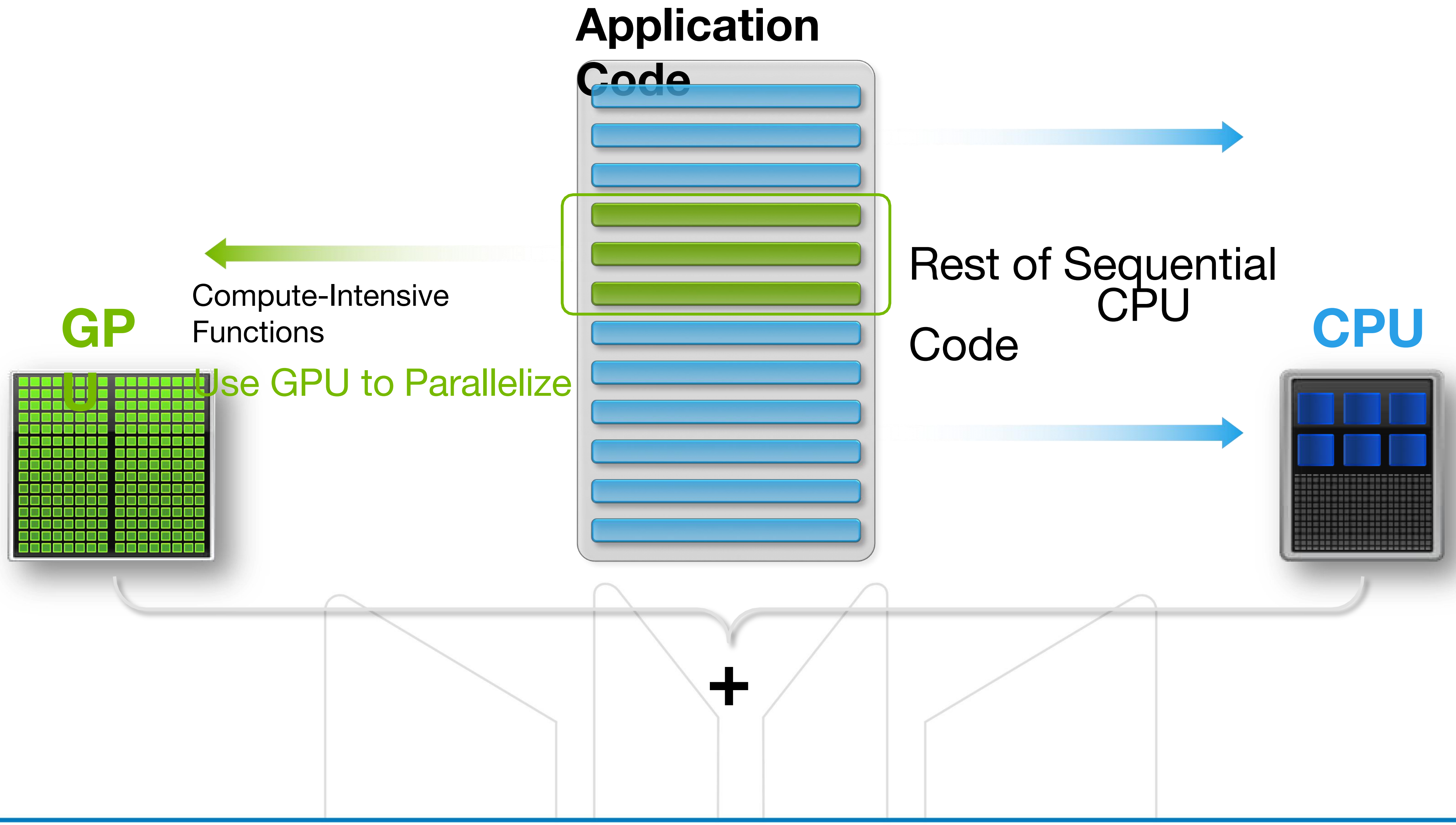
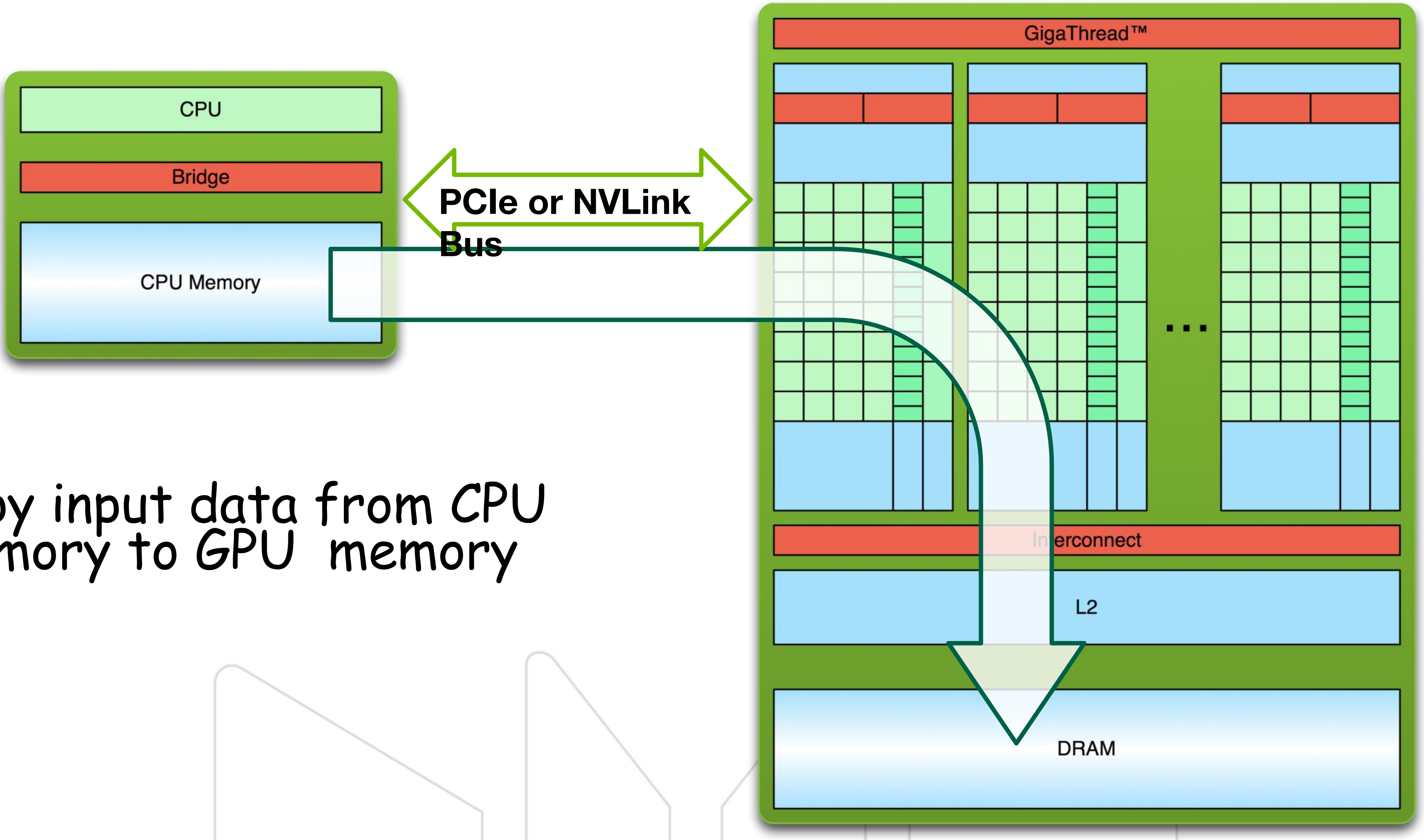
WHAT IS CUDA?

