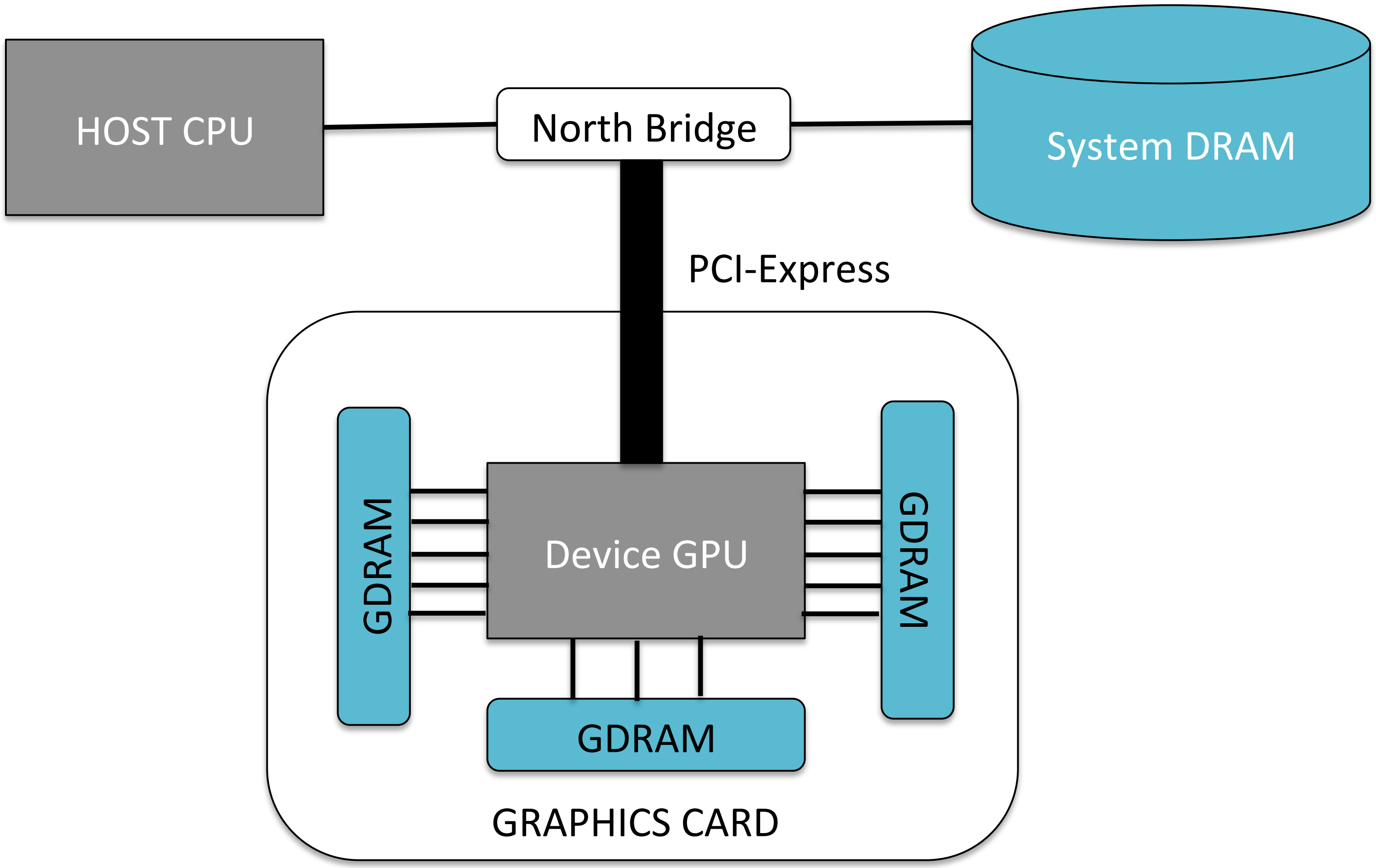
Introduction to CUDA

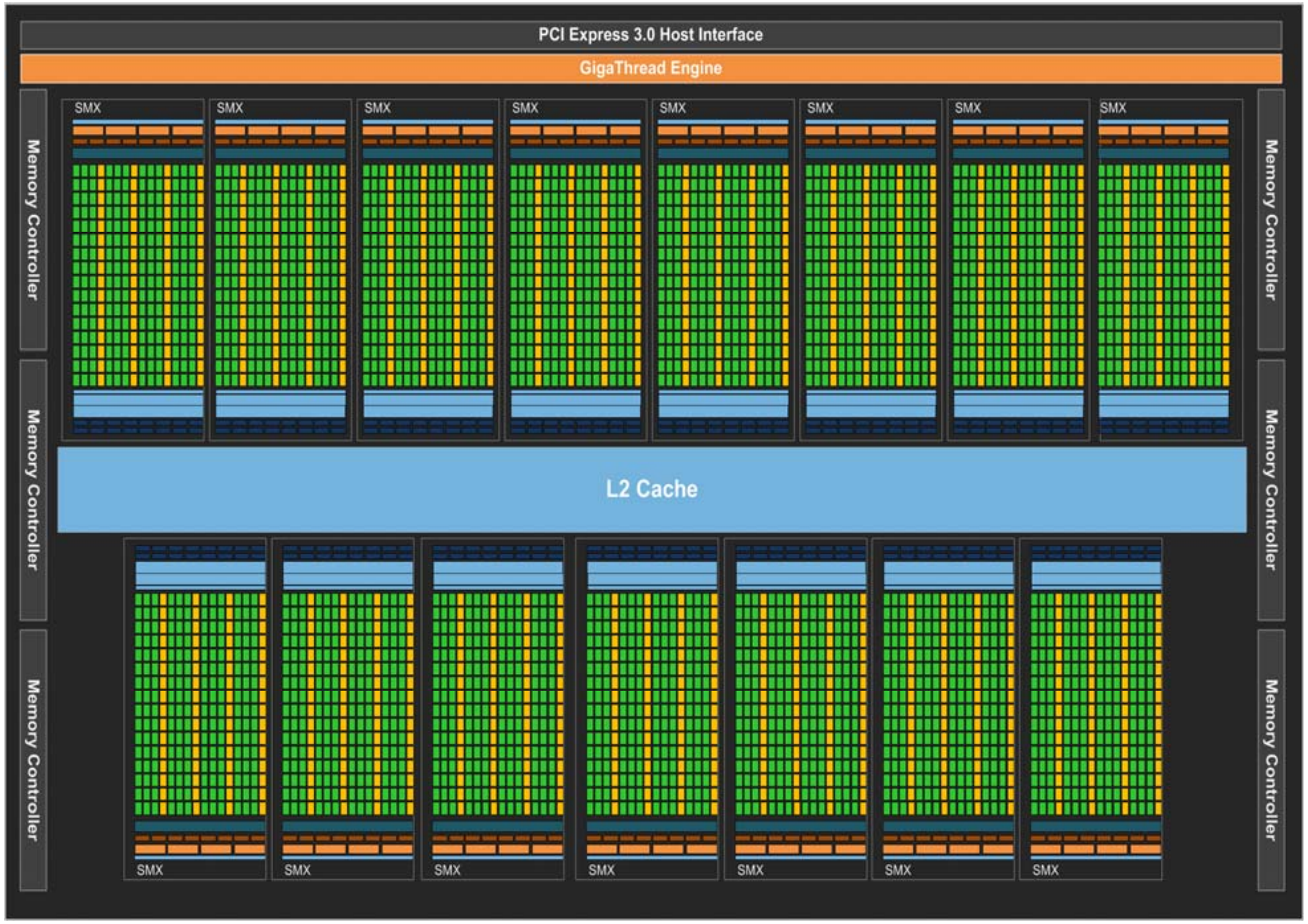
Subodh Sharma

Indian Institute of Technology Delhi

GPU Architecture

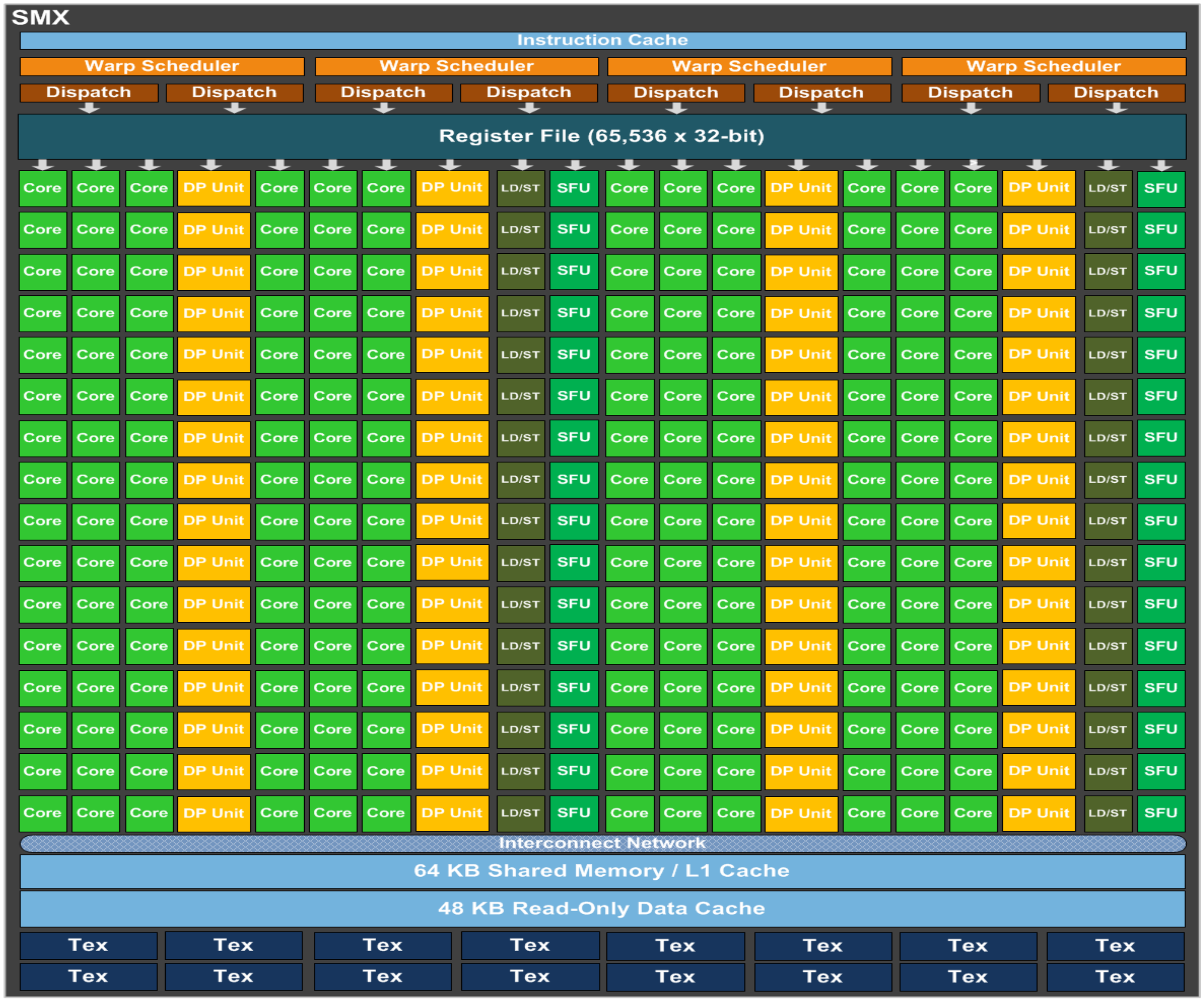


