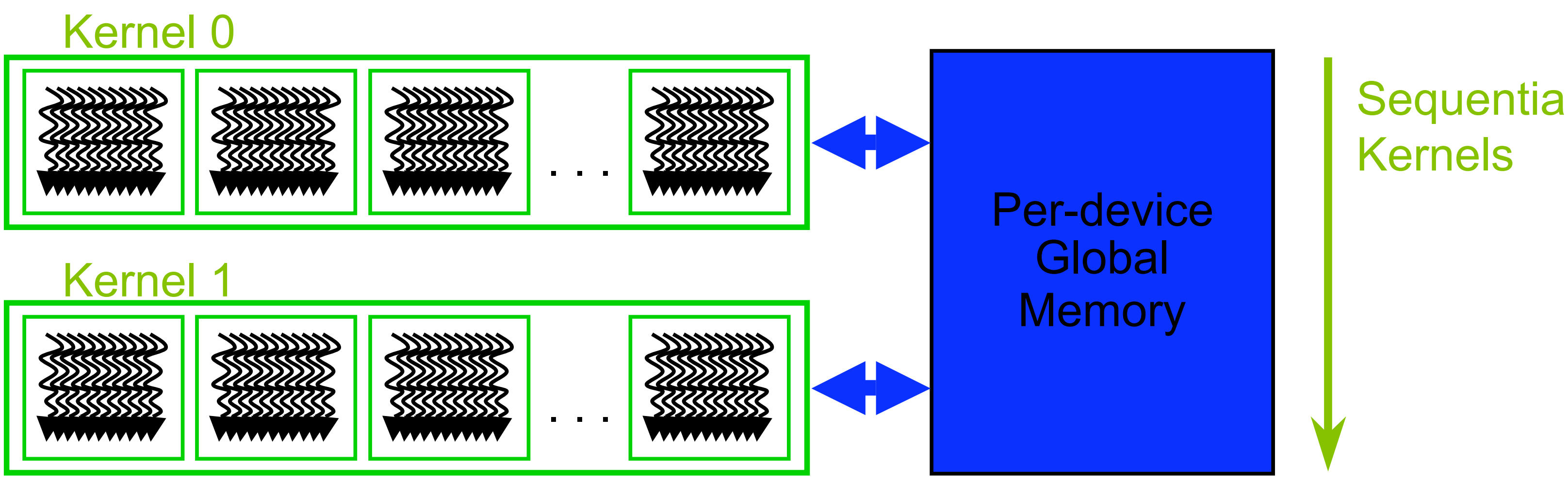
- Local storage
  - Each thread has own local storage
  - Data lifetime = thread lifetime
- Shared memory
  - Each thread block has own shared memory
    - Accessible only by threads within that block
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