- ❑ **CUDA Architecture**
  - » Expose GPU parallelism for general-purpose computing
  - » Expose/Enable performance
- ❑ **CUDA C++**
  - » Based on industry-standard C++
  - » Set of extensions to enable heterogeneous programming
  - » Straightforward APIs to manage devices, memory etc.
- ❑ **This session introduces CUDA C++**
  - » Other languages/bindings available: Fortran, Python, Matlab, etc.

PORTING TO CUDA

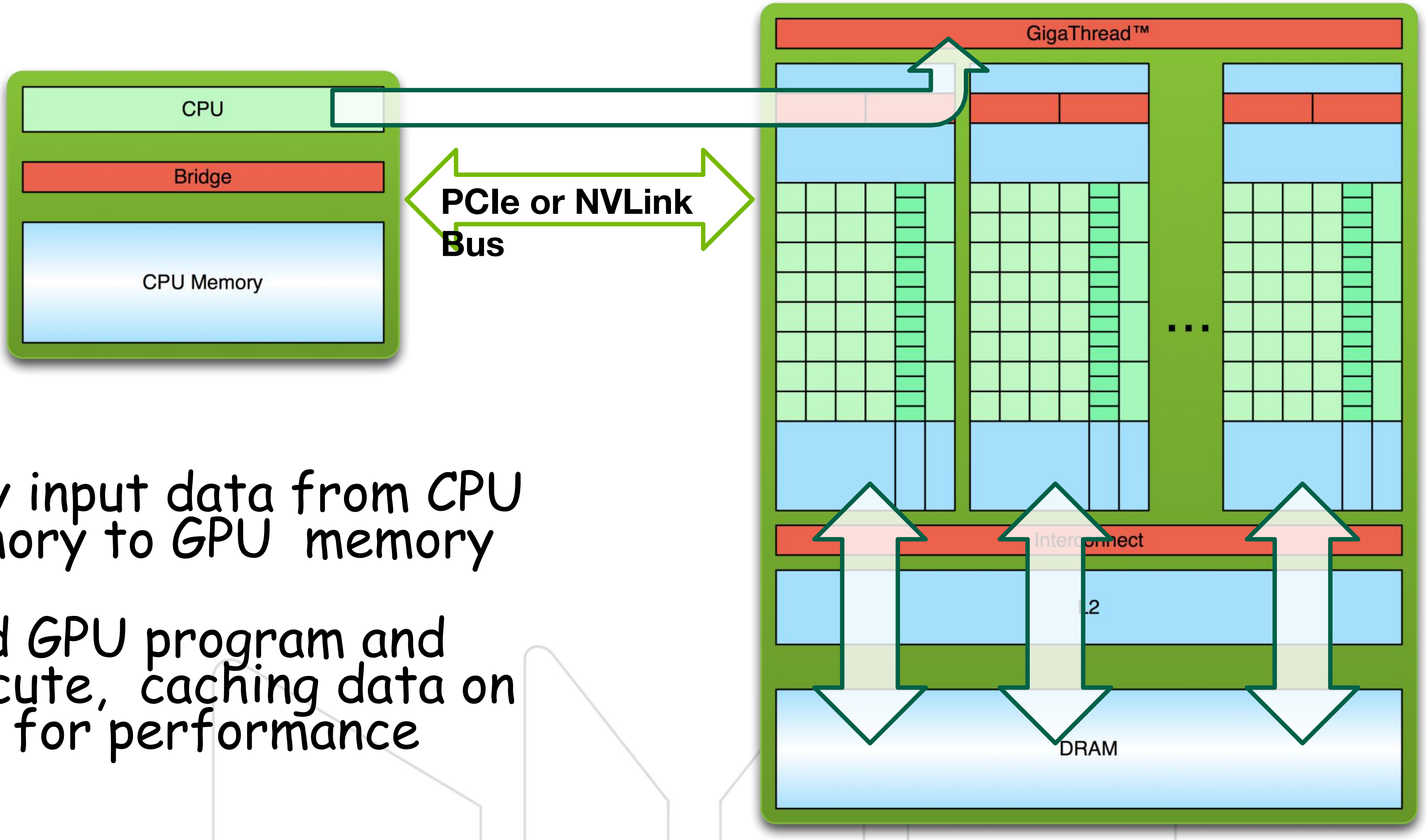


SIMPLE PROCESSING FLOW



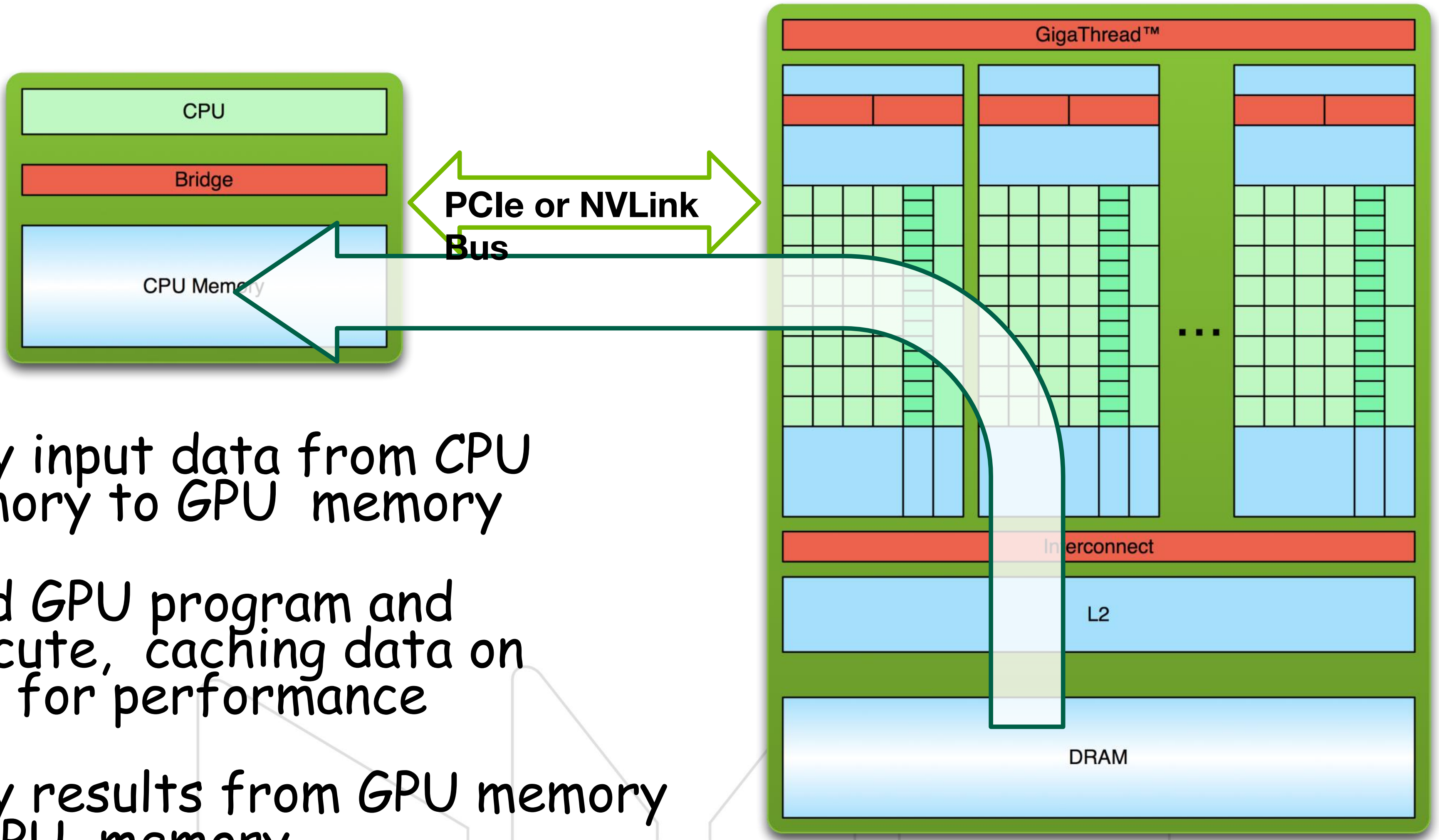
1. Copy input data from CPU memory to GPU memory

SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

GPU KERNELS: DEVICE CODE

```
__global__ void mykernel(void) {}
```

- ❑ CUDA C++ keyword `__global__` indicates a defined function that:
  - » Runs on the device
  - » Is called from host code (can also be called from other device code)
- ❑ nvcc separates source code into host and device components
  - » Device functions (e.g. **mykernel()**) processed by NVIDIA compiler
  - » Host functions (e.g. **main()**) processed by standard host compiler:
  - » gcc, cl.exe