Device GPU



Kepler GK110 Full chip block diagram

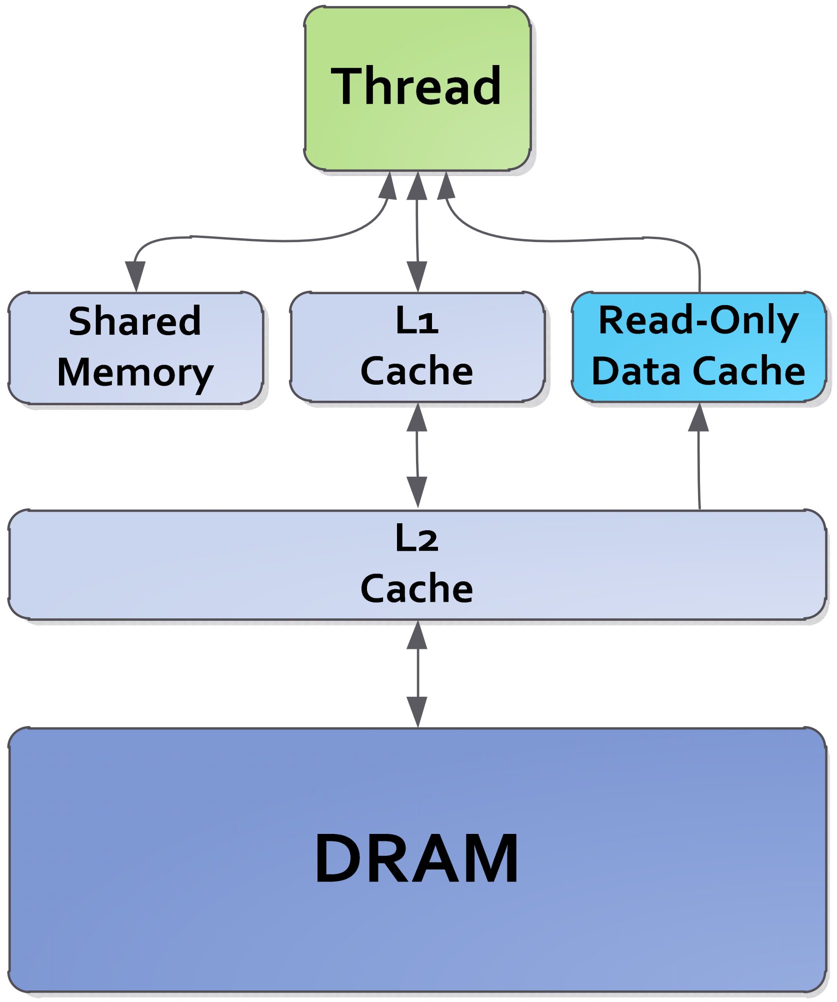
SMX (Streaming Multiprocessor)



SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

Memory Hierarchy