Memory model

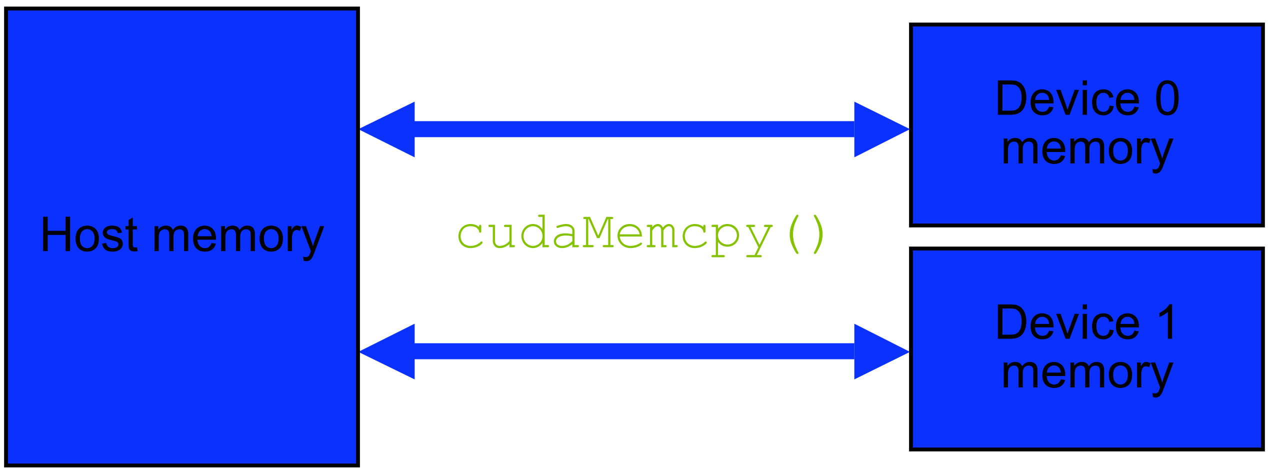




Memory model



Memory model



CUDA Programming Basics

Outline of CUDA Basics

- Basics to setup and execute CUDA code:
  - Extensions to C for kernel code
  - GPU memory management
  - GPU kernel launches
- Some additional basic features:
  - Checking CUDA errors
  - CUDA event API
  - Compilation path
- See the Programming Guide for the full API

Code executed on GPU

- C function with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU
    must return void
  - __device__ : called from other GPU functions,
    cannot be launched by the CPU
  - __host__ : can be executed by CPU
  - __host__ and __device__ qualifiers can be combined
    - sample use: overloading operators
- Built-in variables:
  - gridDim, blockDim, blockIdx, threadIdx

Variable Qualifiers (GPU code)

- __device__
  - stored in global memory (not cached, high latency)
  - accessible by all threads
  - lifetime: application
- __constant__
  - stored in global memory (cached)
  - read-only for threads, written by host
  - Lifetime: application
- __shared__
  - stored in shared memory (latency comparable to registers)
  - accessible by all threads in the same threadblock
  - lifetime: block lifetime
- Unqualified variables:
  - Stored in local memory:
    - scalars and built-in vector types are stored in registers
    - arrays are stored in device memory

Kernel Source Code

```
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[ ];  // allocated during kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[ threadIdx.x ] = g_input[ idx ];
    __syncthreads( );

    // compute sum for the threadblock
    for ( int dist = blockDim.x/2;  dist > 0;  dist /= 2 )
    {
        if ( threadIdx.x < dist )
            s_data[ threadIdx.x ] += s_data[ threadIdx.x + dist ];
        __syncthreads( );
    }

    // write the block's sum to global memory
    if ( threadIdx.x == 0 )
        g_output[ blockIdx.x ] = s_data[0];
}
```

Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

GPU Atomic Integer Operations

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
- Requires hardware with 1.1 compute capability

Launching kernels on GPU

- Launch parameters:
  - grid dimensions (up to 2D)
  - thread-block dimensions (up to 3D)
  - shared memory: number of bytes per block
    - for extern smem variables declared without size
    - Optional, 0 by default
  - stream ID
    - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

GPU Memory Allocation / Release

- Host (CPU) manages GPU memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking memcopies are provided

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

Example: Host Code

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b, N);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop,
    int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control one GPU
    - multiple CPU threads can control the same GPU
      - calls are serialized by the driver

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - returns the code for the last error (no error has a code)
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminted character string describing the error

