Algorithmen für die Echtzeitgrafik

Algorithmen für die Echtzeitgrafik

Daniel Scherzer
scherzer@cg.tuwien.ac.at

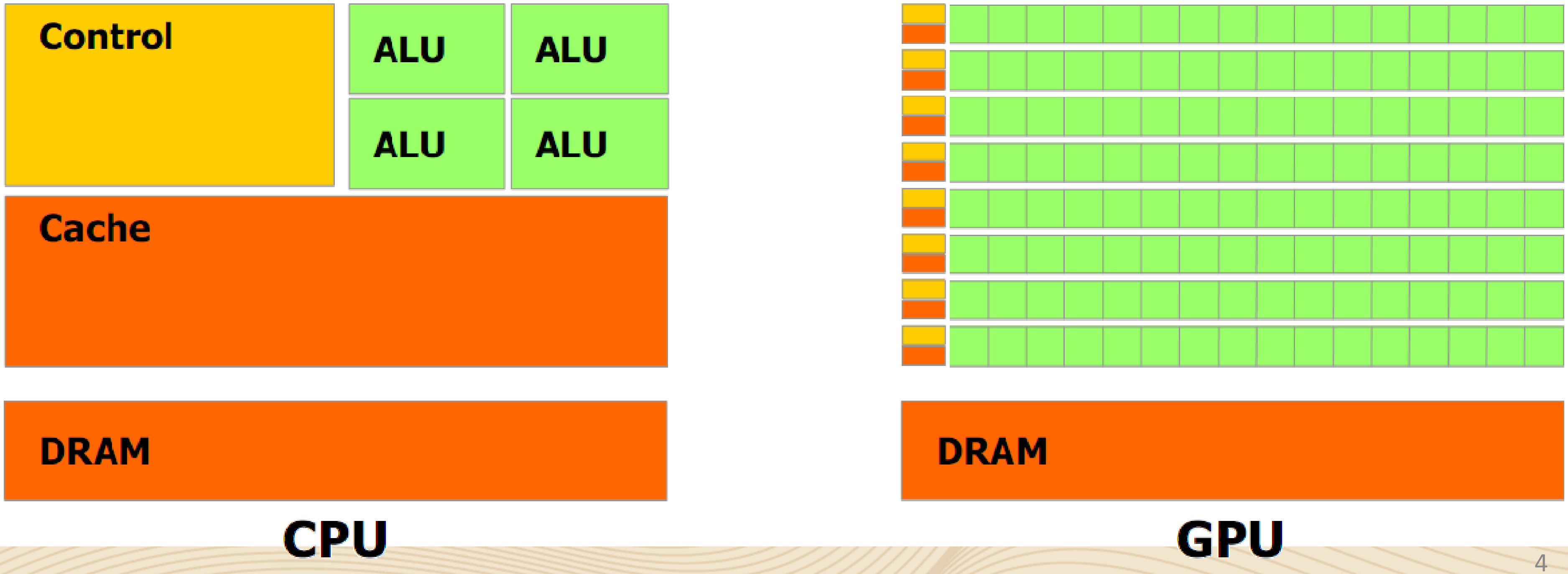
LBI Virtual Archeology

CUDA

Hardware

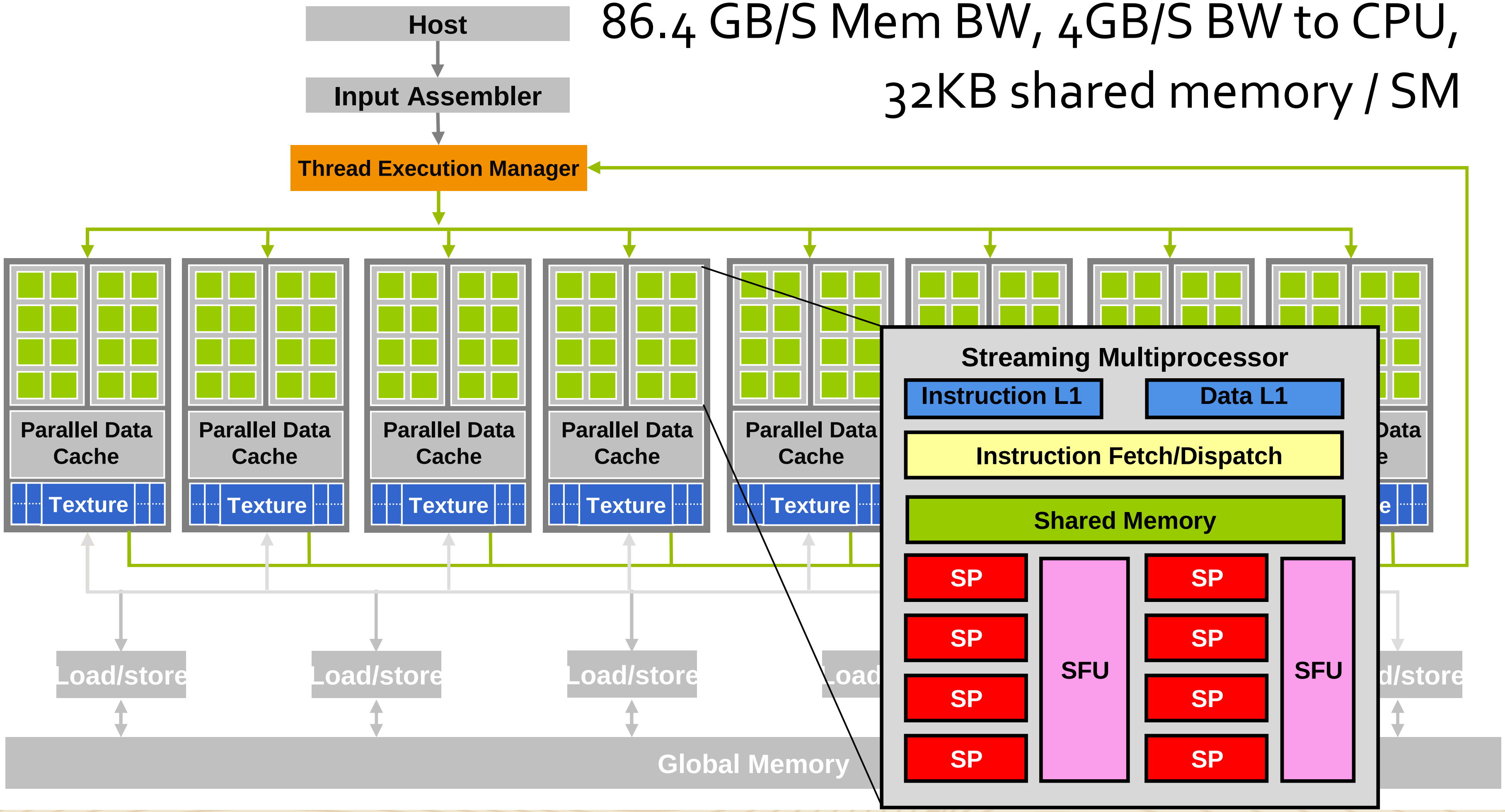
GPU Hardware

- Specialized for
  - Compute-intensive
  - Highly parallel computation
- GPU devotes more transistors to data processing instead of caching

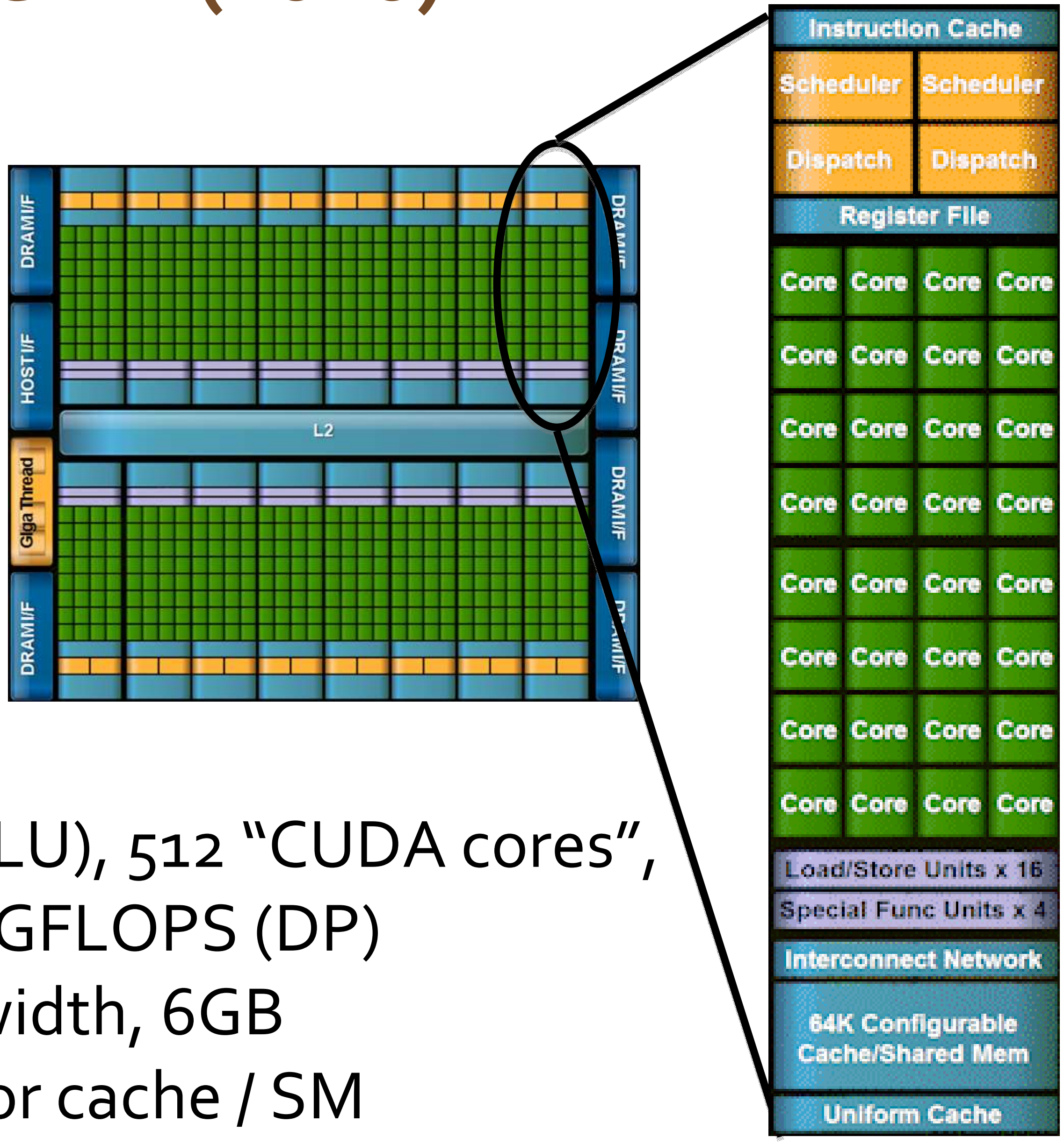


GeForce 8800 (2007)

16 SM (each 8 FPU), >128 FPU's,
367 GFLOPS, 768 MB DRAM,
86.4 GB/S Mem BW, 4GB/S BW to CPU,
32KB shared memory / SM



Fermi (2010)



16 SM (each 32 FPU+ALU), 512 "CUDA cores",
~1.5TFLOPS (SP) / 800GFLOPS (DP)
230 GB/s DRAM Bandwidth, 6GB
64KB shared memory or cache / SM