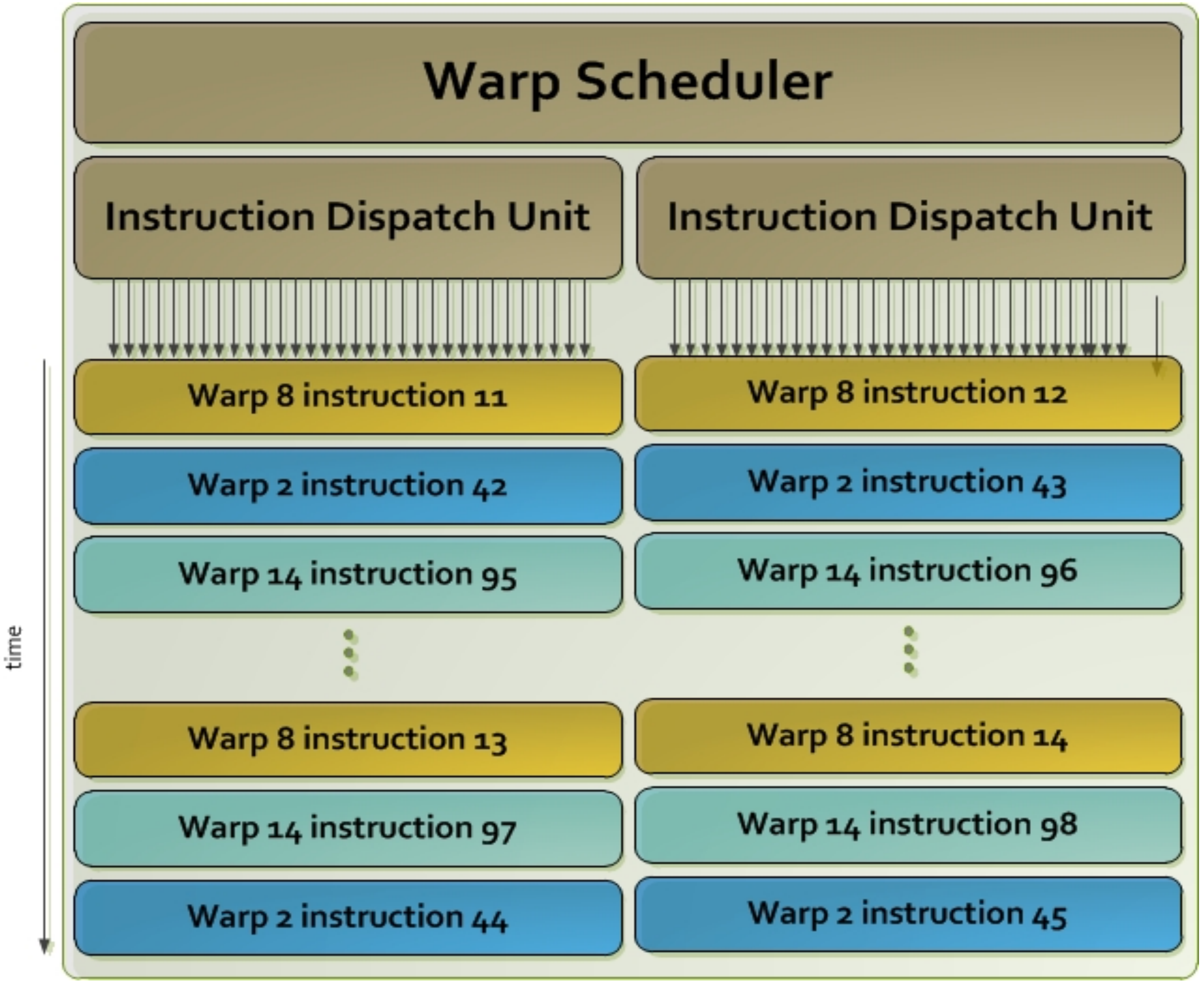


- 64Kb configurable shared mem & L1 cache (16,32,48)
- Shared mem b/w = 256 B/core/clock
- Kepler introduces 48KB read-only cache for ld ops
- L2 – 1536 KB