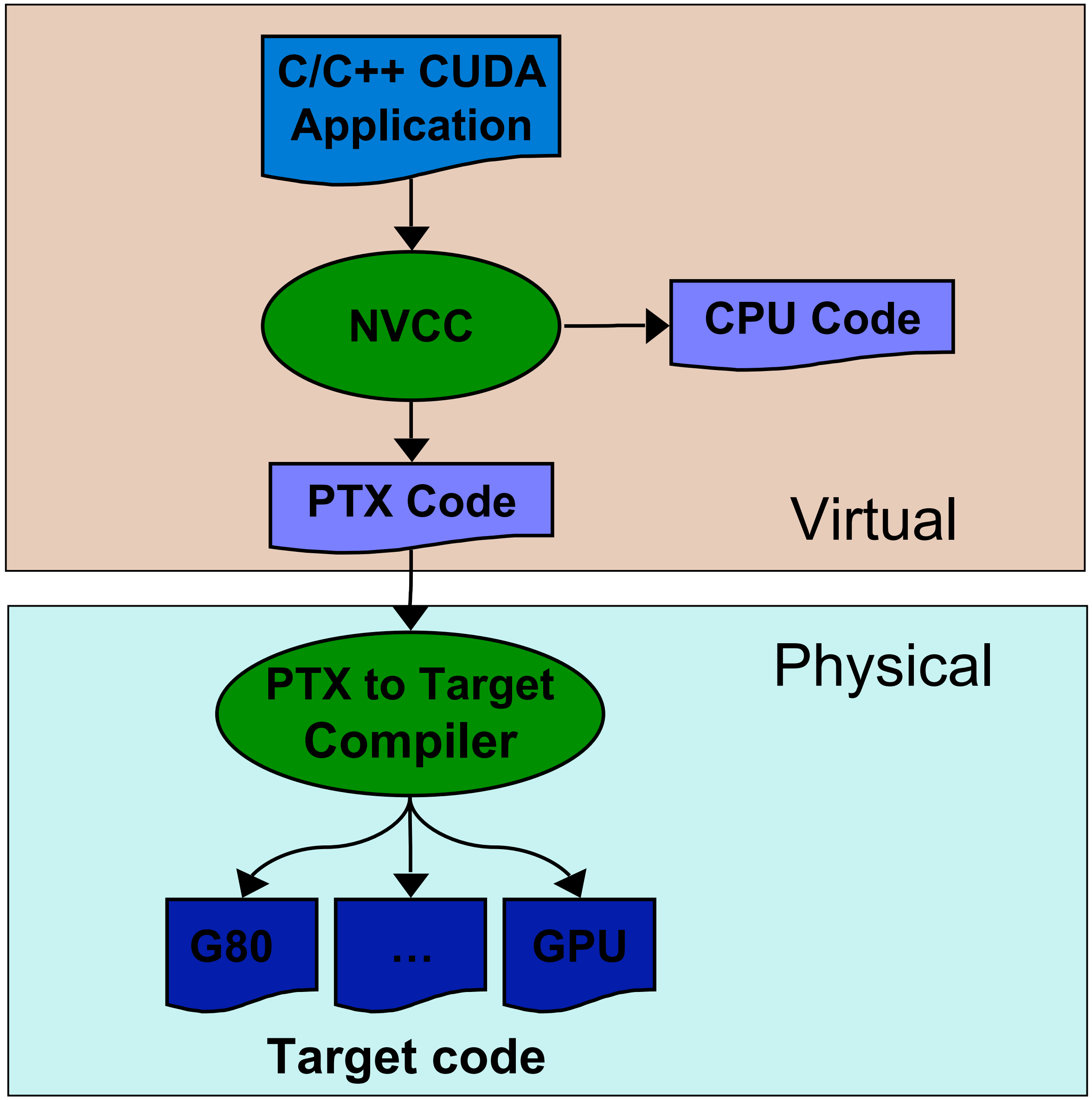
```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

```
cudaEvent_t start, stop;
cudaEventCreate(&start);              cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);   cudaEventDestroy(stop);
```

Compiling CUDA



PTX Example (SAXPY code)

```
cvt.u32.u16     $blockid, %ctaid.x;     // Calculate i from thread/block IDs
cvt.u32.u16     $blocksize, %ntid.x;
cvt.u32.u16     $tid, %tid.x;
mad24.lo.u32    $i, $blockid, $blocksize, $tid;
ld.param.u32    $n, [N];                // Nothing to do if n ≤ i
setp.le.u32     $p1, $n, $i;
@$p1 bra        $L_finish;

mul.lo.u32      $offset, $i, 4;         // Load y[i]
ld.param.u32    $yaddr, [Y];
add.u32         $yaddr, $yaddr, $offset;
ld.global.f32   $y_i, [$yaddr+0];
ld.param.u32    $xaddr, [X];            // Load x[i]
add.u32         $xaddr, $xaddr, $offset;
ld.global.f32   $x_i, [$xaddr+0];

ld.param.f32    $alpha, [ALPHA];        // Compute and store alpha*x[i] + y[i]
mad.f32         $y_i, $alpha, $x_i, $y_i;
st.global.f32   [$yaddr+0], $y_i;