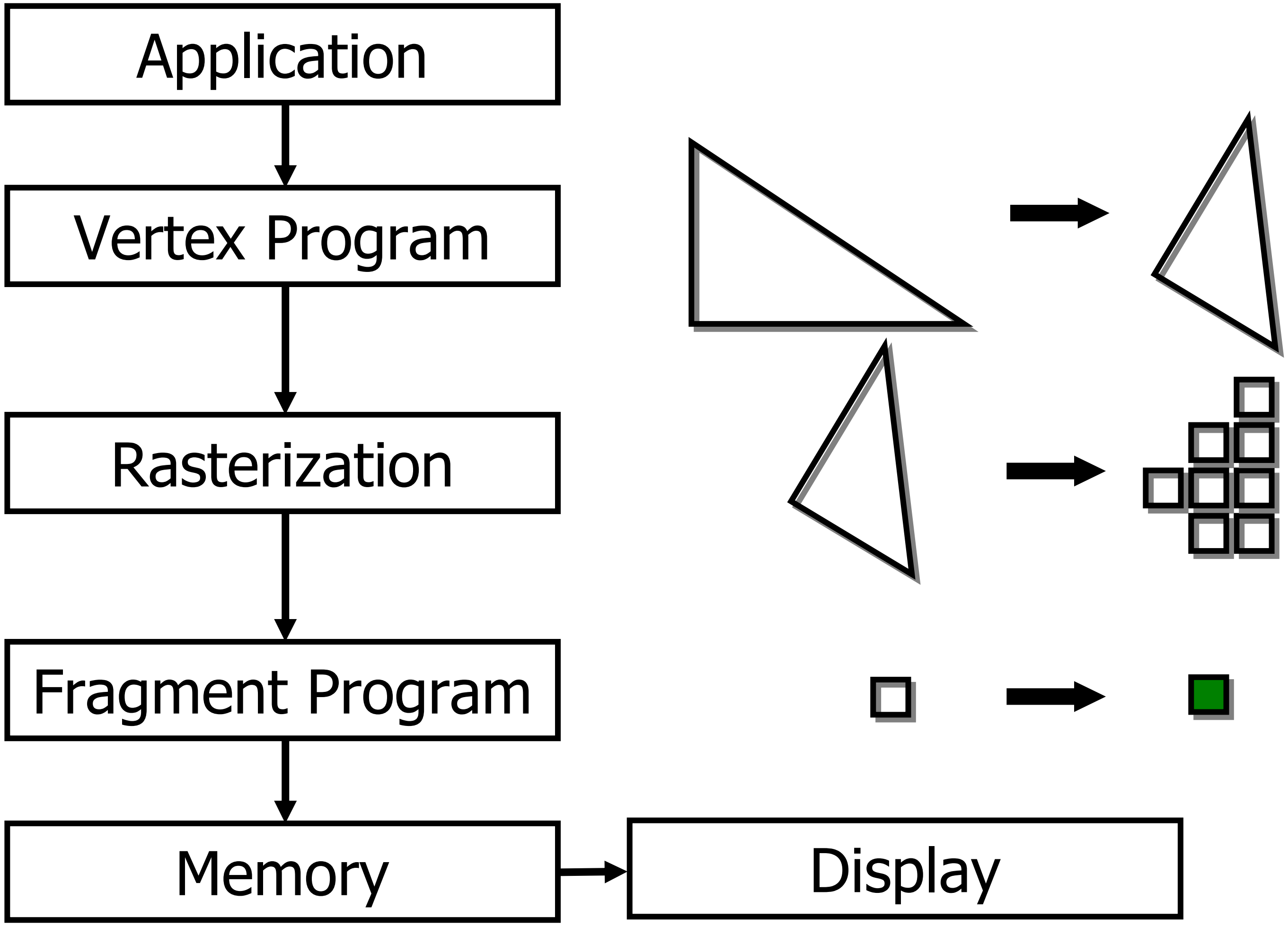
CUDA

GPGPU

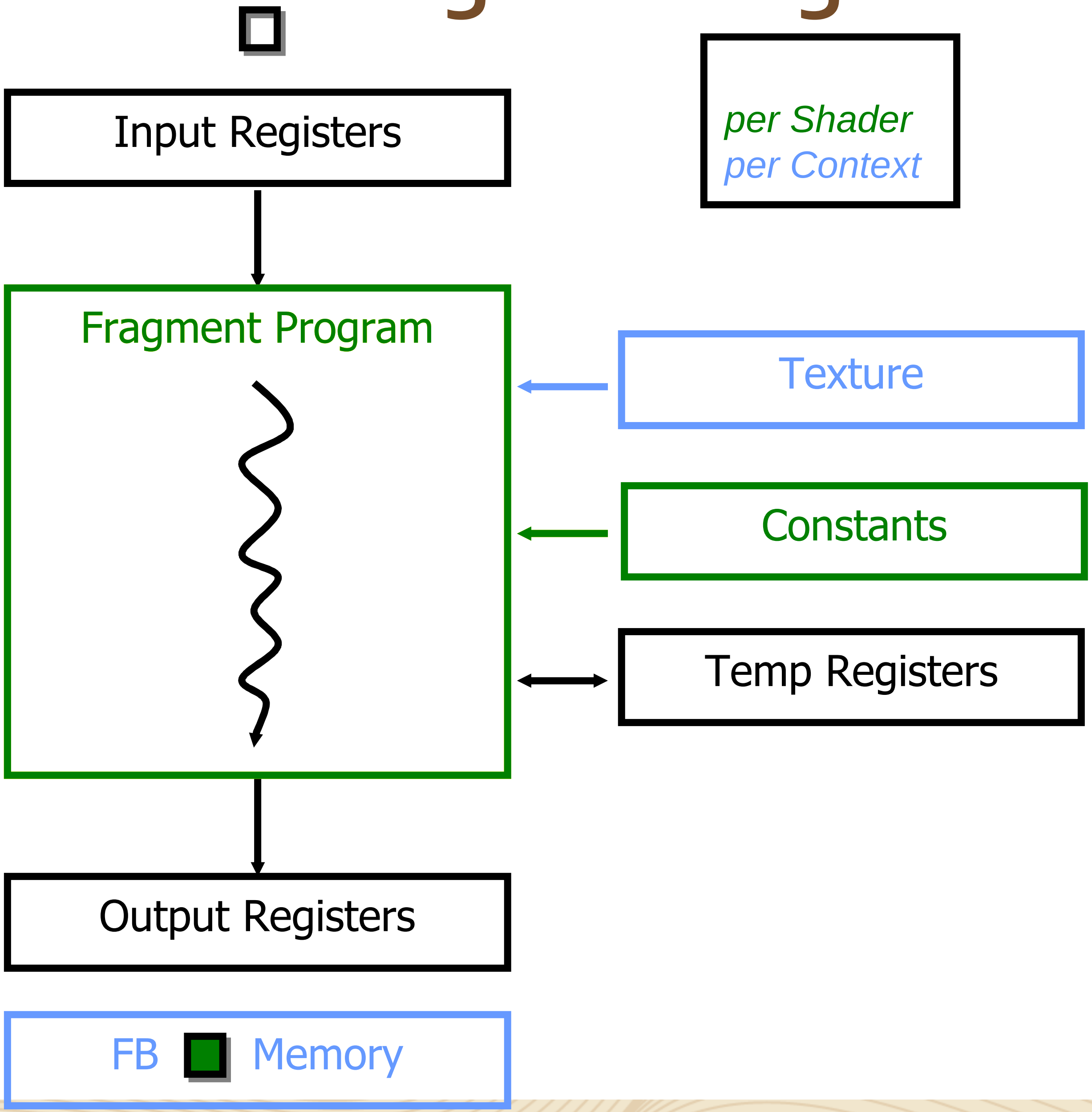
What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism (vector processing)
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

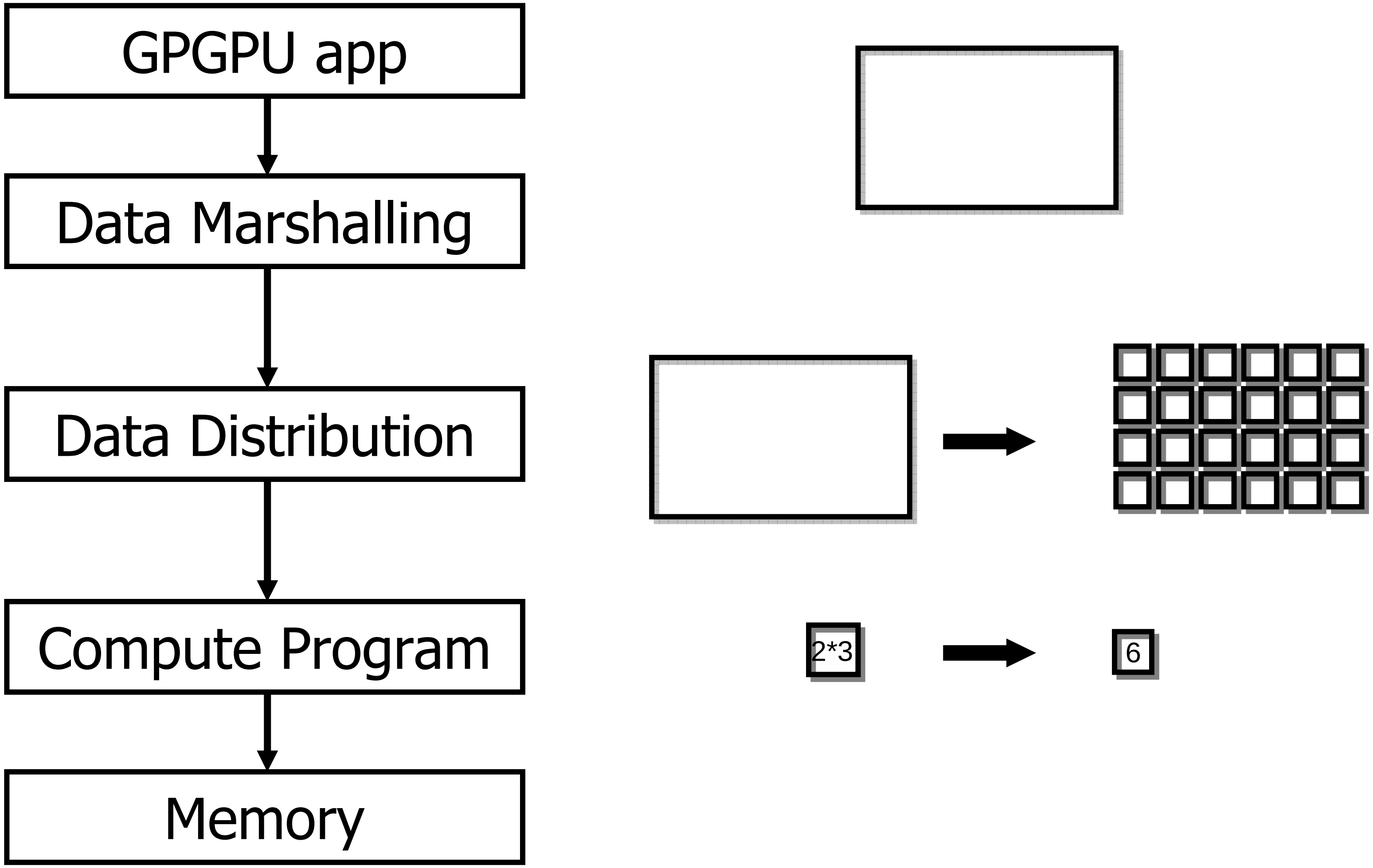
Graphics Programming Model



Pixel Shader Programming Model

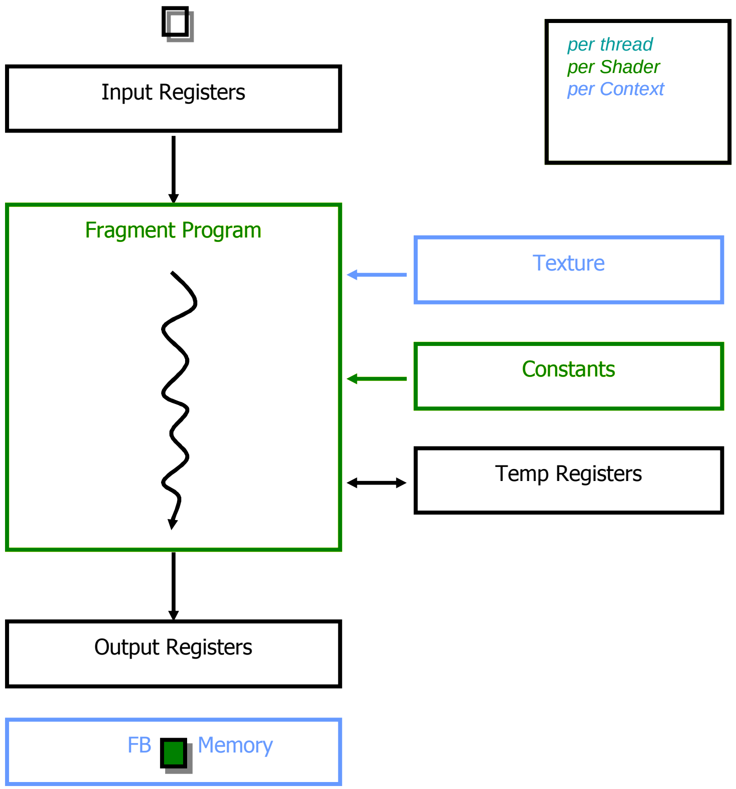


GPGPU Programming Model



Previous GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - **Limited outputs**
- Instruction sets
  - Lack of Integer & bit ops
- **Communication limited**
  - Between pixels
  - Scatter a[i] = p



CUDA

Programming Model

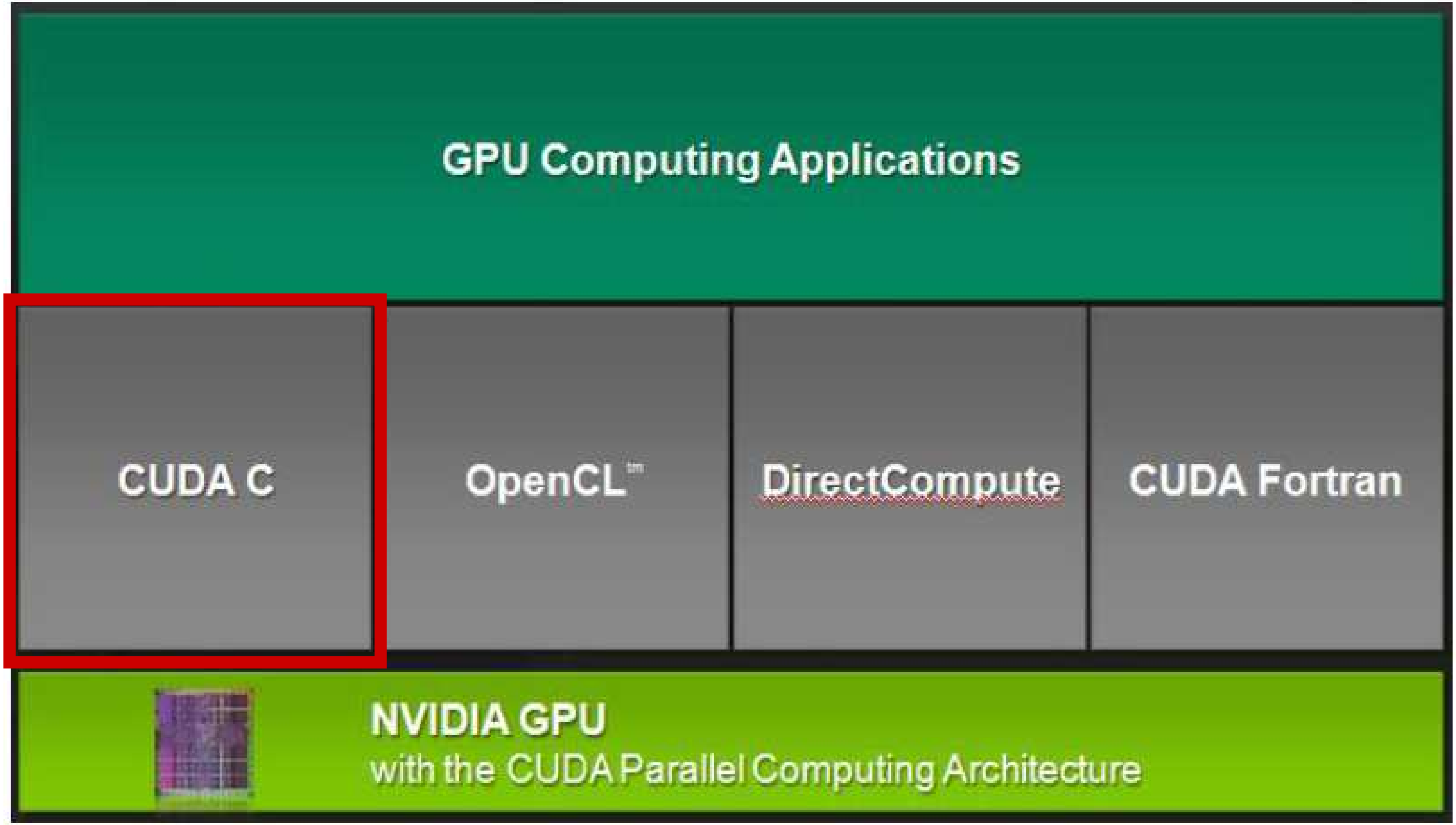
CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & read-back speeds
  - Explicit GPU memory management

CUDA Programming Model

- A highly multithreaded coprocessor
- The GPU is viewed as a compute device that
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of application are executed on device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Where is CUDA?



What is CUDA C?

- CUDA C extends C++
  - Recursion-free subset of C++ (for *kernels*)
  - Extensions for code that runs on device
- Define special functions, called *kernels*
  - Executed N times in parallel by N different *CUDA threads.*

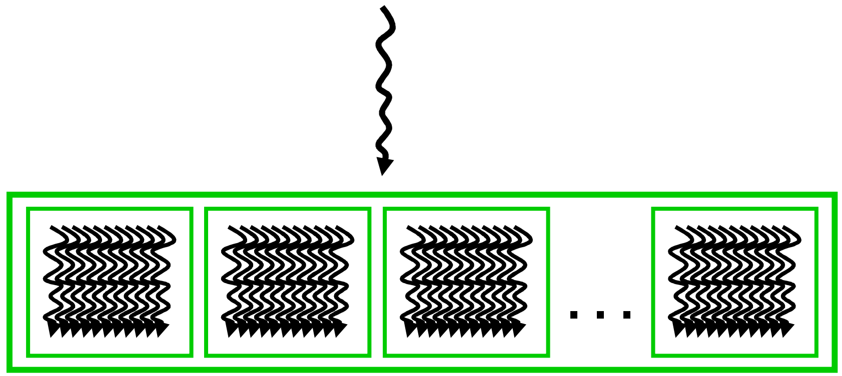
CUDA – C without Shader Limitations!