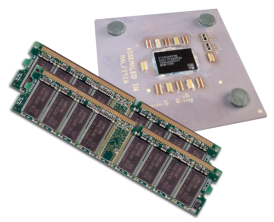
GPU KERNELS: DEVICE CODE

```
mykernel<<<1,1>>>(); // run a kernel on GPU
```

- **Triple angle brackets mark a call to device code**
  - Also called a "kernel launch"
  - the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> execution configuration syntax
  - Each thread that executes the kernel is given a unique thread ID (threadIdx) that is accessible within the kernel through built-in variables.
  - The parameters inside the triple angle brackets are the CUDA kernel execution configuration
- **That's all that is required to execute a function on the GPU!**

MEMORY MANAGEMENT

- ❑ **Host and device memory are separate entities**
  - » Device pointers point to GPU memory
    - Typically passed to device code
    - Typically not dereferenced in host code
- ❑ **Host pointers point to CPU memory**
  - » Typically not passed to device code
  - » Typically not dereferenced in device code
- ❑ **Simple CUDA API for handling device memory**
  - » cudaMalloc(), cudaFree(), cudaMemcpy()
  - » Similar to the C equivalents malloc(), free(), memcpy()



RUNNING CODE IN PARALLEL

- GPU computing is about massive parallelism
  - » So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
     ↓
add<<< N, 1 >>>();
```

  - » Instead of executing add() once, execute N times in parallel

VECTOR ADDITION ON THE DEVICE

- With **add()** running in parallel we can do vector addition
- Terminology: each parallel invocation of **add()** is referred to as a block
  - The set of all blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using blockIdx.x to index into the array, each block handles a different index
- Built-in variables like blockIdx.x are zero-indexed (C/C++ style), 0..N-1, where N is from the kernel execution configuration indicated at the kernel launch

VECTOR ADDITION ON THE DEVICE

```
#define N 512
int main(void) {
   int *a, *b, *c;                 // host copies of a, b, c
   int *d_a, *d_b, *d_c;           // device copies of a, b, c
   int size = N * sizeof(int);
   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);
   // Alloc space for host copies of a, b, c and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

VECTOR ADDITION ON THE DEVICE

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Shared Memory

- Shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it.

- Launching parallel kernels
  - » Launch N copies of add() with add<<<N,1>>>(…);
  - » Use blockIdx.x to access block index

Shared Memory: SHARING DATA BETWEEN THREADS

» Terminology: within a block, threads share data via shared memory

» Shared memory is equivalent to a user-managed cache:

- The application explicitly allocates and accesses it.

» Extremely fast on-chip memory, user-managed

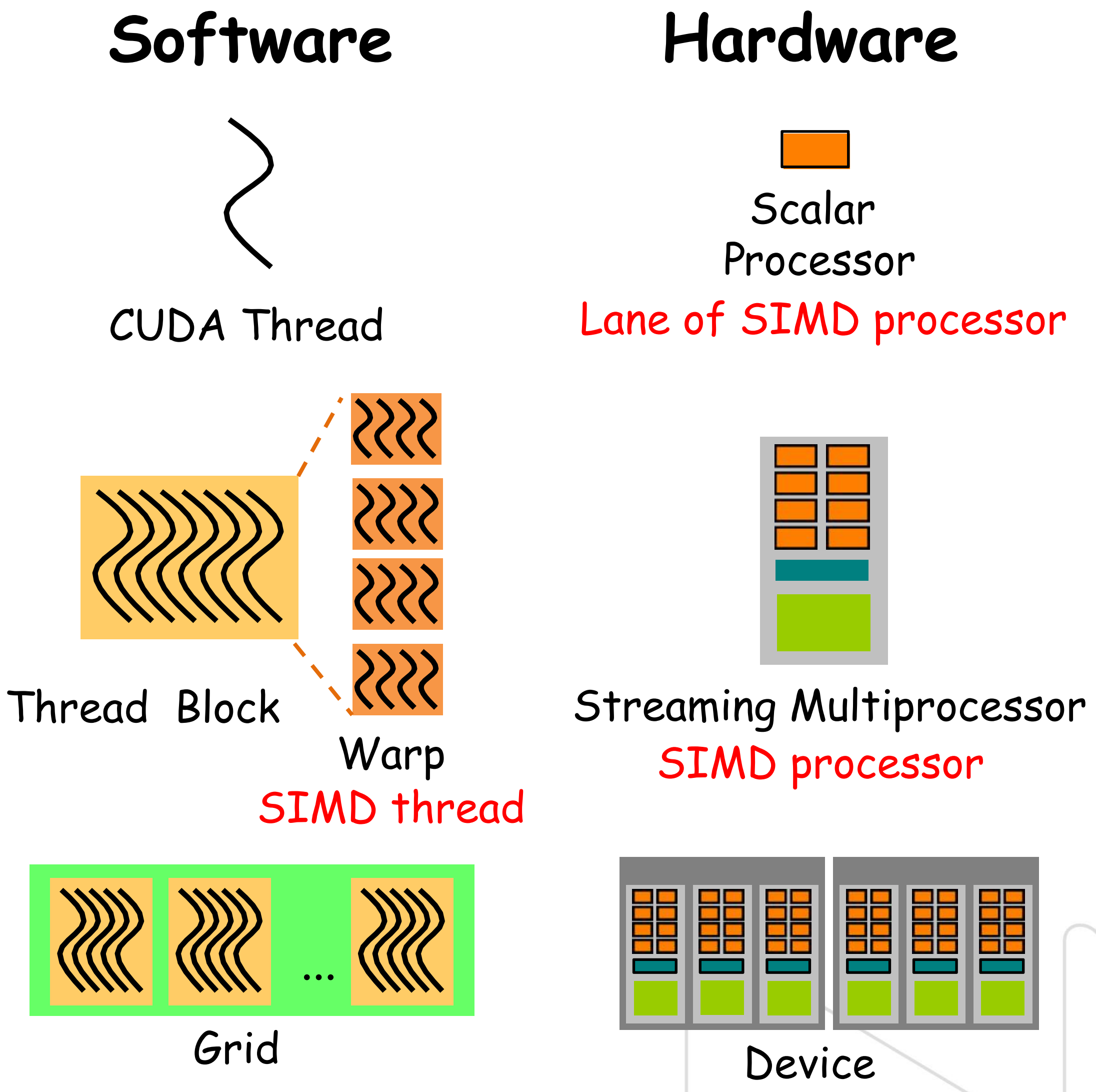
» Declare using __shared__, allocated a variable per block that:

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Has a distinct object per block,
- Is only accessible from all the CUDA threads within the block,

Shared Memory: SHARING DATA BETWEEN THREADS

- **Typical programming pattern:**
  - » Load data from device memory to shared memory
  - » Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads,
  - » Process the data in shared memory
  - » Synchronize again if necessary to make sure that shared memory has been updated with the results, __syncthreads();
  - » Write the results back to device memory.

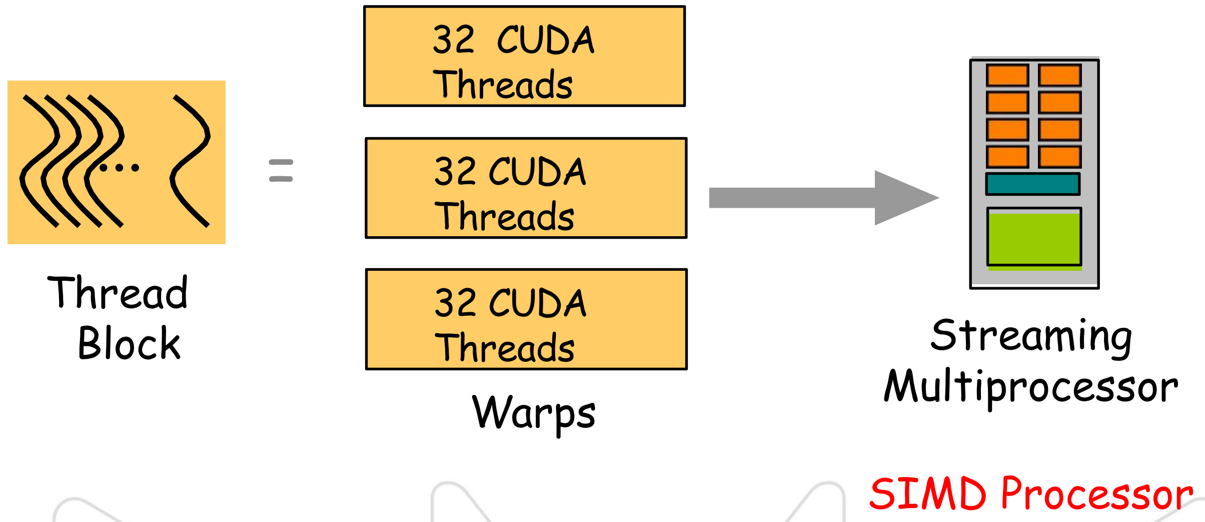
EXECUTION MODEL



- CUDA threads are executed by scalar processors (Lane of SIMD processor)
- Thread blocks are split to warps and executed on multiprocessors (SIMD processor) in SIMT.
- Several concurrent thread blocks can reside on one SIMD processor (when the number of thread blocks greater than multiprocessors) - limited by SIMD processor resources (shared memory and register file)
- A kernel is launched as a grid of thread blocks

WARPS 线程束

- ❑ A thread block consists of thread warps
- ❑ A warp is executed physically in parallel (SIMD) on a multiprocessor



LAUNCH CONFIGURATION: SUMMARY

- **Need enough total threads to keep GPU busy**
  - » Typically, you'd like 512+ CUDA threads per SIMD Processor (aim for 2048 - maximum "occupancy")
    - More if processing one fp32 element per thread
  - » Of course, exceptions exist
- **Threadblock configuration**
  - » Threads per block should be a multiple of warp size (32)
  - » SIMD Processor can concurrently execute at least 16 thread blocks (Maxwell/Pascal/Volta/Ampere: 32)
    - Really small thread blocks prevent achieving good occupancy
    - Really large thread blocks are less flexible
    - Could generally use 128-256 threads/block, but use whatever is best for the application

GPU MEM OPERATIONS

» Loads:

- Caching
  - Default mode
  - Attempts to hit in L1, then L2, then GMEM
  - Load granularity is 128-byte line

» Stores:

- Invalidate L1, write-back for L2

GPU MEM OPERATIONS

» Loads:

- Non-caching
  - Compile with -Xptxas -dlcm=cg option to nvcc
  - Attempts to hit in L2, then GMEM

    Do not hit in L1, invalidate the line if it's in L1 already

    The L1 can be bypassed with a non-caching load.
  - Load granularity is 32-bytes (segment)

LOAD OPERATION

» Memory operations are issued per warp (32 CUDA threads, a SIMD thread)

- Just like all other instructions

» Operation:

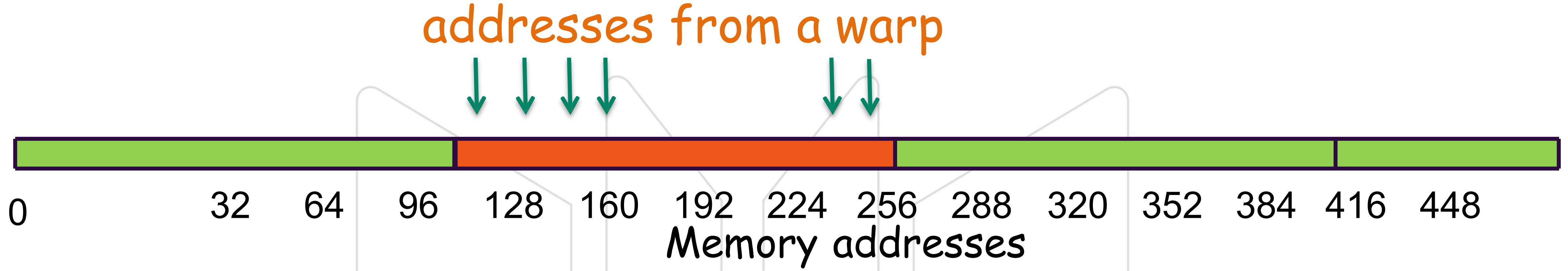
- CUDA Threads in a warp provide memory addresses
- Determine which lines/segments are needed
- Request the needed lines/segments

CACHING LOAD

» Warp requests 32 aligned, consecutive 4-byte words

» Addresses fall within 1 cache-line

- Warp needs 128 bytes
- 128 bytes move across the bus on a miss (data load granularity)
- Bus utilization: 100%
- int c = a[idx];...

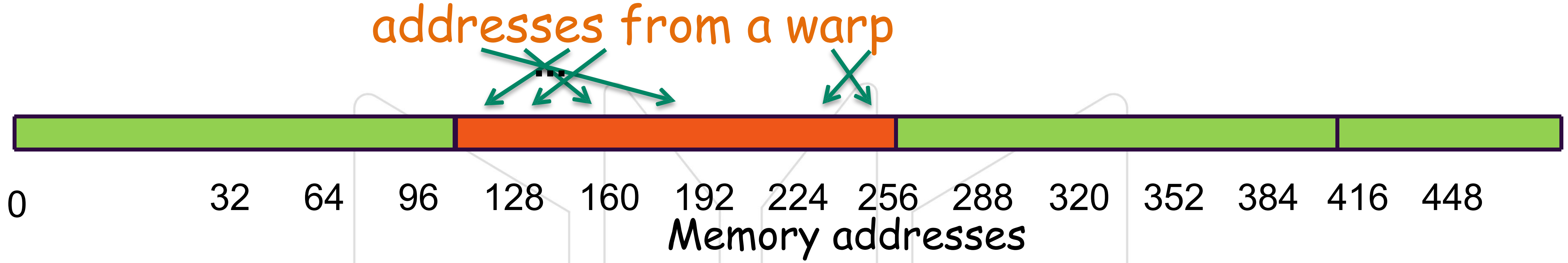


CACHING LOAD

» Warp requests 32 aligned, permuted 4-byte words

» Addresses fall within 1 cache-line

- Warp needs 128 bytes
- 128 bytes move across the bus on a miss
- Bus utilization: 100%
- int c = a[rand()%warpSize];

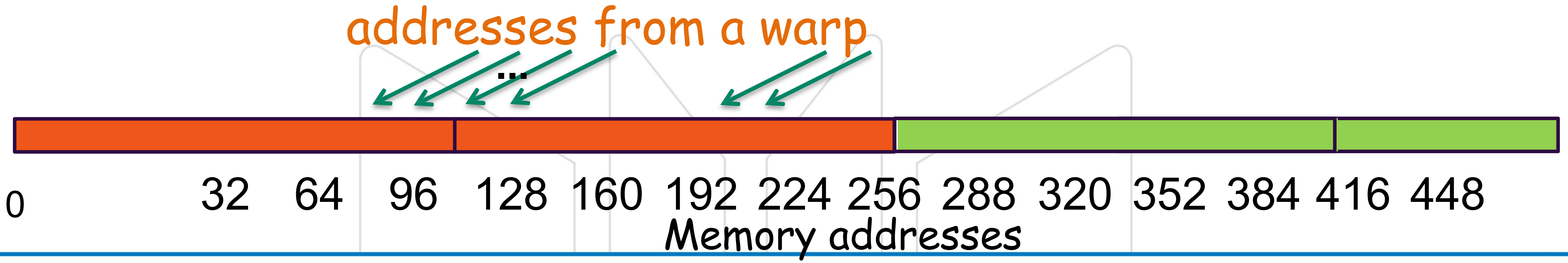


CACHING LOAD

» Warp requests 32 misaligned, consecutive 4-byte words, e.g. bytes[126, 253]

» Addresses fall within 2 cache-lines

- Warp needs 128 bytes
- 256 bytes move across the bus on misses
- Bus utilization: 50%
- int c = a[idx-2];