$L_finish:      exit;
```

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - Must be compiled with a C compiler
  - Or PTX object code directly
- An executable with CUDA code requires:
  - The CUDA core library (`cuda`)
  - The CUDA runtime library (`cudart`)
    - if runtime API is used
    - loads cuda library

CUDA Development Tools

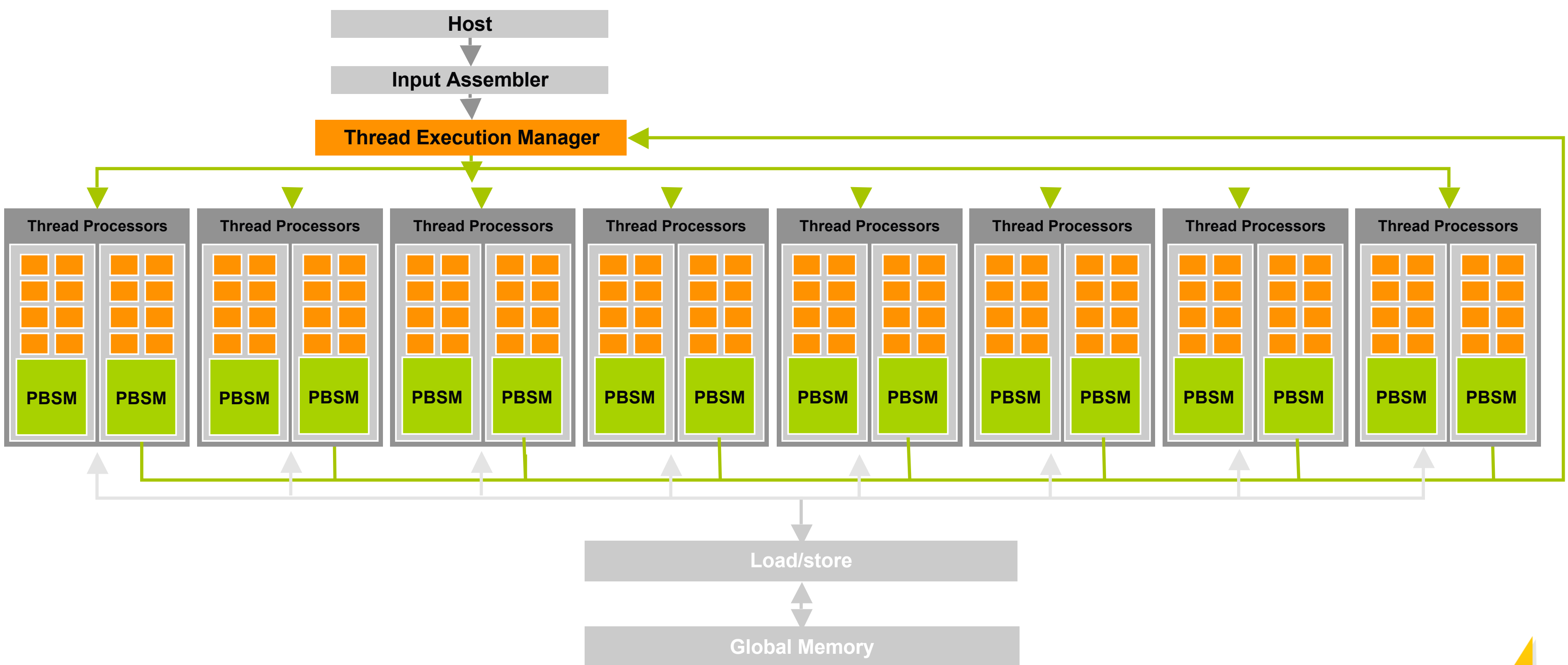
GPU Tools

- Profiler
  - Available now for all supported OSs
  - Command-line or GUI
  - Sampling signals on GPU for:
    - Memory access parameters
    - Execution (serialization, divergence)
- Debugger
  - Runs on the GPU
- Emulation mode