SMX

- 192 single precision cores
  - Each core has fully pipelined FP and Integer ALU
- 32 special function units
  - fast approximate transcendental ops
- 32 ld/st units and 32 double precision units
- 4 warp schedulers
  - each warp schedule has 2 instruction dispatch units

Warp Scheduler

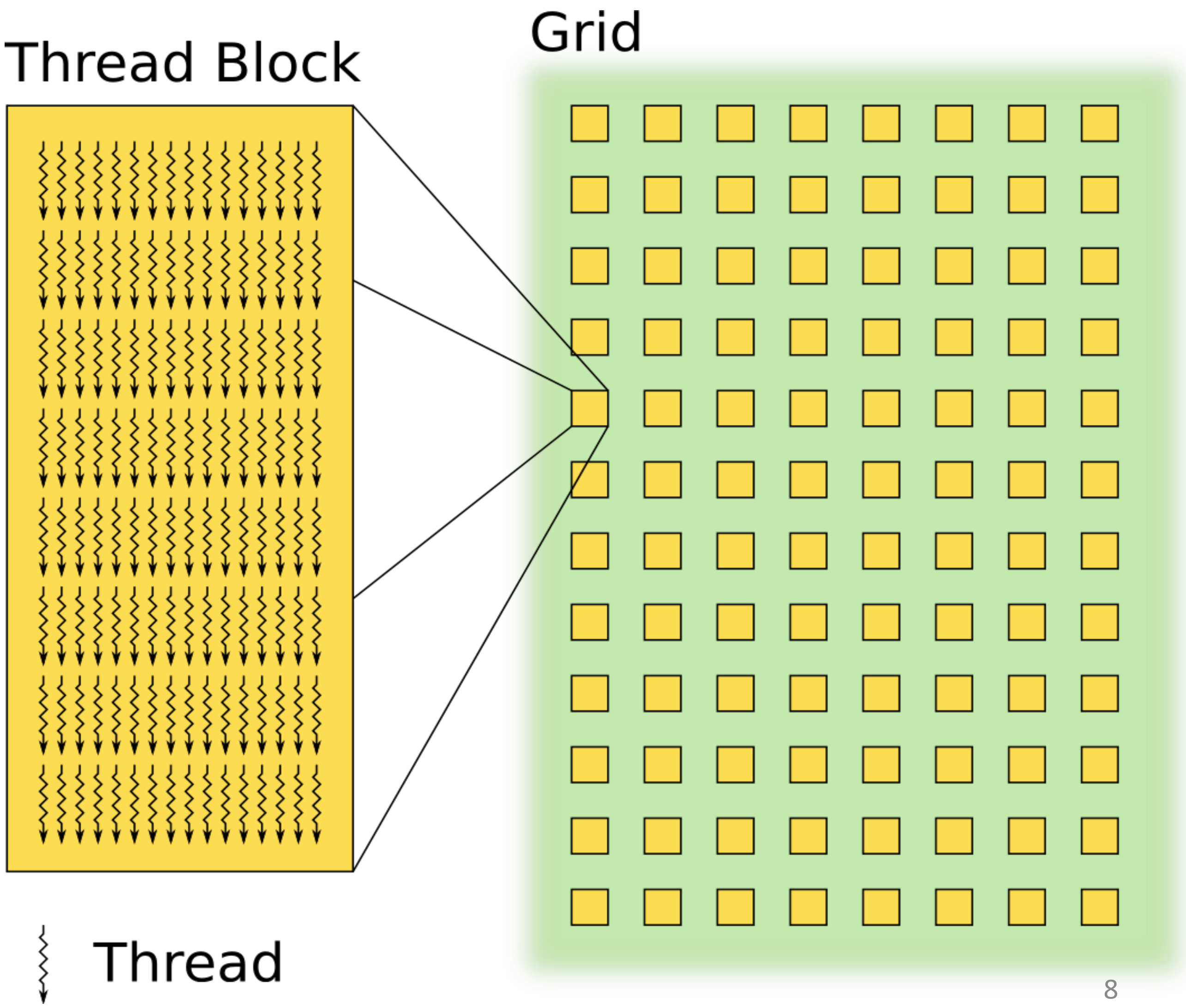


- 2 independent instructions/ warp
- scheduler logic: pick the best warp to go next
  - thread block level scheduling (Giga thread Engine)