- Integrated host + device app C++ program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

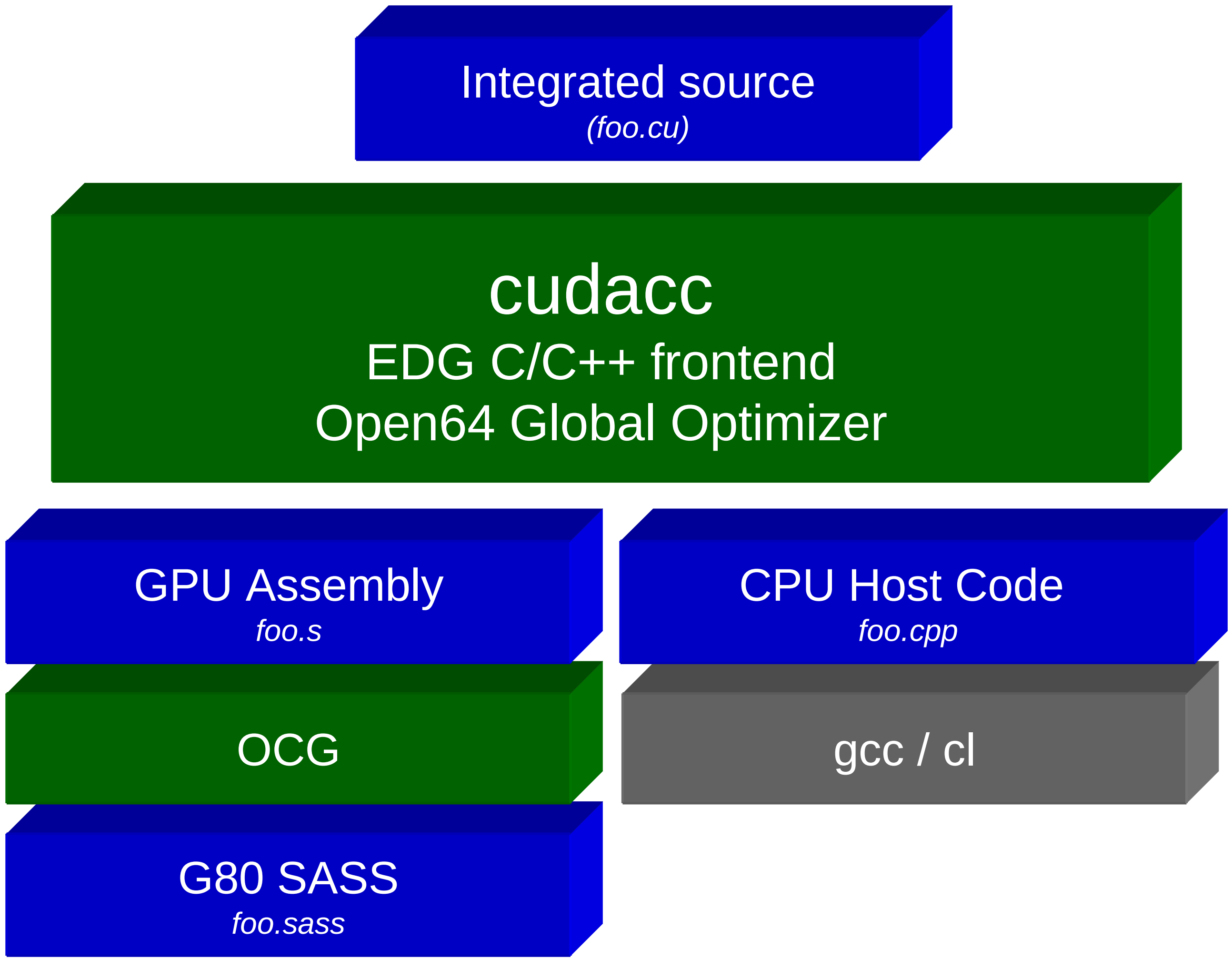
Serial Code (host)

Parallel Kernel (device)

`KernelA<<< nBlk, nTid >>>(args);`



Compilation



Extended C

- **Type Qualifiers**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image) {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  ...
  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Function Declarations

	executed on	callable from
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__device__` and `__host__` can be used together
- `__device__` functions: no address operator
- `__global__` defines a kernel function
  - Must return `void`

Function Declarations (cont.)

- `__global__`
  - Must specify its execution configuration (details later)
  - Asynchronous
- Functions executed on the device
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
  - No indirect function calls

Kernel: Sum of Vectors

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

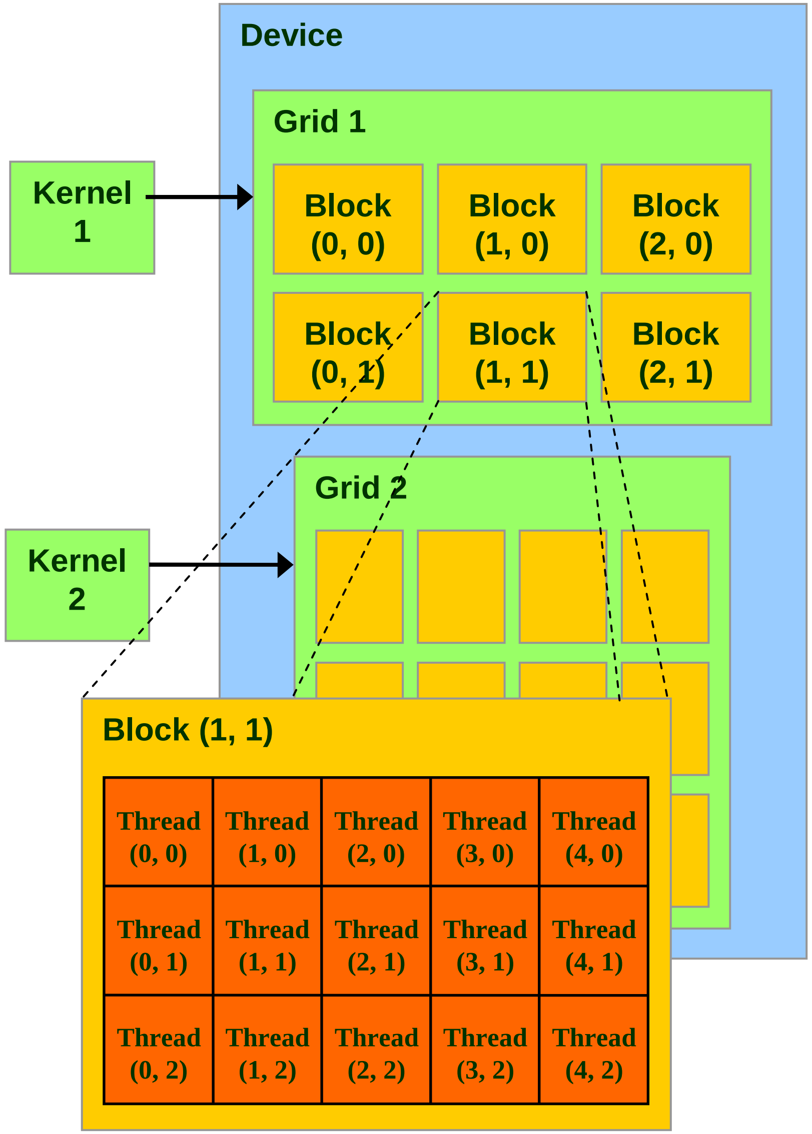
runs on device (GPU)

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

runs on host (CPU)

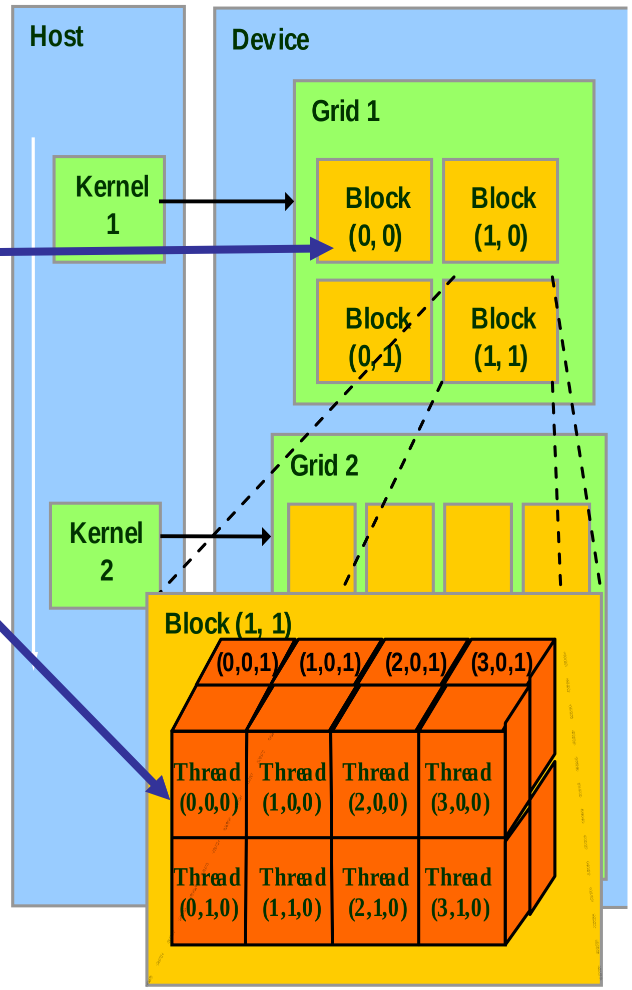
Thread Hierarchy

- <= 512(1024) threads/block
- 1D, 2D, 3D(only threads)
- Blocks executed in independent order; serial or parallel
- Threads within a block can cooperate by sharing data through *shared memory*
- **__syncthreads():** wait-barrier for all threads in a block



Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration

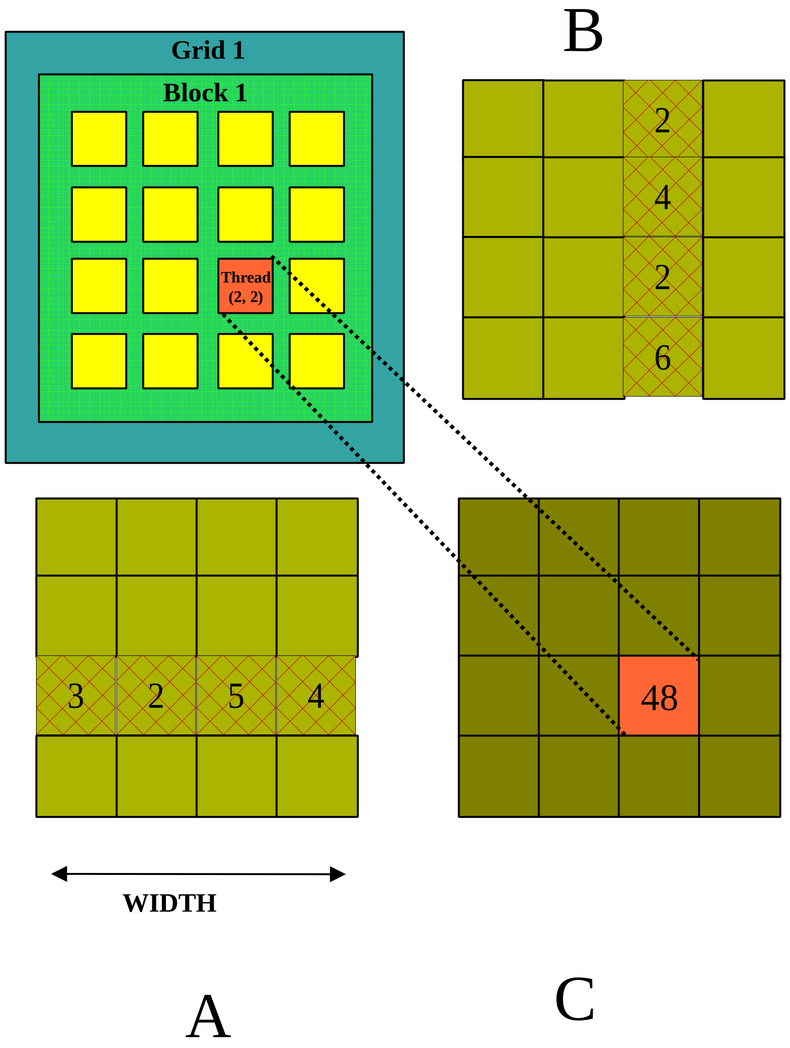
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```

Kernel: Sum of Matrices

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Only One Thread Block Used

- One Block of threads compute matrix C
  - Each thread computes one element of C
- Each thread
  - Loads a row of matrix A
  - Loads a column of matrix B
  - Perform one multiply and addition for each pair of A and B elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block