  - Compile and execute in emulation on CPU
  - Allows CPU-style debugging in GPU source

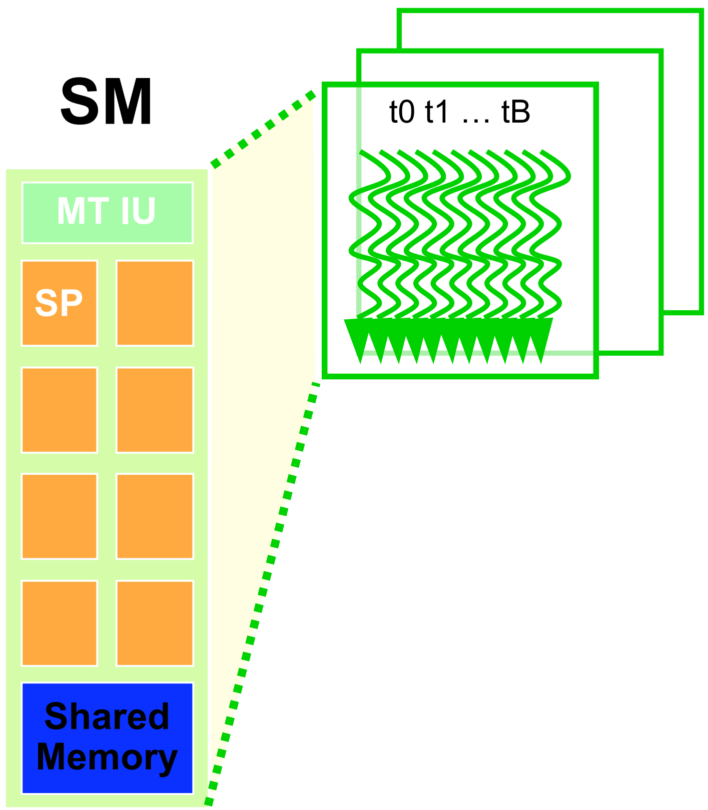
GPU Architecture

Block Diagram (G80 Family)

- G80 (launched Nov 2006)
- 128 Thread Processors execute kernel threads
- Up to 12,288 parallel threads active



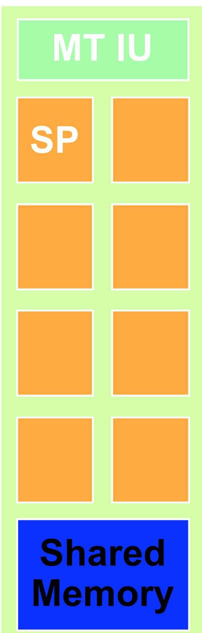
Streaming Multiprocessor (SM)



- **Processing elements**
  - 8 scalar thread processors (SP)
  - 32 GFLOPS peak at 1.35 GHz
  - 8192 32-bit registers (32KB)
    - ½ MB total register file space!
  - usual ops: float, int, branch, …
- **Hardware multithreading**
  - up to 8 blocks resident at once