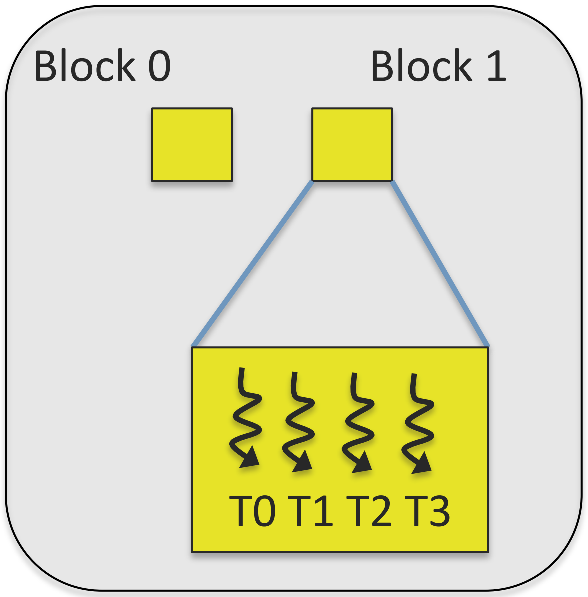
Thread Hierarchy in CUDA

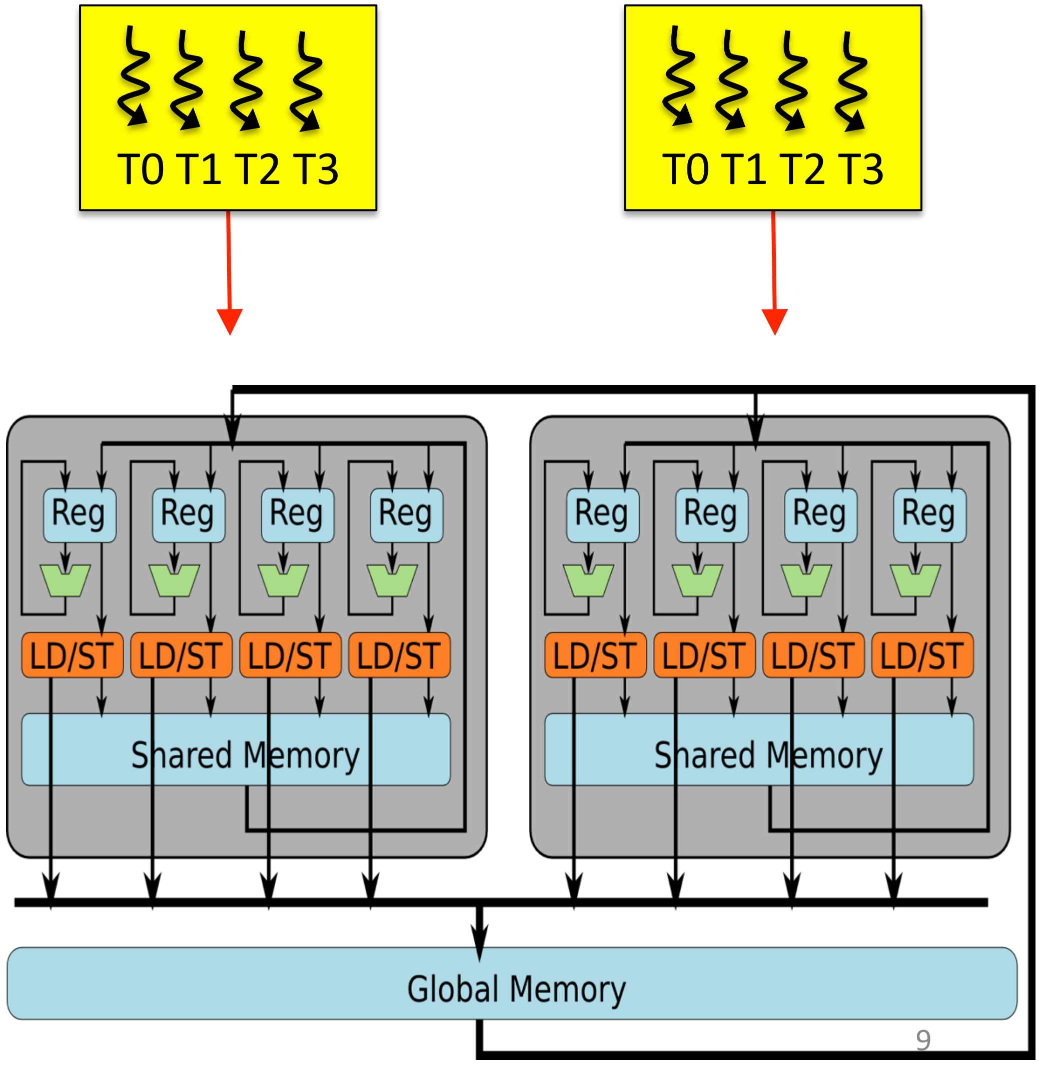
Grid
contains
Thread Blocks

Thread Block
contains
Threads



How Are Threads Scheduled?