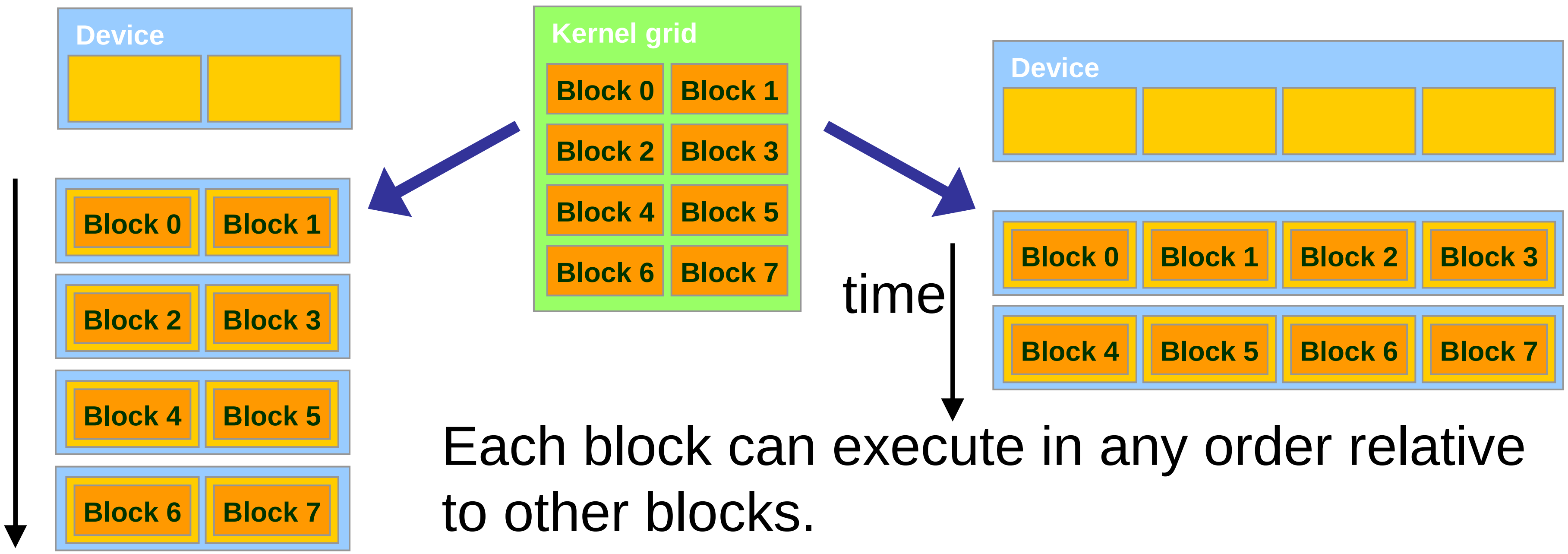


Kernel: Sum of Matrices MK2

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

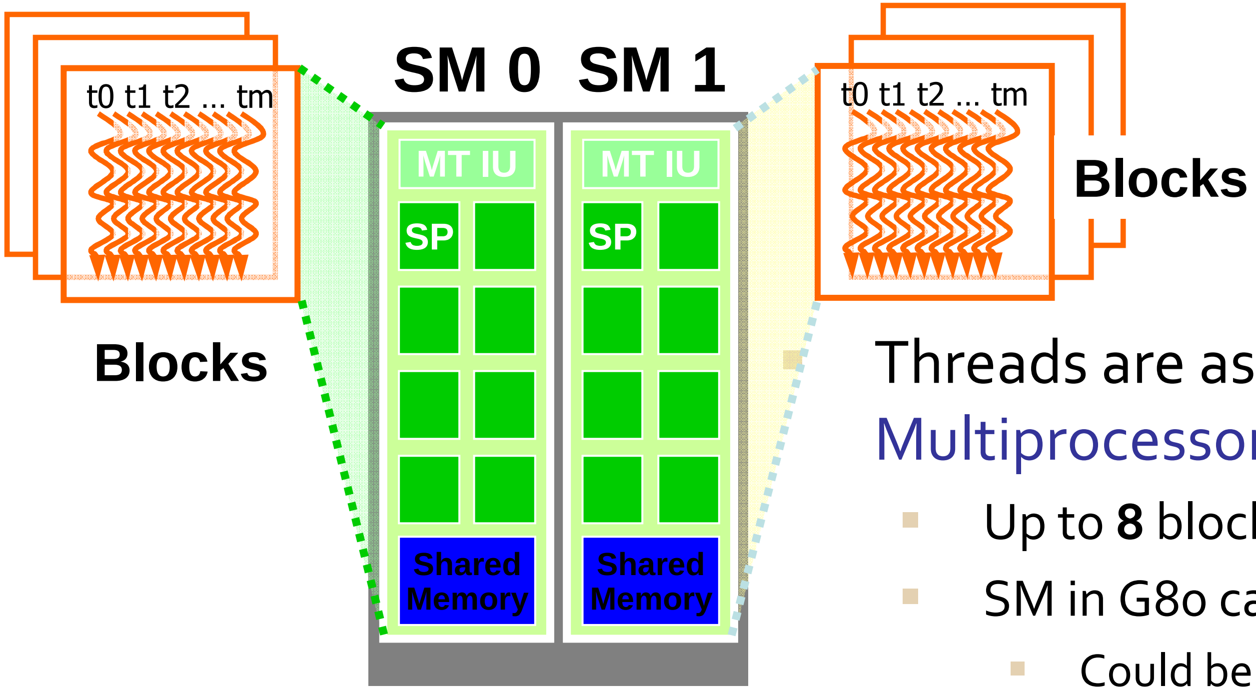
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Each block can execute in any order relative to other blocks.

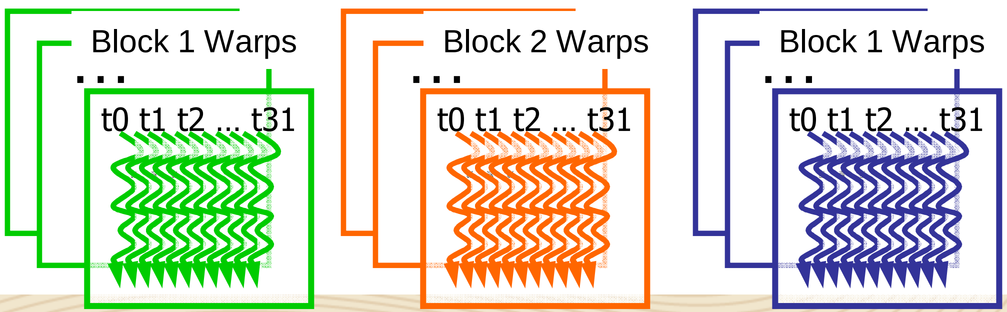
G80 Example: Executing Thread Blocks

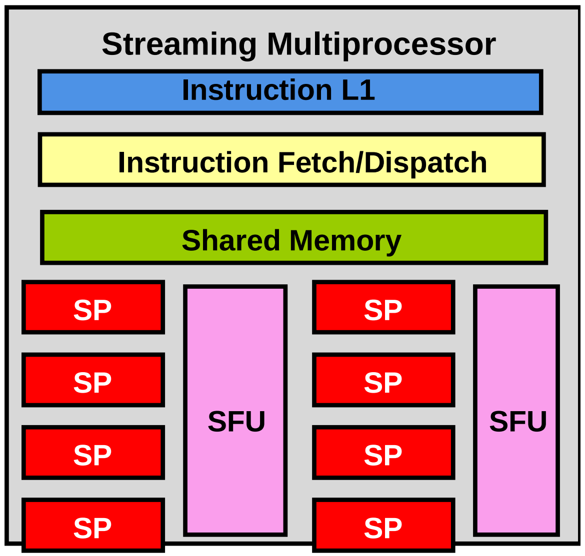


- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks
    - …
- Threads run concurrently
  - SM maintains thread/block ids
  - SM manages/schedules thread execution

G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - HW decision, not part of the CUDA programming model!
  - Warps are scheduling units in SM (only **one** warp is executed)
  - All threads in a warp execute the same instruction (branching!)
- 3 blocks, each 256 threads are assigned to an SM; How many Warps are there in an SM?
  - Each Block has 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps





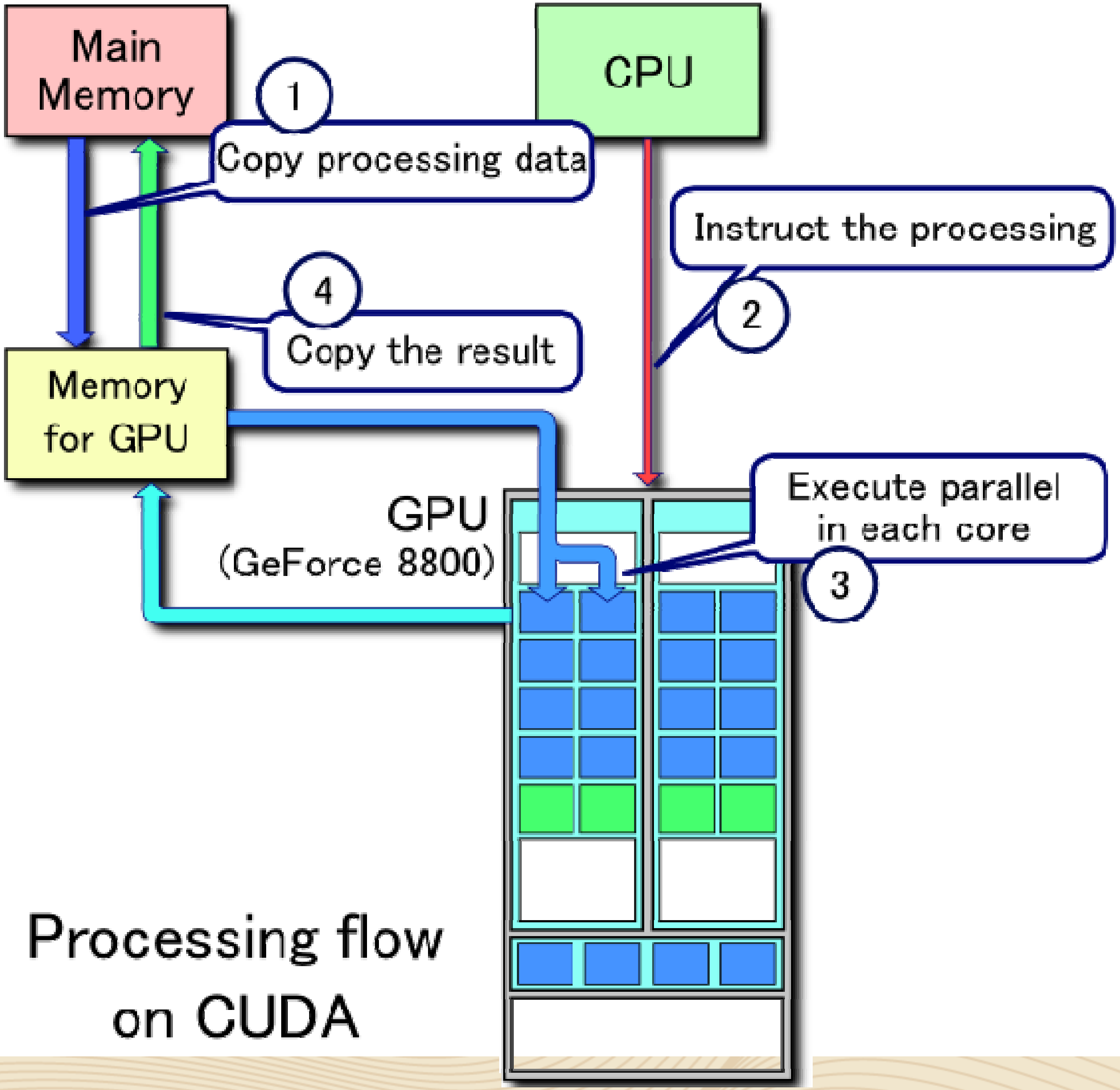
G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM! Wastes 1/3.
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule. Sweet Spot.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM! Will not run on G80. (Fermi can)

Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths must be serialized
- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - Branch granularity < warp size
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Branch granularity is a whole multiple of warp size

Processing Flow On CUDA



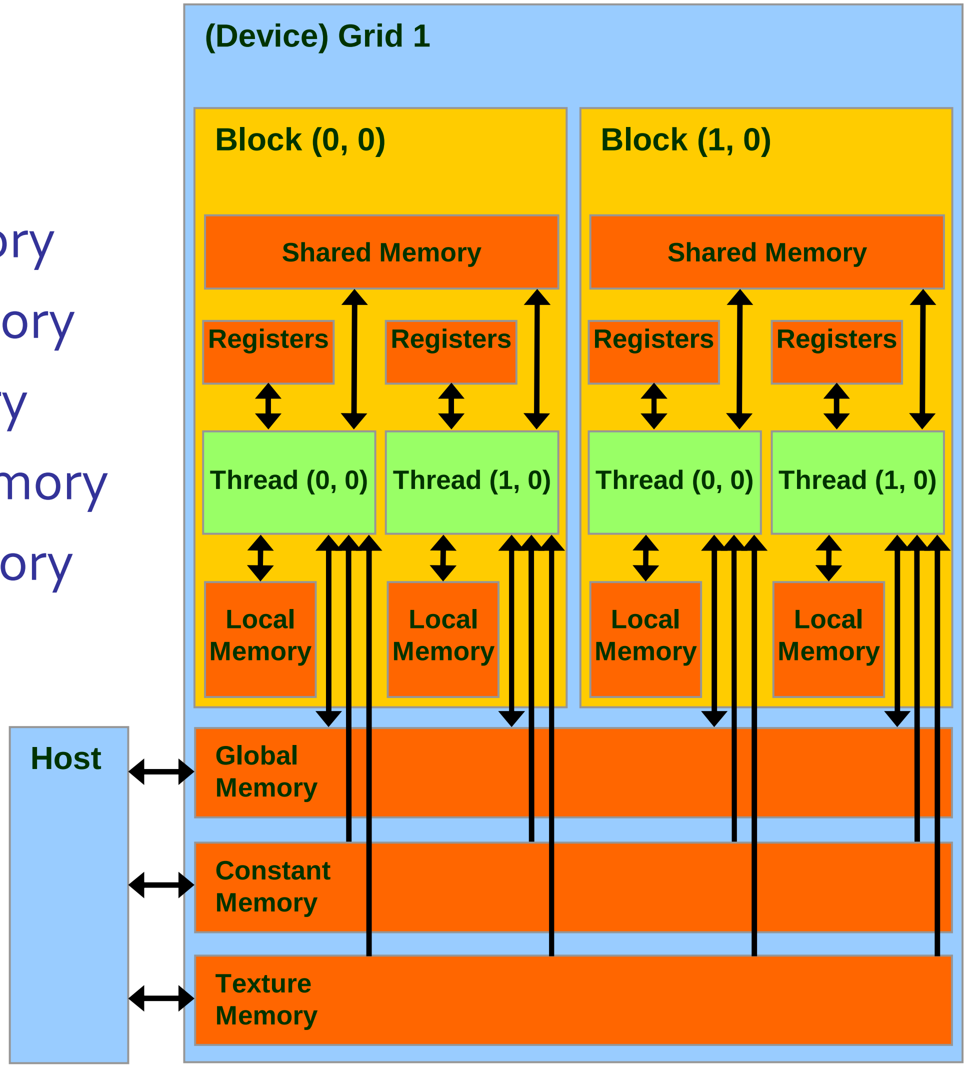
Processing flow on CUDA

CUDA

Memory Model

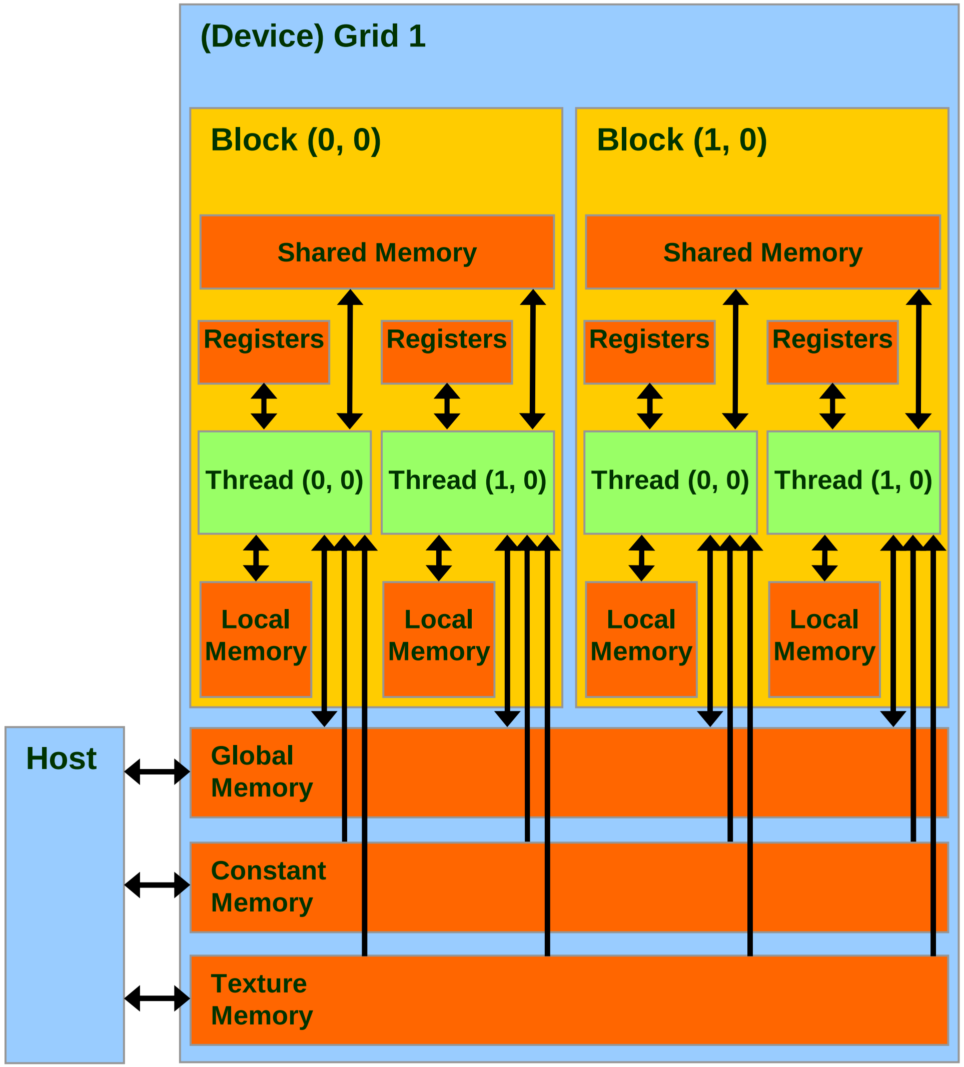
Device Memory Space Overview

- Each thread can
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read per-grid constant memory
  - Read per-grid texture memory
- The host can R/W
  - global,
  - constant
  - texture memories