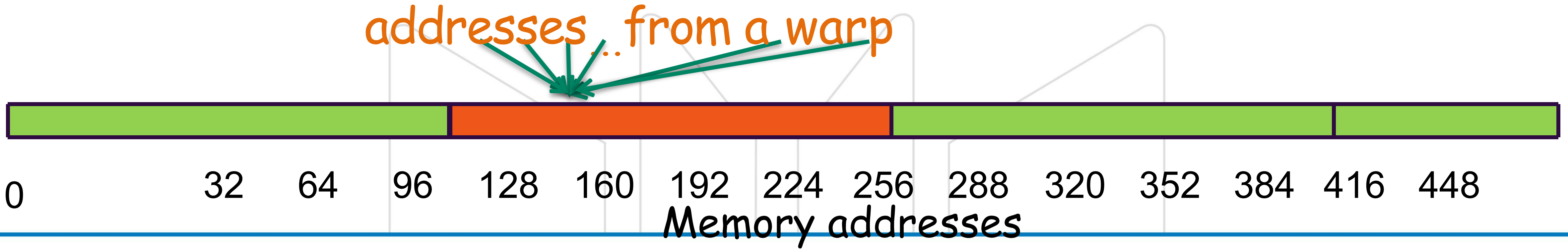


CACHING LOAD

» All threads in a warp request the same 4-byte word

» Addresses fall within a single cache-line

- Warp needs 4 bytes
- 128 bytes move across the bus on a miss
- Bus utilization: 3.125% (4/128), load 128 bytes, only using 4 bytes
- int c = a[40];

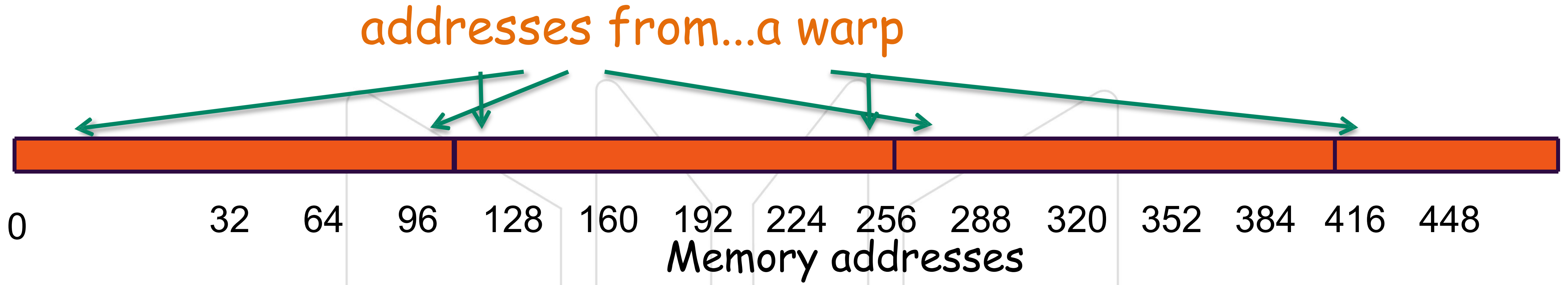


CACHING LOAD

» Warp requests 32 scattered 4-byte words

» Addresses fall within N cache-lines

- Warp needs 128 bytes
- N*128 bytes move across the bus on a miss
- Bus utilization: 128 / (N*128) (3.125% worst case N=32)
- int c = a[rand()];

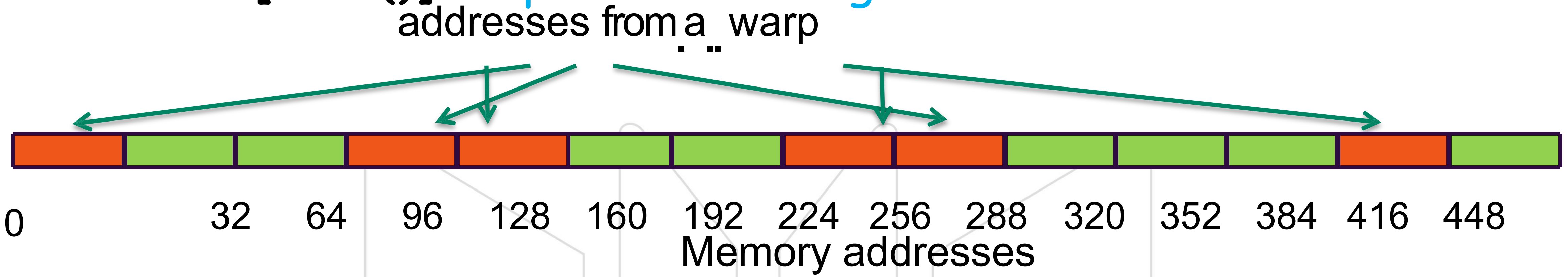


NON-CACHING LOAD

» Warp requests 32 scattered 4-byte words

» Addresses fall within N segments

- Warp needs 128 bytes
- N*32 bytes move across the bus on a miss
- Bus utilization: 128 / (N*32) (12.5% worst case N = 32)
- int c = a[rand()]; -Xptxas -dlcm=cg



GPU MEM OPTIMIZATION GUIDELINES

» Strive for perfect coalescing

- (Align starting address - may require padding)
- A warp should access within a contiguous region

» Have enough concurrent accesses to saturate the bus

- Process several elements per thread
  - Multiple loads get pipelined
  - Indexing calculations can often be reused
- Launch enough warps to maximize throughput
  - Latency is hidden by switching warps

» Use all the caches!