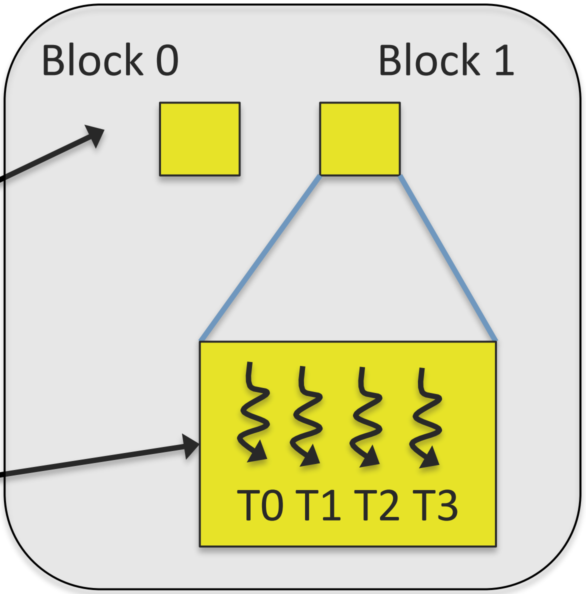




Simple CUDA

```
int A[2][4];
for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        A[i][j]++;
```

```
int A[2][4];
kernelF<<<2,4>>>(A);
__global__  kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```



/* Grid sz= 2, Block sz=4, total =
* 8 threads
* all threads run same kernel
*/

Keyword – indicates that the code runs on the device and called from the "host"

Device Qualifiers

- __global__ : on device, called from the host
- __host__: called from and executes on the host
- __device__: called from device & executes on device

Variable Type Qualifiers

- `__device__ type var_name;`
  - resides in GDRAM, scope is lifetime of the app
- `__constant__ type var_name;`
  - resides in constant memory space on the device
  - scope is the lifetime of the application
- `__shared__ type var_name;`
  - resides in the shared memory space of the thread block
  - scope is the lifetime of the block

Memory Management

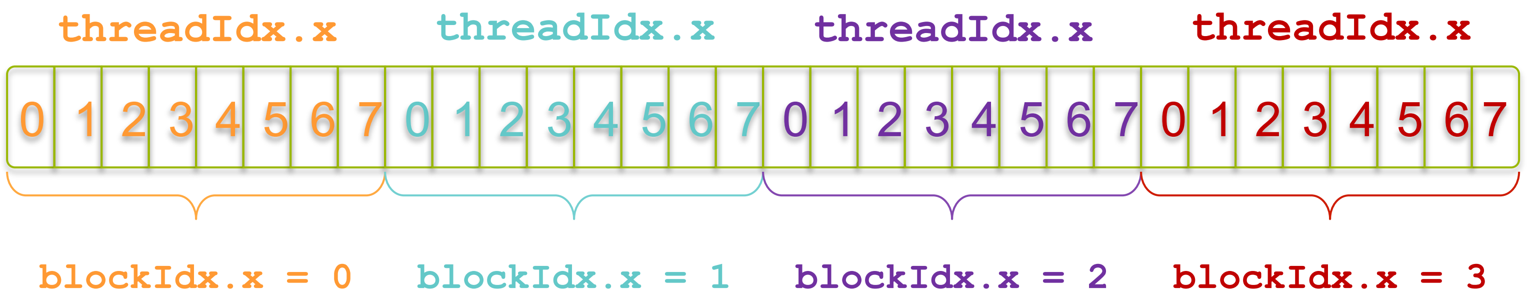
```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
    }
```

- Device ptrs: point to GPU mem, (can be) passed to/from host code
- CUDA APIs: cudaMalloc, cudaFree, cudaMemcpy

Memory Management

```
int main(void) {
    […] // declaration

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    […] // set up initial values

    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,N>>>(d_a, d_b, d_c);

    cudaDeviceSynchronize(); // wait for GPU to finish

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

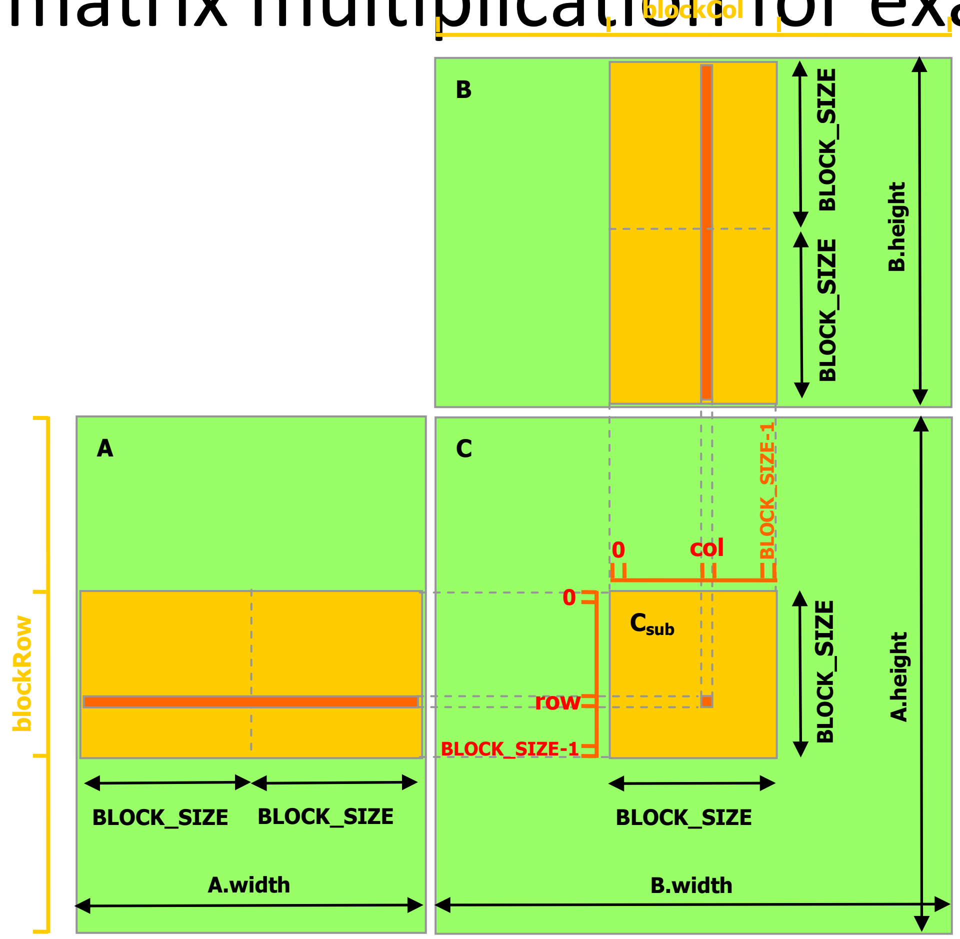
Thread Indexing



- index = threadIdx.x + blockIdx.x * blockDim.x

Shared Memory

- Allocated via **__shared__** qualifier
- Take matrix multiplication for example



Shared Memory - MatMul

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
```

Shared Memory - MatMul

```
 // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Shared Memory - MatMul

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;
    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
```

Shared Memory - Mat Mul

```
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
```

Shared Memory - Mat Mul

```
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
       Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
   }
  // Write Csub to device memory
  // Each thread writes one element
  SetElement(Csub, row, col, Cvalue);
 }
```

Synchronization Primitives

```
__device__ void sum(float* g_idata, float* g_odata)
{
    unsigned int tid = threadIdx.x;
    extern __shared__ float s_data[];
    // Assign initial value
    s_data[tid] = g_idata[...];
    __syncthreads();
   // Perform sum in shared memory.
   // This code sample assumes that the block size is 256
   // (see the reduction sample in the GPU Computing SDK
   // for a complete and general implementation
   if (tid < 128) s_data[tid] += s_data[tid + 128];
   __syncthreads();
   if (tid < 64) s_data[tid] += s_data[tid + 64];
   __syncthreads();
```

Synchronization Primitives