Device Memory Space Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Not cached
- Texture and Constant memories
  - Constants initialized by host
  - Contents visible to all threads
  - Cached



Device Global Memory Allocation

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);
```

Device Global Memory Transfer

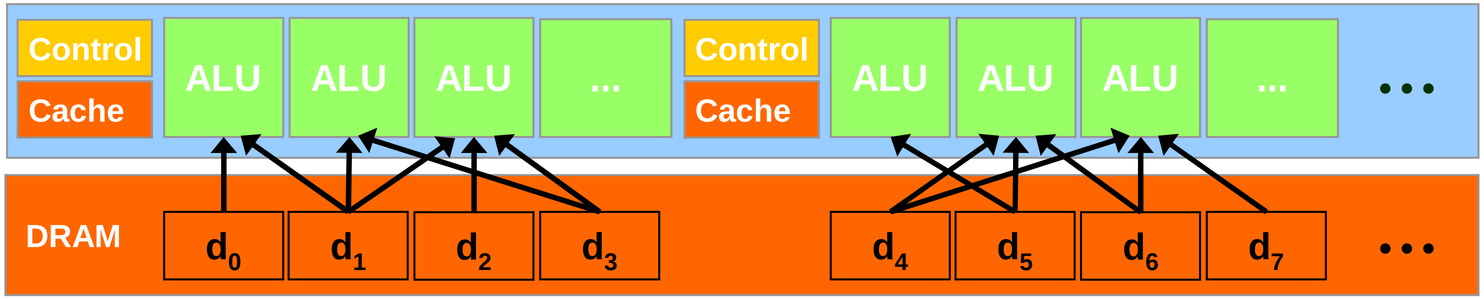
```
// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// Free host memory
...
}
```

Device Copy To Constant Memory

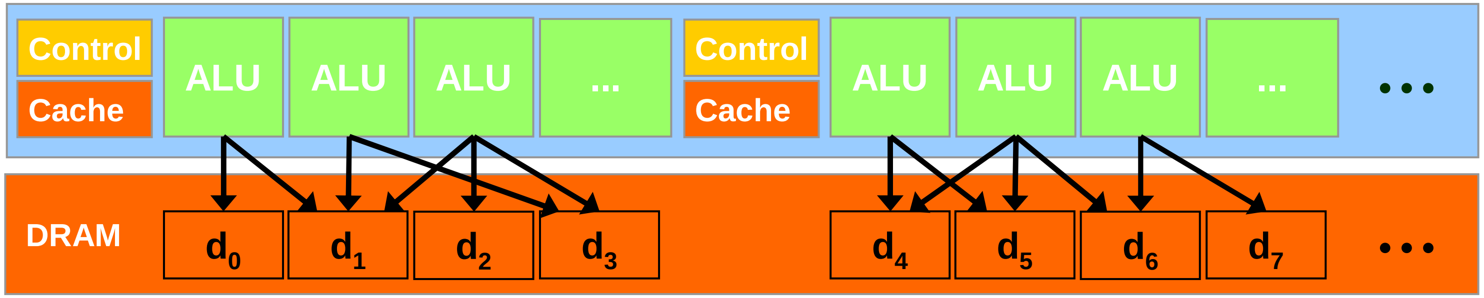
```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

CUDA Highlights: Scatter

- CUDA provides generic DRAM memory addressing
  - Gather:



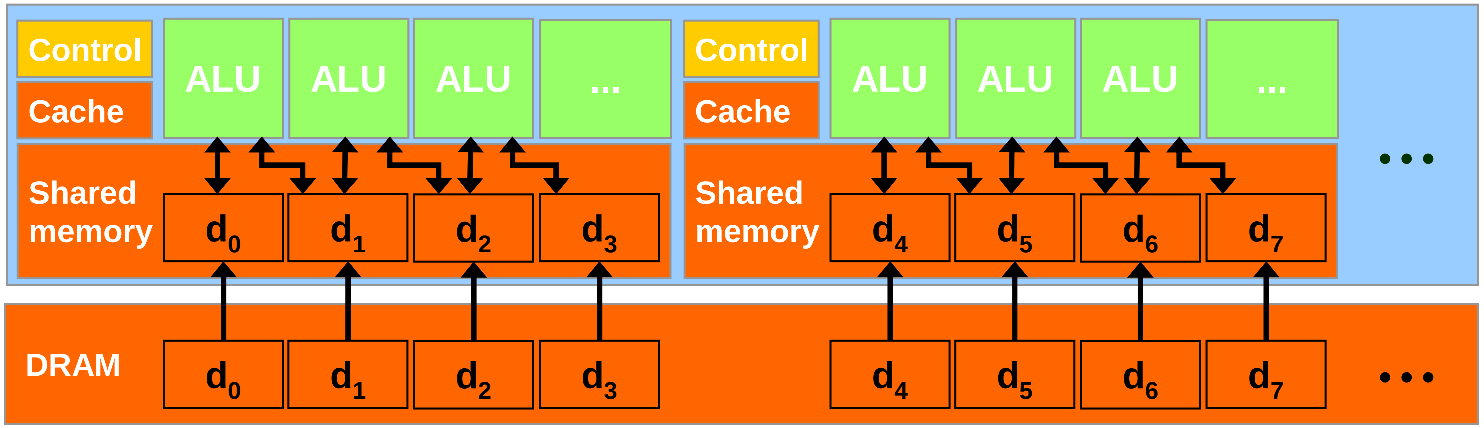
  - And scatter: no longer limited to write one pixel



→ More programming flexibility

CUDA Highlights: Shared Memory

- On-chip (very fast access)
- Efficient data sharing between threads of a block



→ Big memory bandwidth savings

Access Times

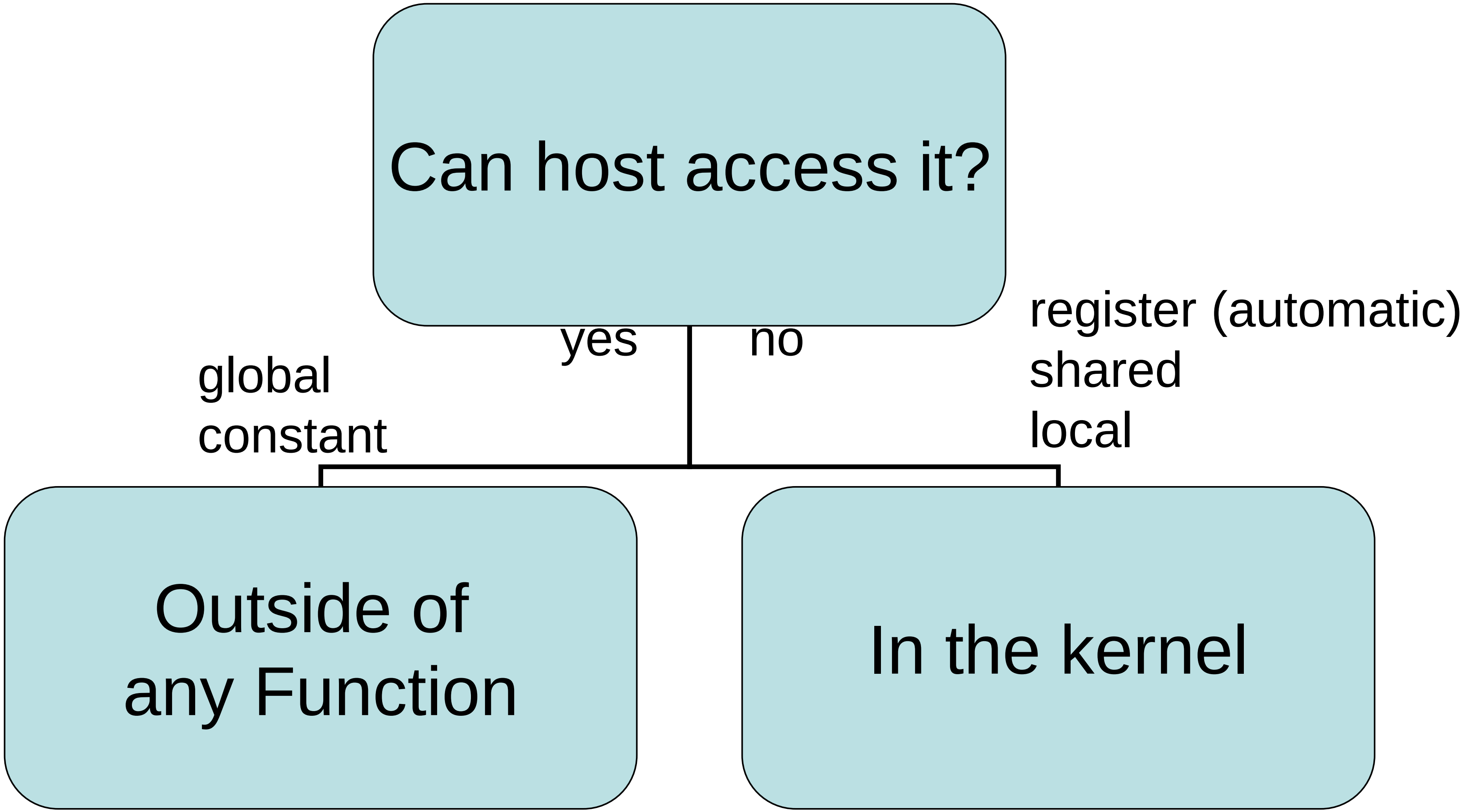
- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - **single cycle**
- Local Memory – DRAM, no cache - ***slow***
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality, symbol data
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality, array type data
- Instruction Memory (invisible) – DRAM, cached

Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
  - Registers spill to local memory

Where to Declare Variables?



Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

    ```
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable:

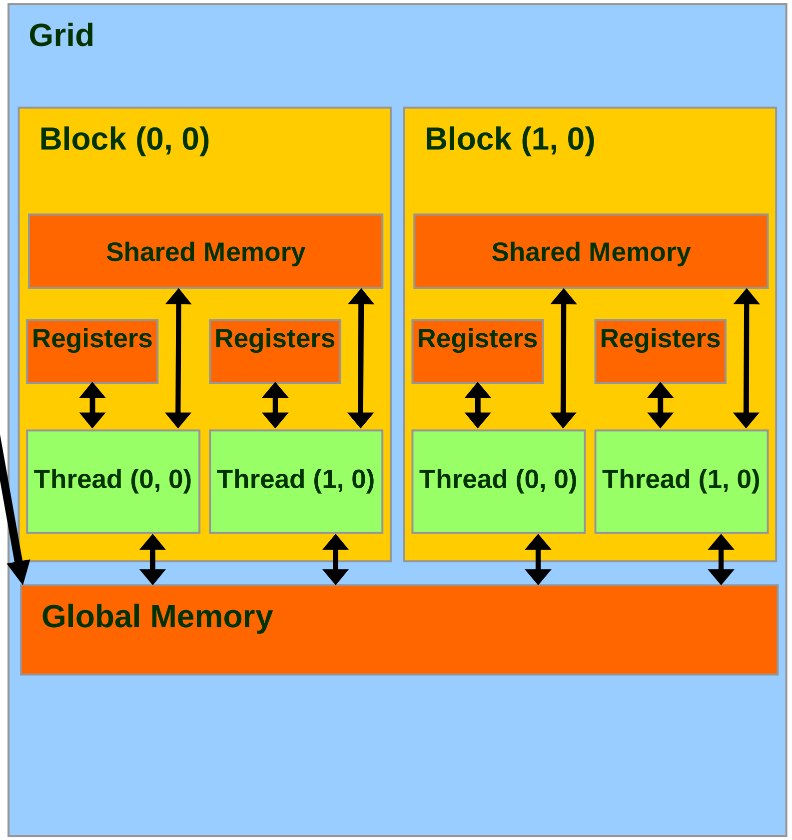
    ```
    float* ptr = &GlobalVar;
    ```

Example: Matrix Mul

```
global__ void MatrixMulKernel(float* A, float* B,
  float* C, intWidth)
{
// Calculate the row index of the C element and A
int Row = blockId.y * TILE_WIDTH + threadId.y;
// Calculate the column idenx of C and B
Int Col = blockId.x * TILE_WIDTH + threadId.x;
Cvalue = 0;
// each thread computes one element of block sub-matrix
for (int k = 0; k < Width; ++k)
  Cvalue += A[Row][k] * B[k][Col];
C[Row][Col] = Cvalue;
}
```

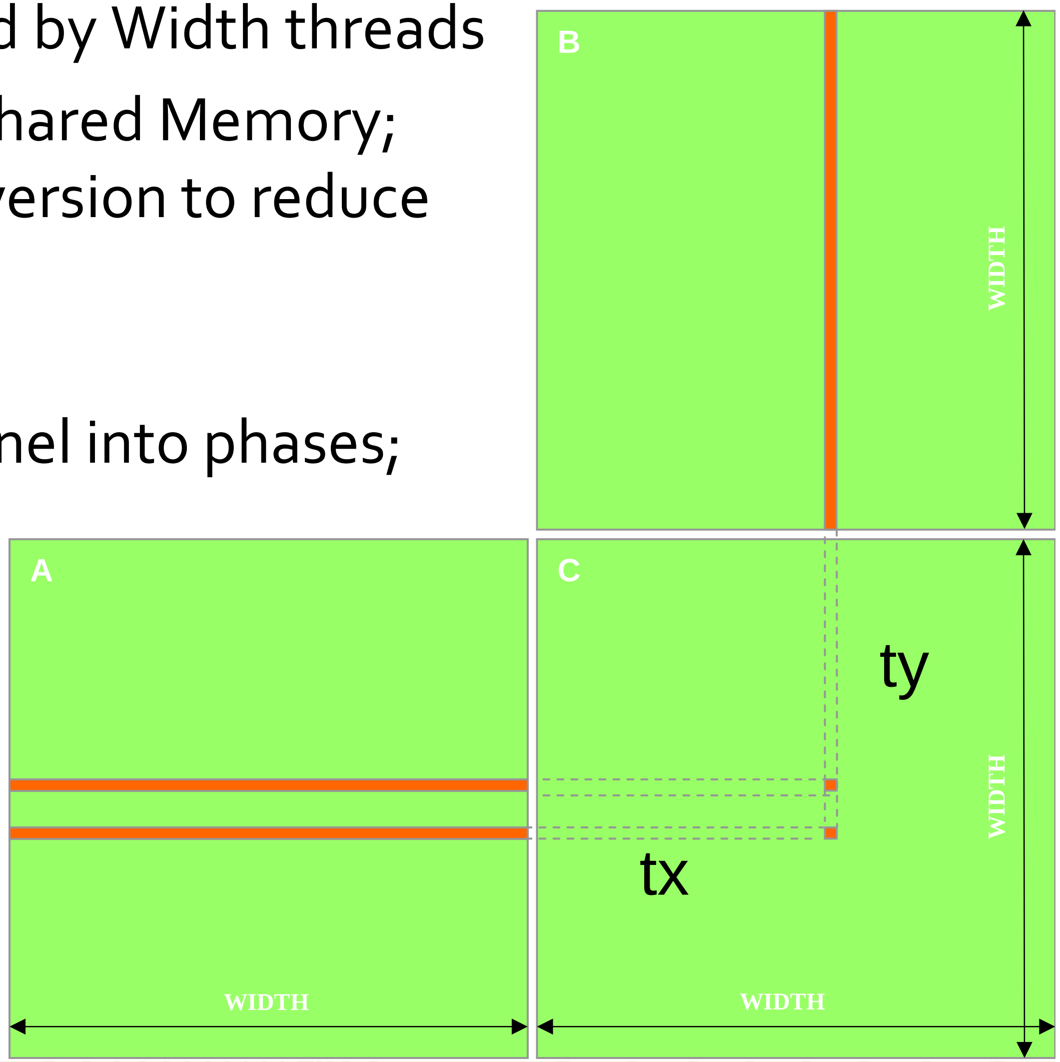
How About Performance on G80?