  - up to 768 active threads in total
- **16KB on-chip memory**
  - low latency storage
  - shared among threads of a block
  - supports thread communication

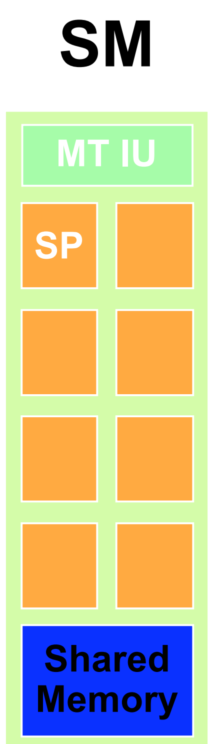
Hardware Multithreading

SM



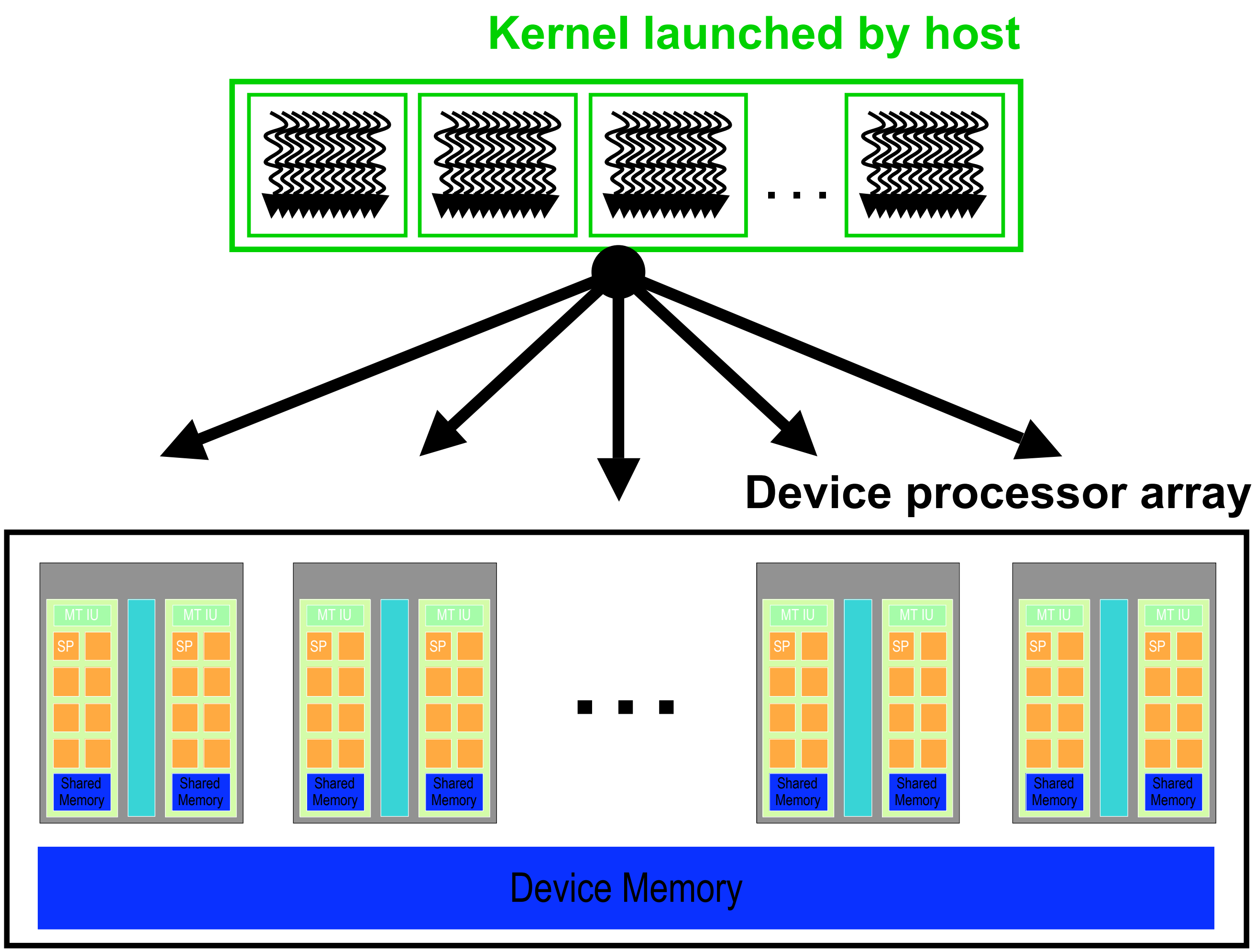
- Hardware allocates resources to blocks
  - blocks need: thread slots, registers, shared memory
  - blocks don't run until resources are available
- Hardware schedules threads
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free – every cycle
- Hardware relies on threads to hide latency
  - i.e., parallelism is necessary for performance