```
if (tid <  32) {
 // No __syncthreads() necessary after each of the
 // following lines (as long as we access the data via
 // a pointer declared as volatile) because the 32 threads
 // in each warp execute in lock-step with each other
  volatile float* s_ptr = s_data;
  s_ptr[tid] += s_ptr[tid + 32];
  s_ptr[tid] += s_ptr[tid + 16];
  s_ptr[tid] += s_ptr[tid +  8];
  s_ptr[tid] += s_ptr[tid +  4];
  s_ptr[tid] += s_ptr[tid +  2];
  s_ptr[tid] += s_ptr[tid +  1];
 }
// Write result for this thread block to global memory
 if (tid == 0) g_odata[blockIdx.x] = s_data[0];

}
```

Synchronization Primitives

- void __threadfence_block()
  - blocks until all global/shared memory accesses by the calling thread prior to this instr. are visible to all threads in the thread block
- void __threadfence();
  - blocks until all shared memory accesses by the calling thread prior to this instr. are visible to all threads in the thread block and all threads in the device for global accesses

Synchronization Primitives

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    float* result)
// Each block sums a subset of the input array
{
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
// Thread 0 of each block stores the partial sum
// to global memory
result[blockIdx.x] = partialSum;
// Thread 0 makes sure its result is visible to
// all other threads
__threadfence();
```

Synchronization Primitives

```
// Thread 0 of each block signals that it is done
unsigned int value = atomicInc(&count, gridDim.x);
// Thread 0 of each block determines if its block is
// the last block to be done
isLastBlockDone = (value == (gridDim.x - 1));
}
// Synchronize to make sure that each thread reads
// the correct value of isLastBlockDone
__syncthreads();
if (isLastBlockDone) {
    // The last block sums the partial sums
    // stored in result[0 .. gridDim.x-1]
   float totalSum = calculateTotalSum(result);
   if (threadIdx.x == 0) {
// Thread 0 of last block stores total sum
// to global memory and resets count so that
// next kernel call works properly
result[0] = totalSum; count = 0;
}}}
```

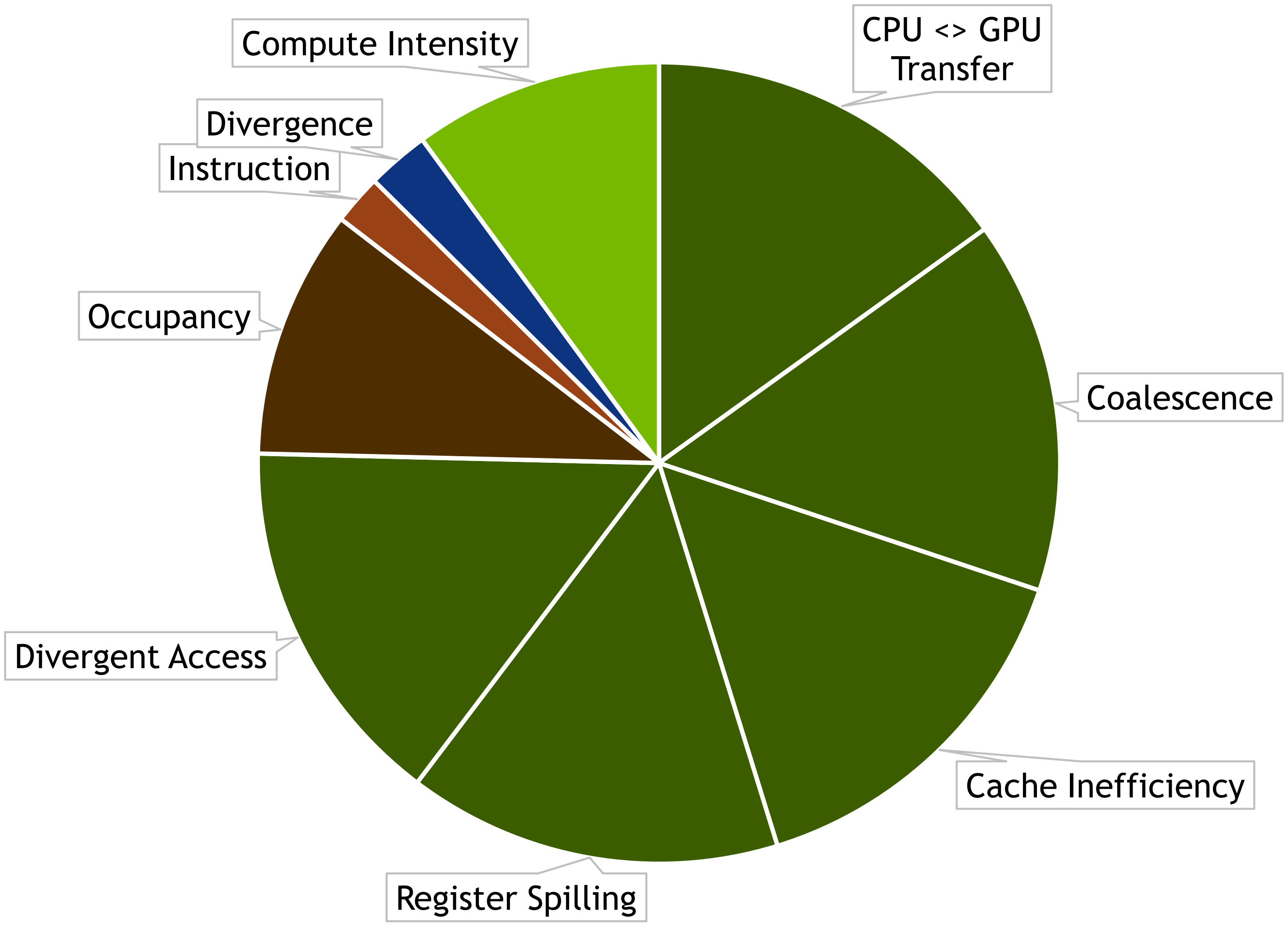
Atomic Functions

- atomicAdd(), atomicSub(), atomicMin(), …
  - Lds 32/64 bit addr, updates and stores in the same addr.
- atomicExch(), atomicCAS(),
- atomicAnd(), atomicOR(), …