- All threads access global memory for input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 86.4 GB/s global memory access bandwidth limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS

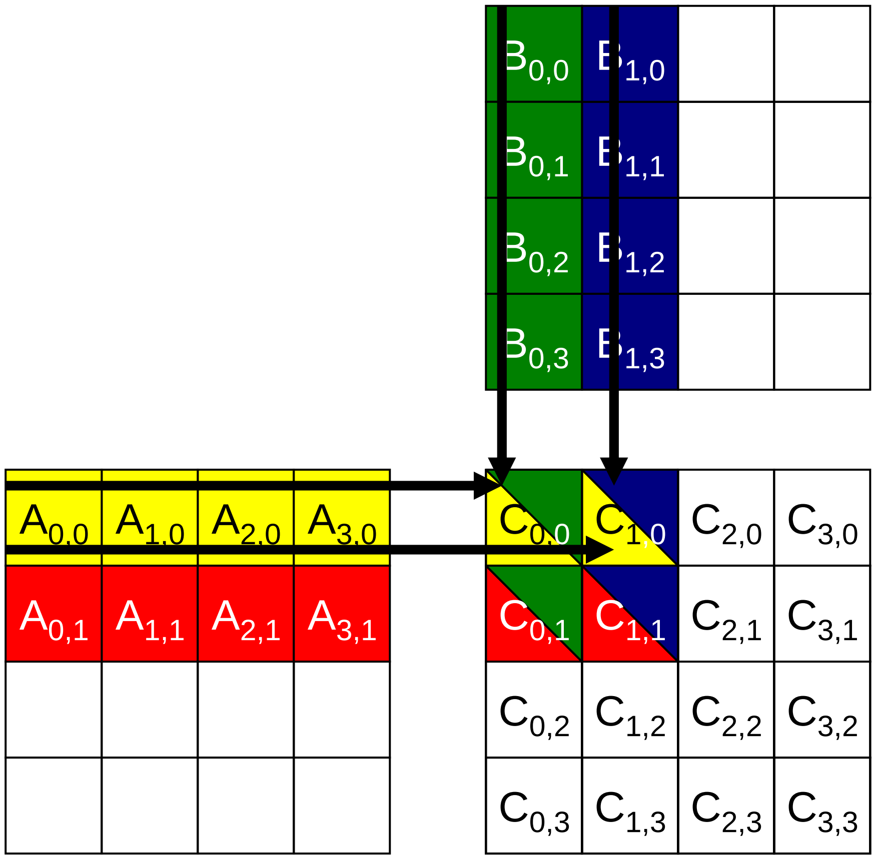


Idea: Use Shared Memory for Reuse

- Each input element is read by Width threads
- Load each element into Shared Memory; several threads use local version to reduce memory bandwidth
  - Tiled algorithms
- Break up execution of kernel into phases; data accesses in each phase is focused on one subset (tile) of A and B



Example: Using 2x2 Block; No Tiling in A and B



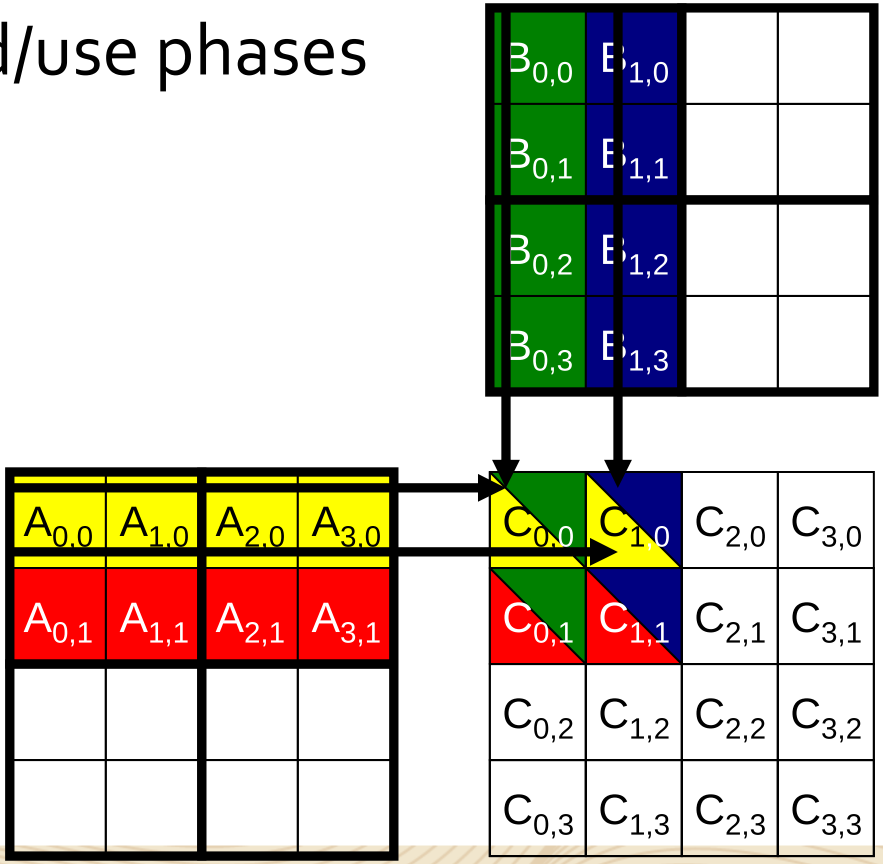
Every A and B Element is Used Exactly Twice in generating a 2X2 tile of C

Access order

$C_{0,0}$ $\text{thread}_{0,0}$	$C_{1,0}$ $\text{thread}_{1,0}$	$C_{0,1}$ $\text{thread}_{0,1}$	$C_{1,1}$ $\text{thread}_{1,1}$
$A_{0,0} * B_{0,0}$	$A_{0,0} * B_{1,0}$	$A_{0,1} * B_{0,0}$	$A_{0,1} * B_{1,0}$
$A_{1,0} * B_{0,1}$	$A_{1,0} * B_{1,1}$	$A_{1,1} * B_{0,1}$	$A_{1,1} * B_{1,1}$
$A_{2,0} * B_{0,2}$	$A_{2,0} * B_{1,2}$	$A_{2,1} * B_{0,2}$	$A_{2,1} * B_{1,2}$
$A_{3,0} * B_{0,3}$	$A_{3,0} * B_{1,3}$	$A_{3,1} * B_{0,3}$	$A_{3,1} * B_{1,3}$

Colaboration of Threads

- Goal: reduce traffic to global memory
- Idea: collaboratively load A and B tiles before use
- Introduce load/use phases



Each Phase of a Thread Block Uses One Tile from A and One from B

Threads	Phase 1			Phase 2		…
$T_{0,0}$	$\mathbf{A_{0,0}}$ $\downarrow As_{0,0}$	$\mathbf{B_{0,0}}$ $\downarrow$ $Bs_{0,0}$	$CValue_{0,0}$ += $As_{0,0}$*$Bs_{0,0}$ + $As_{1,0}$*$Bs_{0,1}$	$\mathbf{A_{2,0}}$ $\downarrow$ $As_{0,0}$	$\mathbf{B_{0,2}}$ $\downarrow$ $Bs_{0,0}$	$CValue_{0,0}$ += $As_{0,0}$*$Bs_{0,0}$ + $As_{1,0}$*$Bs_{0,1}$
$T_{1,0}$	$\mathbf{A_{1,0}}$ $\downarrow As_{1,0}$	$\mathbf{B_{1,0}}$ $\downarrow$ $Bs_{1,0}$	$CValue_{1,0}$ += $As_{0,0}$*$Bs_{1,0}$ + $As_{1,0}$*$Bs_{1,1}$	$\mathbf{A_{3,0}}$ $\downarrow$ $As_{1,0}$	$\mathbf{B_{1,2}}$ $\downarrow$ $Bs_{1,0}$	$CValue_{1,0}$ += $As_{0,0}$*$Bs_{1,0}$ + $As_{1,0}$*$Bs_{1,1}$
$T_{0,1}$	$\mathbf{A_{0,1}}$ $\downarrow As_{0,1}$	$\mathbf{B_{0,1}}$ $\downarrow$ $Bs_{0,1}$	$CValue_{0,1}$ += $As_{0,1}$*$Bs_{0,0}$ + $As_{1,1}$*$Bs_{0,1}$	$\mathbf{A_{2,1}}$ $\downarrow$ $As_{0'1}$	$\mathbf{B_{0,3}}$ $\downarrow$ $Bs_{0,1}$	$CValue_{0,1}$ += $As_{0,1}$*$Bs_{0,0}$ + $As_{1,1}$*$Bs_{0,1}$
$T_{1,1}$	$\mathbf{A_{1,1}}$ $\downarrow As_{1,1}$	$\mathbf{B_{1,1}}$ $\downarrow$ $Bs_{1,1}$	$CValue_{1,1}$ += $As_{0,1}$*$Bs_{1,0}$ + $As_{1,1}$*$Bs_{1,1}$	$\mathbf{A_{3,1}}$ $\downarrow$ $As_{1,1}$	$\mathbf{B_{1,3}}$ $\downarrow$ $Bs_{1,1}$	$CValue_{1,1}$ += $As_{0,1}$*$Bs_{1,0}$ + $As_{1,1}$*$Bs_{1,1}$

time $\longrightarrow$

Results

- Multiple lookups are satisified from shared memory
- For NxN tiles reduces the number of accesses to the global memory by factor N
- Introduces MATRIX_WIDTH / TILE_WIDTH phases for dot product calculation
- Each phase uses same shared memory locations
  - Small shared memory footprint
- Each phase focuses on small subset of input matrix
  - “Locality“

G80 Shared Memory and Block Size

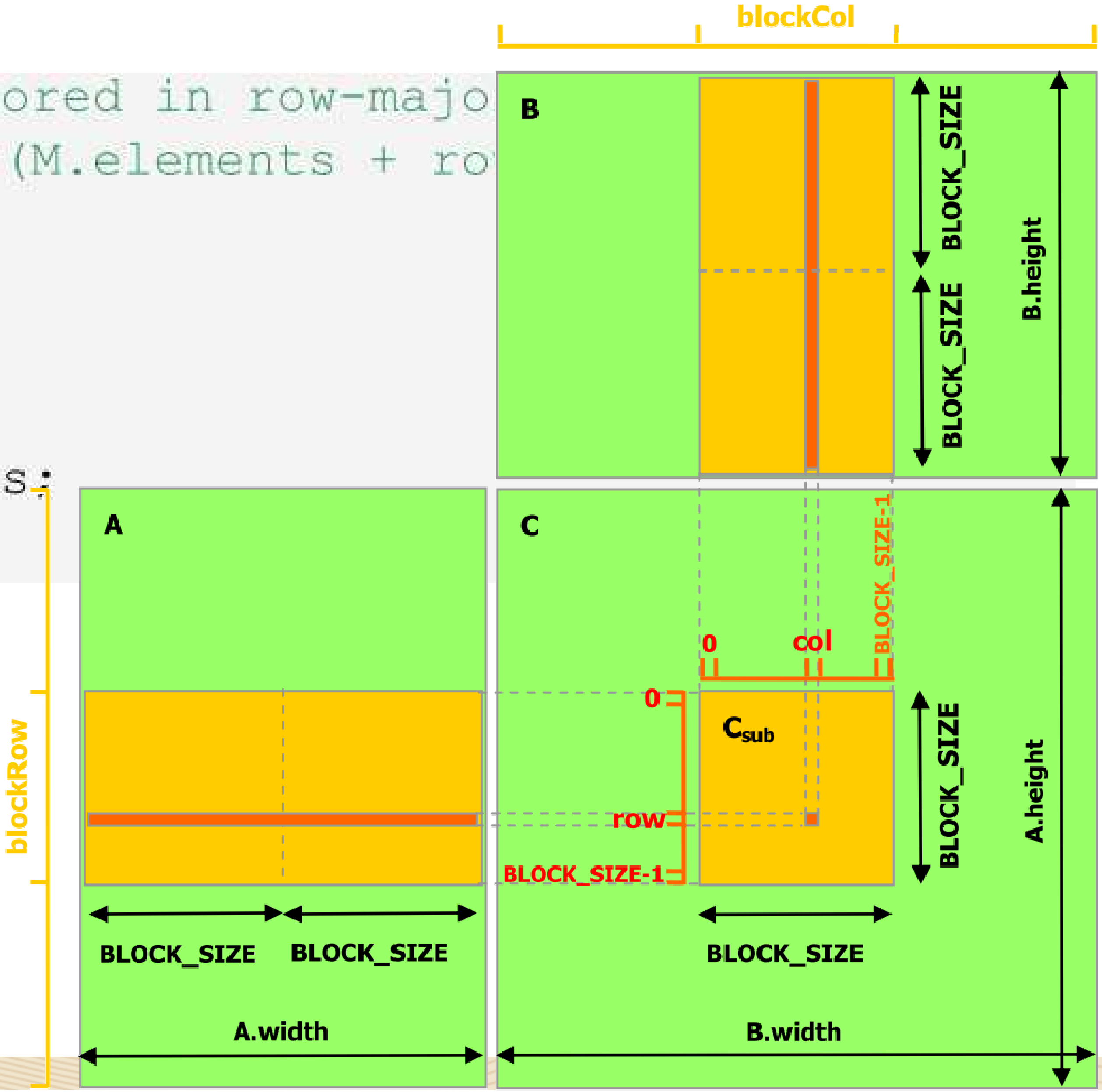
- G80 has 16KB shared memory per streaming multiprocessor (SM)
  - TILE_WIDTH = 16, → 2*256*4B = 2KB of shared memory per thread block → up to 8 thread blocks active at the same time
  - TILE_WIDTH = 32 → 2*32*32*4B= 8KB shared memory per thread block → up to two thread blocks active at the same time (→ scheduling flexibility reduced)
- 16x16 tiling reduces access to global memory by factor 16
  - 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