SIMT Thread Execution

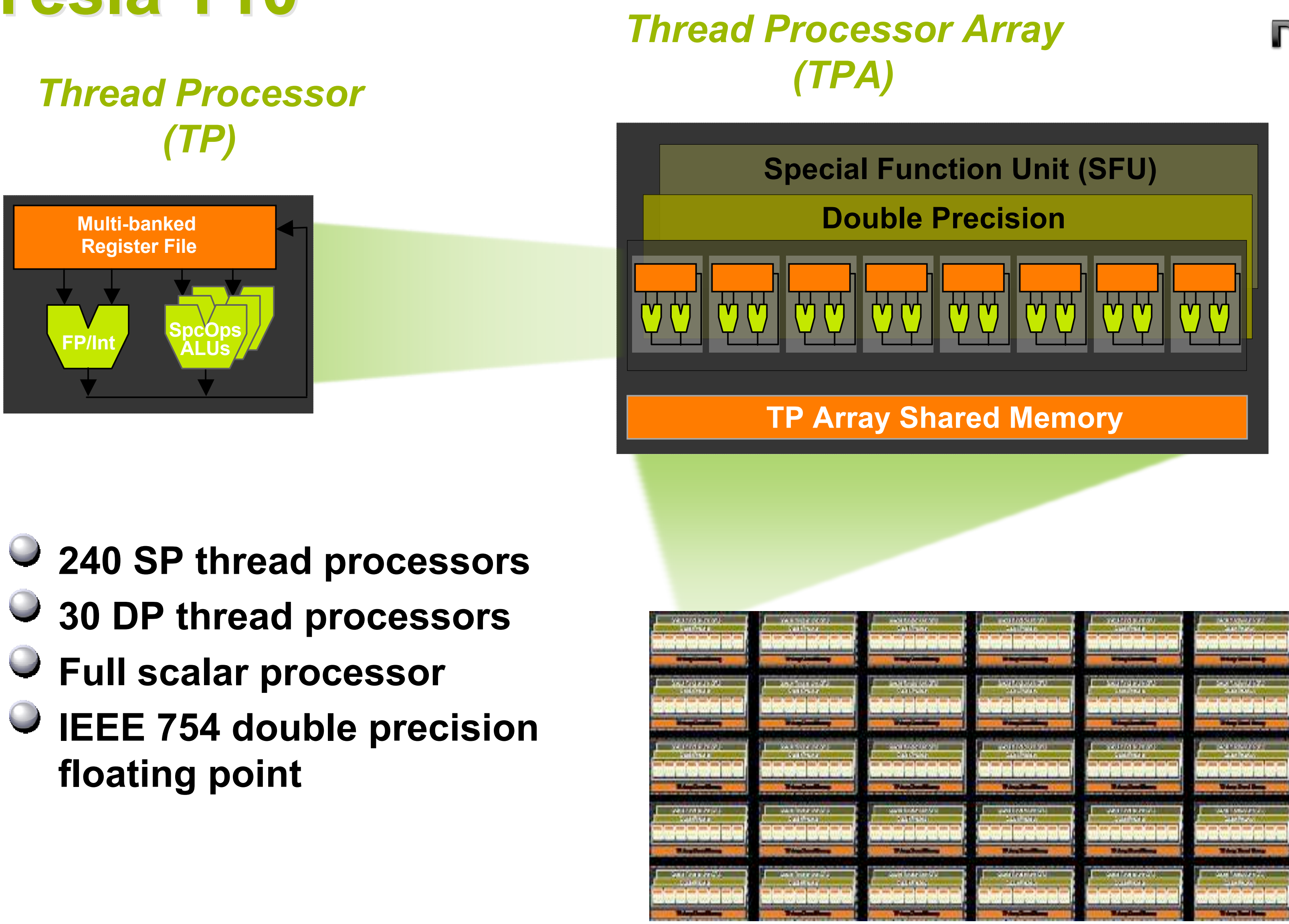


- Groups of 32 threads formed into warps
  - always executing same instruction
  - shared instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- Warps are the primitive unit of scheduling
- SIMT execution is an implementation choice
  - sharing control logic leaves more space for ALUs
  - largely invisible to programmer
  - must understand for performance, not correctness

Blocks Run on Multiprocessors



Tesla T10



- 240 SP thread processors
- 30 DP thread processors
- Full scalar processor
- IEEE 754 double precision floating point