Additional Support

- void assert(int expression);
  - if the expression evaluates to 0, then kernel execution stops
- int printf(const char *,)
  - Much like C, except final formatting of the o/p takes place on the host system

Performance Related Guidelines

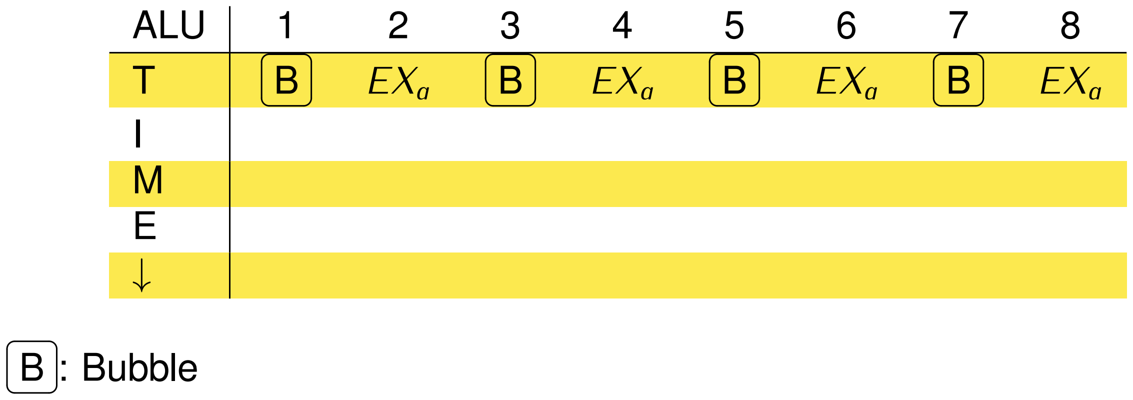


Courtesy: Nvidia S. Jones 2017

Performance Related Guidelines

- Rough thumb of rule
  - Close to 500 threads per SMX
- Use nvprof,
- Minimal control-flow divergence

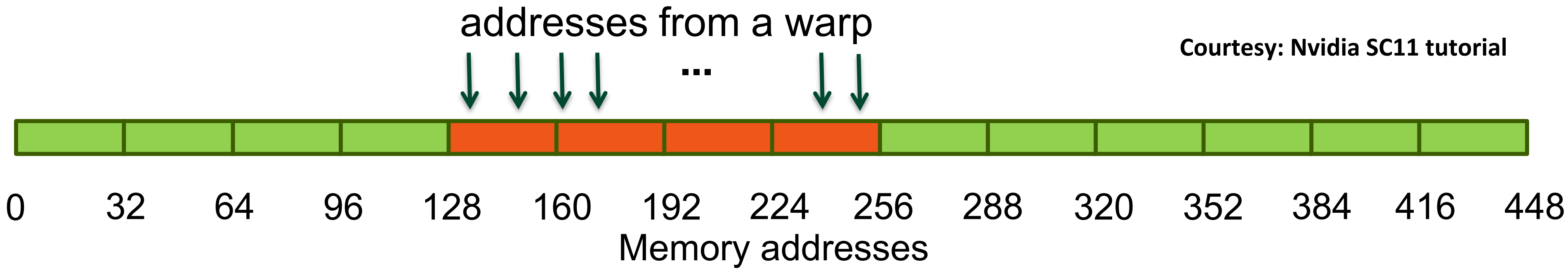
```
if (tid %2){
C1
}
else {C2}
```

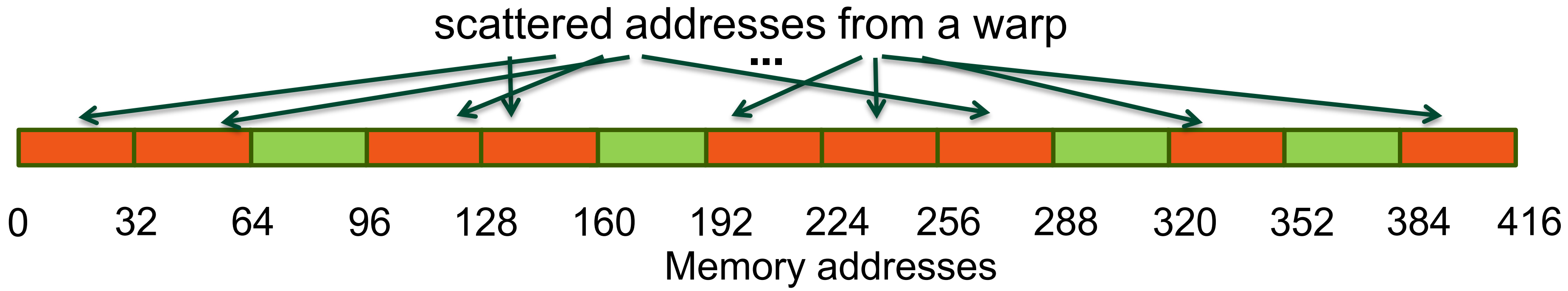


Performance Related Guidelines

- Address request from a warp – converted to line requests
  - Line size 32/128 Bytes

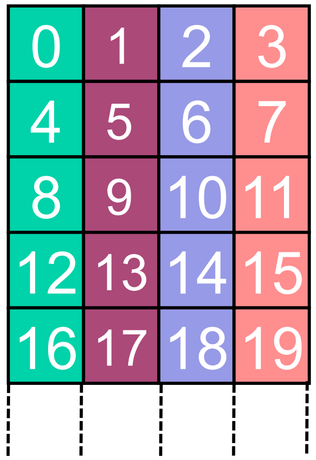
Courtesy: Nvidia SC11 tutorial





More Guidelines

- Avoid shared-mem bank conflicts
- stride = # of banks


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19