First-order Size Considerations in G80

- 16x16 block → each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations → 16:1 ratio

Using Shared Memory: Tiled Matrix Mul.

```c
// Matrices are stored in row-majo
// M(row, col) = *(M.elements + ro
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
```



Using Shared Memory: Tiled Matrix Mul.

```
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}
```

Using Shared Memory: Tiled Matrix Mul.

```
// Thread block size
#define BLOCK_SIZE 16

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}
```

Using Shared Memory: Tiled Matrix Mul.

```
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// 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((void**)&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((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
```

Using Shared Memory: Tiled Matrix Mul.

```
    // 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((void**)&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);
}
```

Using Shared Memory: Tiled Matrix Mul.

```
// 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;
```

Using Shared Memory: Tiled Matrix Mul.

```
// 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
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);
```

Using Shared Memory: Tiled Matrix Mul.

```
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // 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);
}
```

A Common Programming Strategy

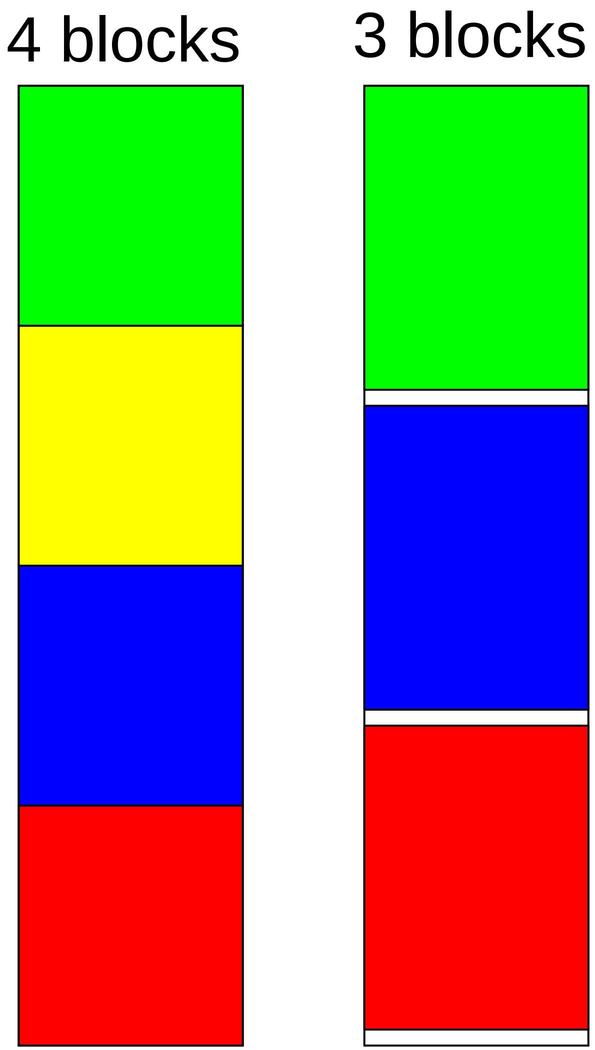
- Global memory has slower access than shared memory
- Tile data to take advantage of fast shared memory
  - Partition data into subsets that fit into shared memory
- Handle each data subset with one thread block
  - Loading subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
  - Perform computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
  - Tiles should be independent
  - Copy results from shared memory to global memory

A Common Programming Strategy

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only no structure → constant memory (fast if in cache)
  - R/Only array structure → texture memory (fast if in cache)
  - R/W shared within block → shared memory (fast)
  - R/W within each thread → registers (fast)
  - R/W inputs/results → global memory (very slow)

Programmer View of Register File

- There are 8192 32bit registers in each SM in G80
  - HW dependent, not part of CUDA
  - Registers are dynamically partitioned across all blocks assigned to the SM (compiler)
  - Once assigned to a block, register is NOT accessible by threads in other blocks
  - Each thread in same block only access registers assigned to itself



Register Usage

- If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each block requires 10*256 = 2560 registers
  - 8192 / 2560 = **3**
  - Three blocks can run in parallel on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 / 2816 = 2
  - Only two Blocks can run on an SM,
    1/3 reduction of parallelism!

Constants

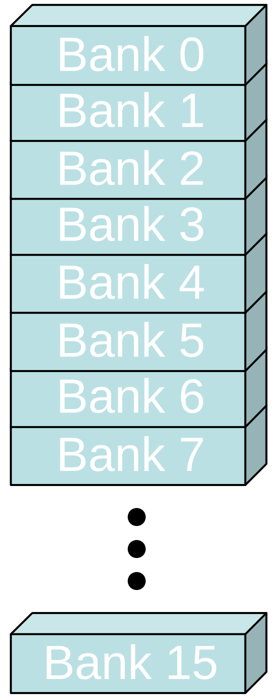
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM (8 KB)
- Constant value can be broadcast to all threads in a warp
  - Extremely efficient way of accessing a value that is common for all threads in a block!

Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other ☺

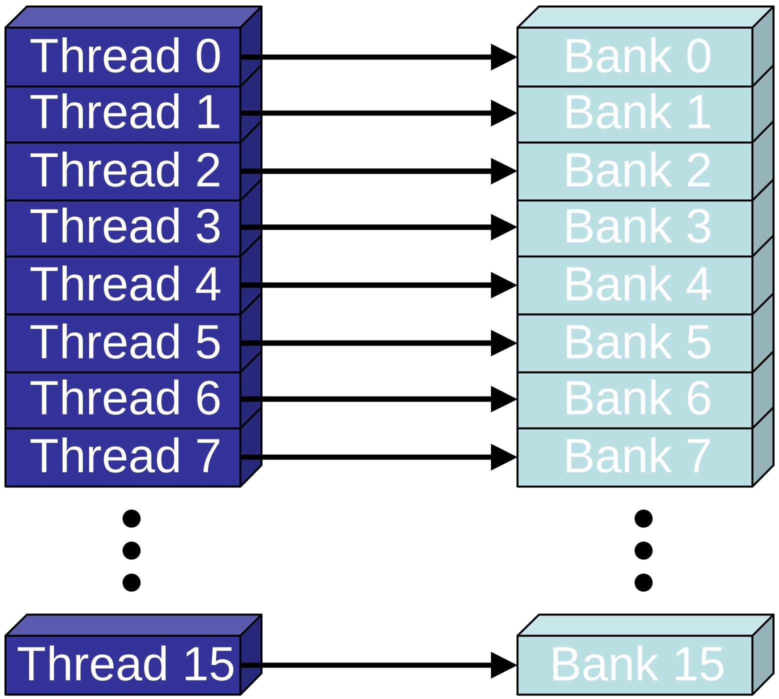
Parallel Memory Architecture

- In a parallel machine, many threads access (shared) memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

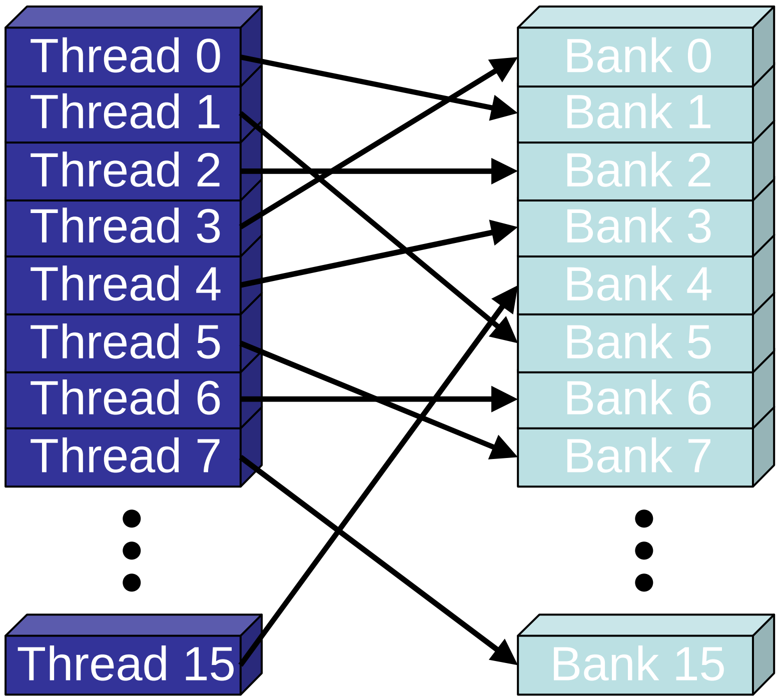


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

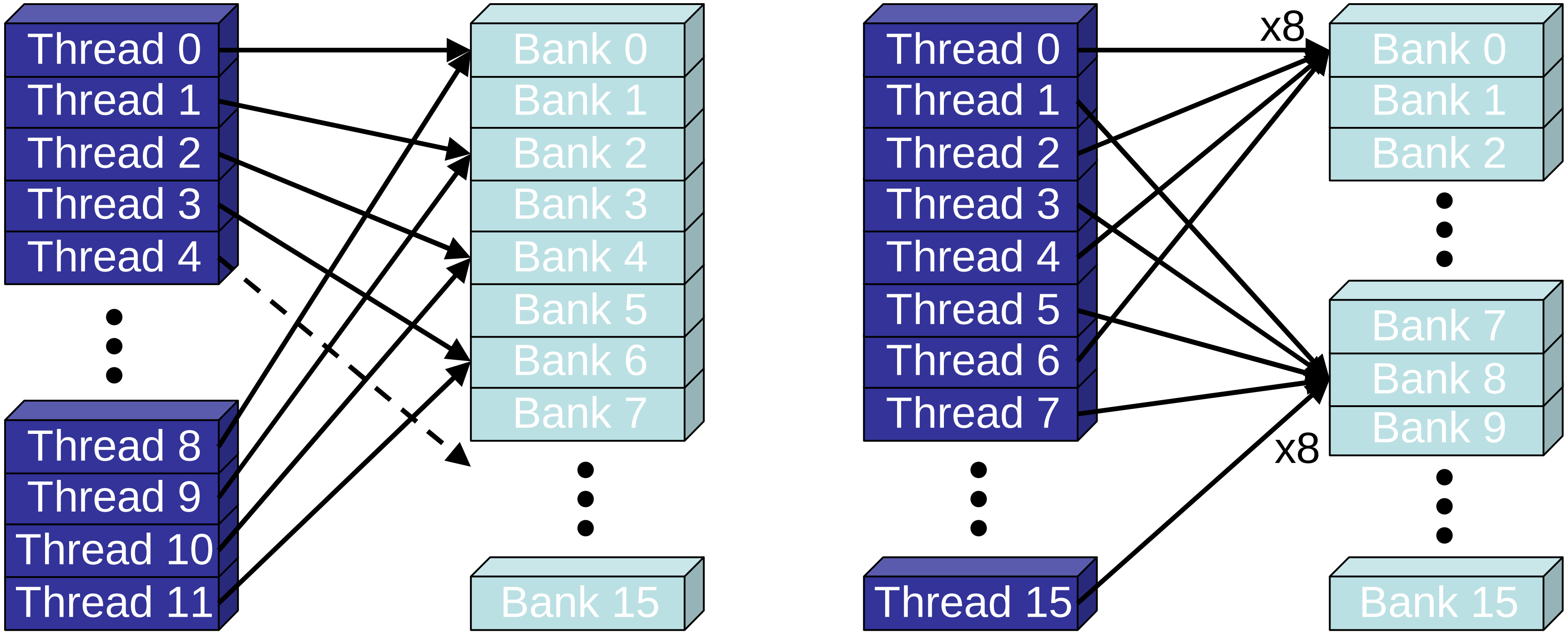


- No Bank Conflicts
  - Random 1:1 Permutation



Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2
- 8-way Bank Conflicts
  - Linear addressing stride == 8



How Addresses Map to Banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

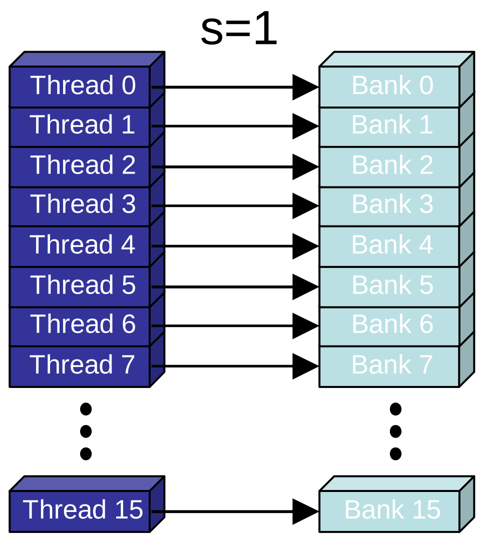
Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

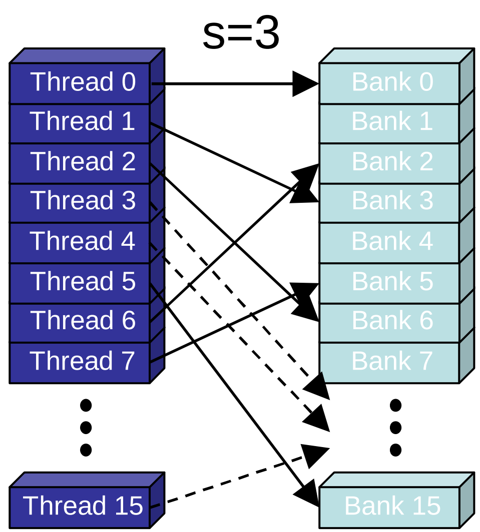
Linear Addressing

- Given:

```
__shared__ float shared[256];
float foo = shared[baseIndex + s
    * threadIdx.x];
```



- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so s must be odd



Parallel Reduction

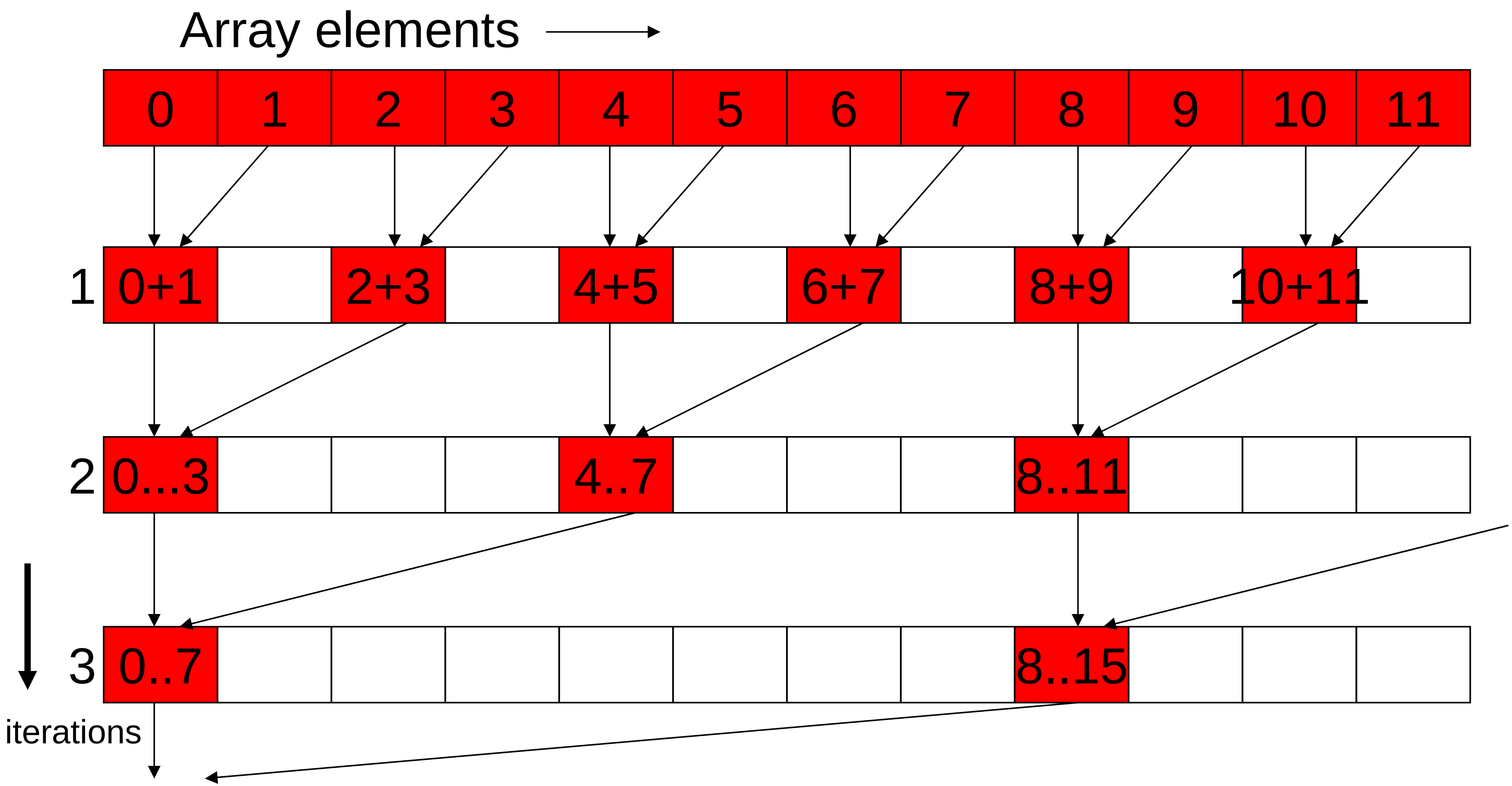
- Given an array of values, “reduce” them to a single value in parallel
- Examples
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array

- Typically parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

A Vector Reduction Example

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

Vector Reduction with Bank Conflicts

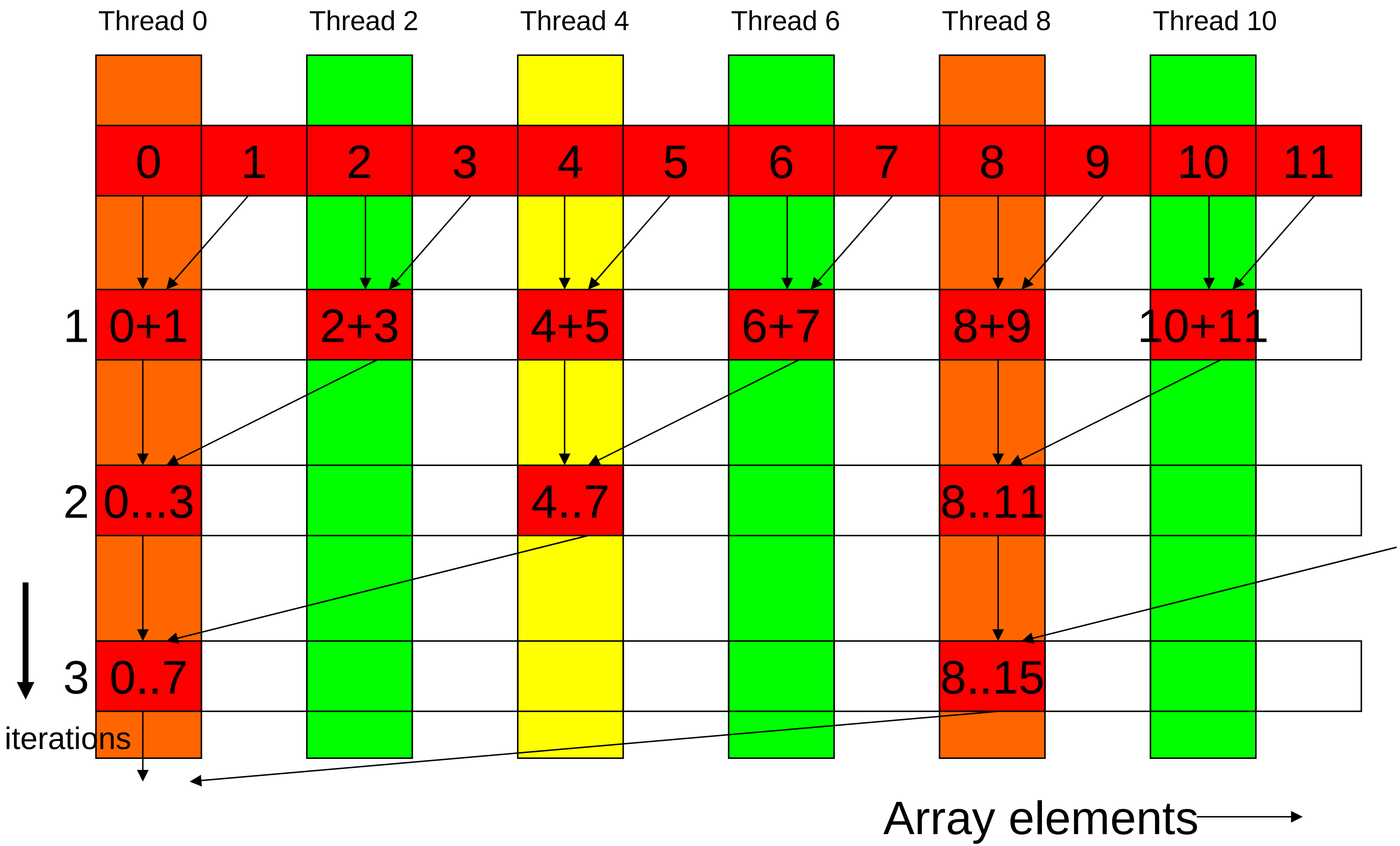


A simple implementation

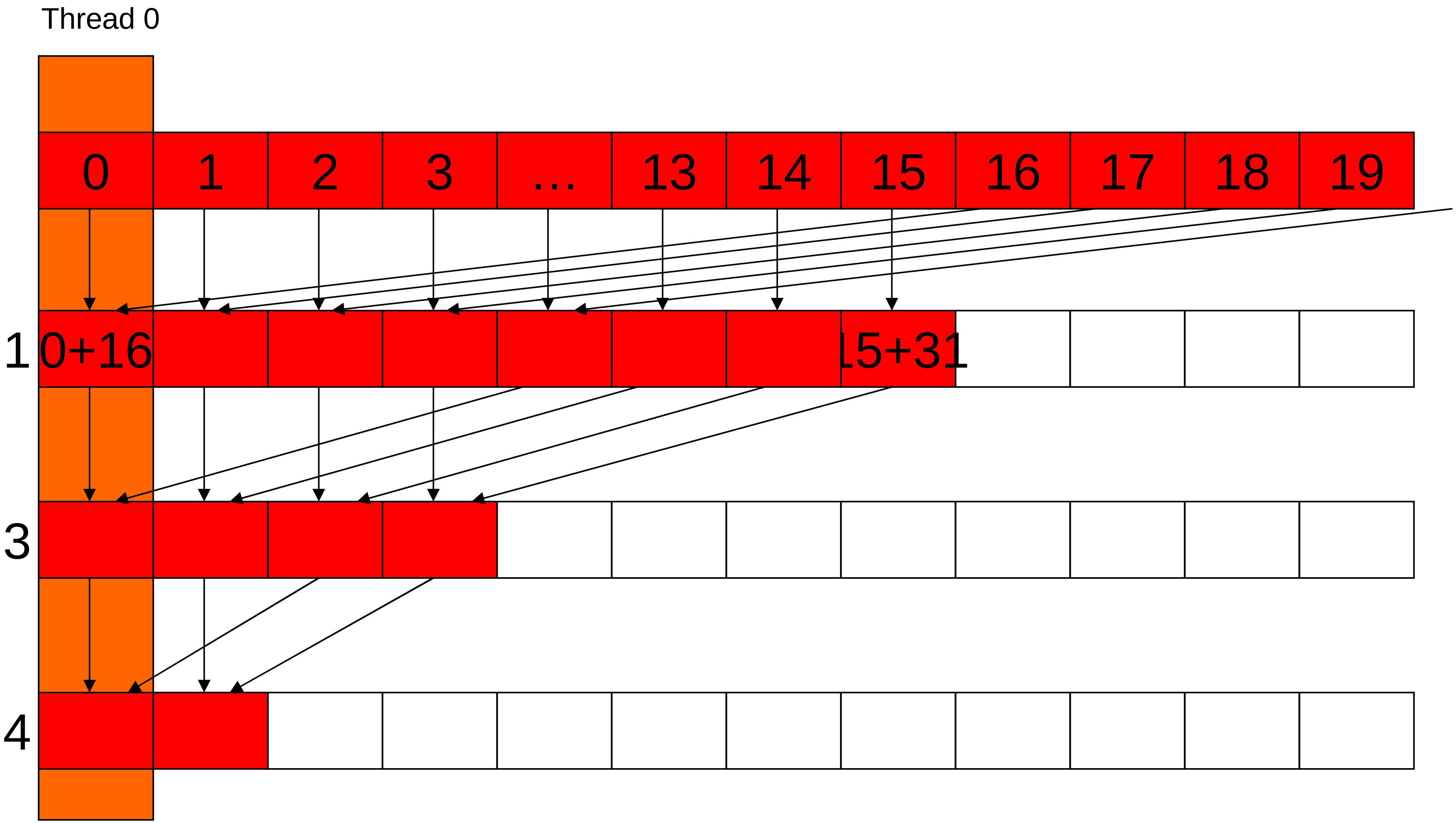
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

Vector Reduction with Branch Divergence



No Divergence until < 16 sub-sums



A Better Implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1;  stride >> 1)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

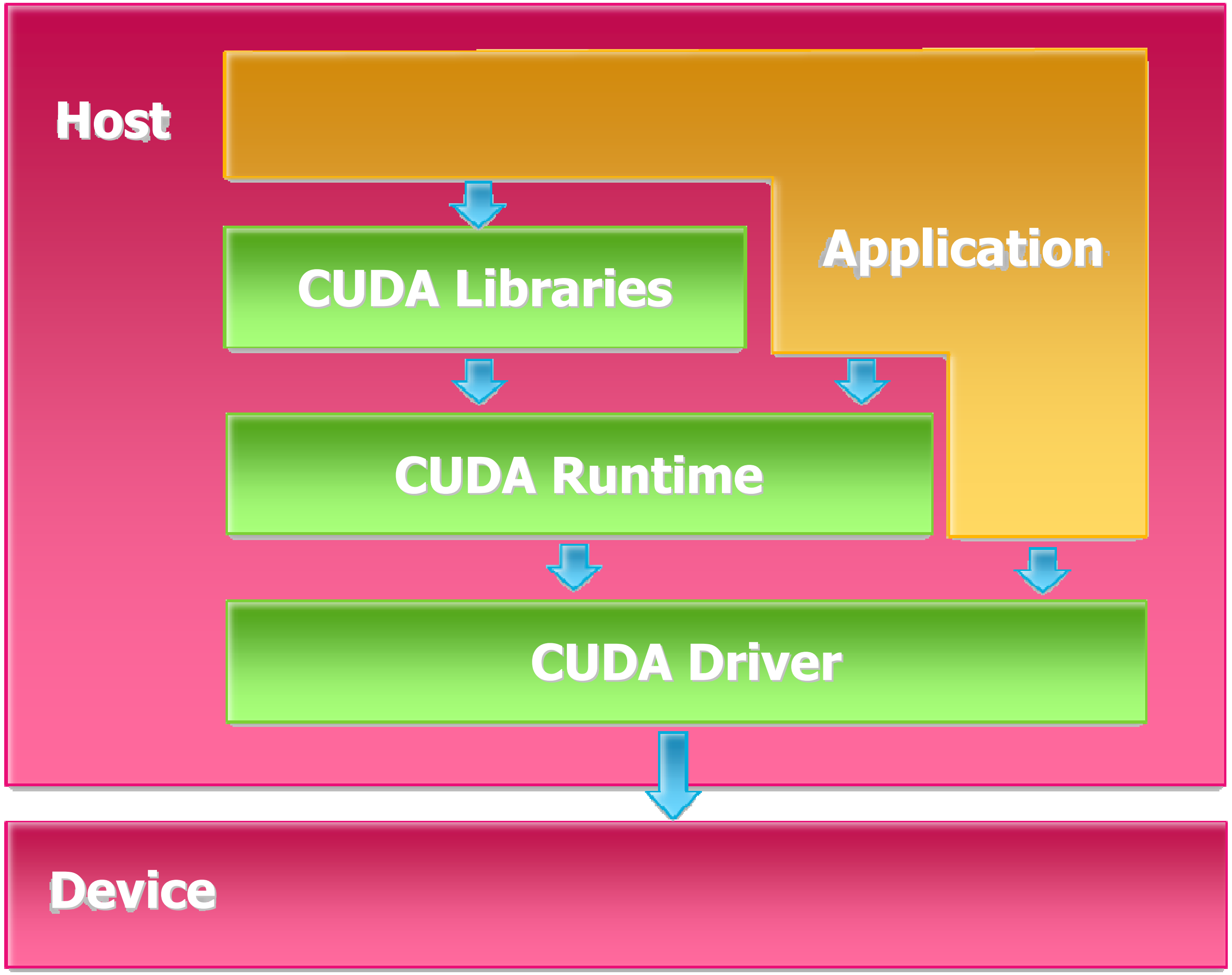
Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of multiprocessors
  - Shared memory size
  - Register file size
  - Threads per block
  - Memory bandwidth

CUDA

Helper Functions

Software Stack



Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware
    - Fast but low accuracy
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy
    - Examples: sin(x), exp(x), pow(x,y)
- When executed on host use C runtime
- The -use_fast_math compiler option forces every func() to compile to __func()

Make your program float-safe!

- Double precision has additional performance cost
  - Careless use of double or undeclared types may run more slowly on G80+
- Avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:
    - `foo = bar * 0.123;    // double assumed`
    - `foo = bar * 0.123f;   // float explicit`

  - Use float version of standard library functions
    - `foo = sin(bar);    // double assumed`
    - `foo = sinf(bar);   // single precision`