SHARED MEMORY

» Uses:

- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses , like CPU cache
- Use it to improve global memory access patterns
- The shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache)
- Can be managed (allocate and free) via programming API

» Organization:

- 32 banks, 4-byte wide banks
- Successive 4-byte words belong to different banks

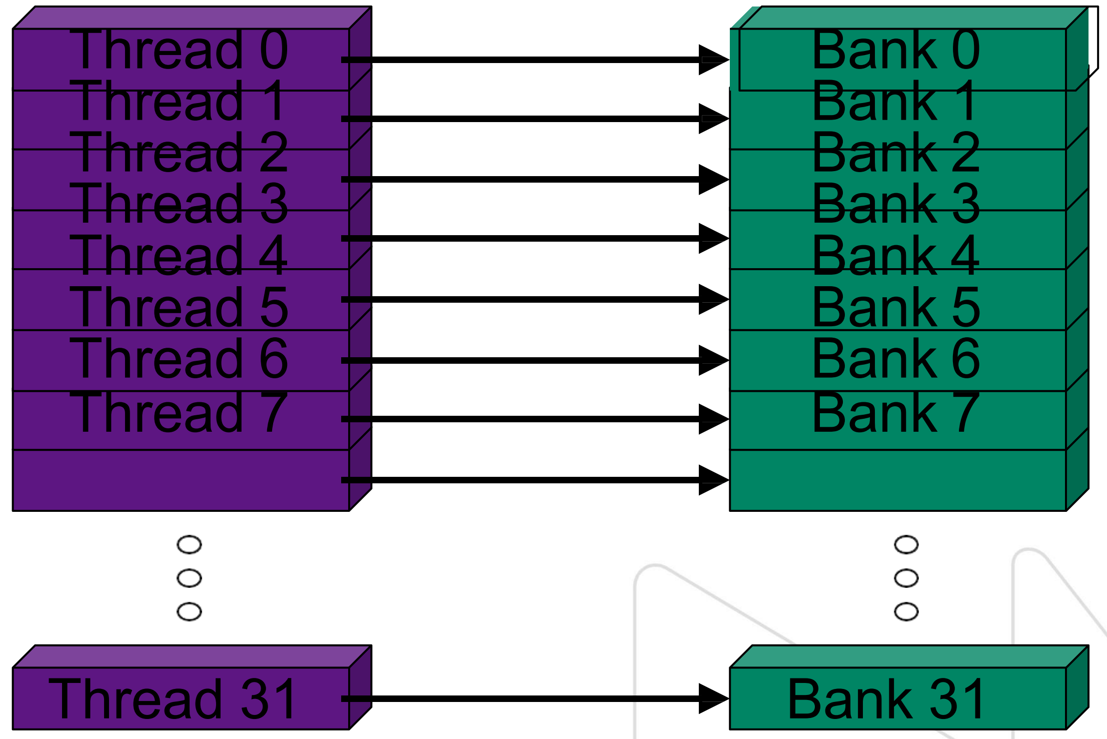
SHARED MEMORY

» Performance:

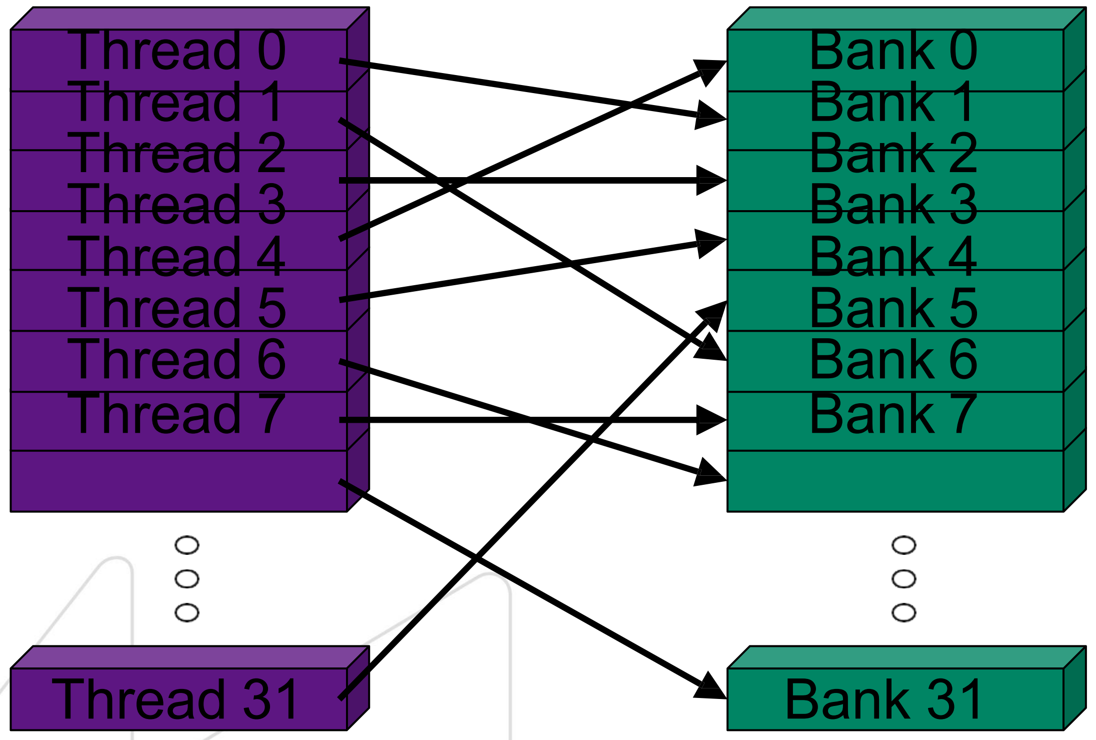
- Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
- shared accesses are issued per 32 threads (warp)
- serialization: if N threads of 32 access different 4-byte words in the same bank (bank conflict), N accesses are executed serially

BANK ADDRESSING EXAMPLES

No Bank Conflicts

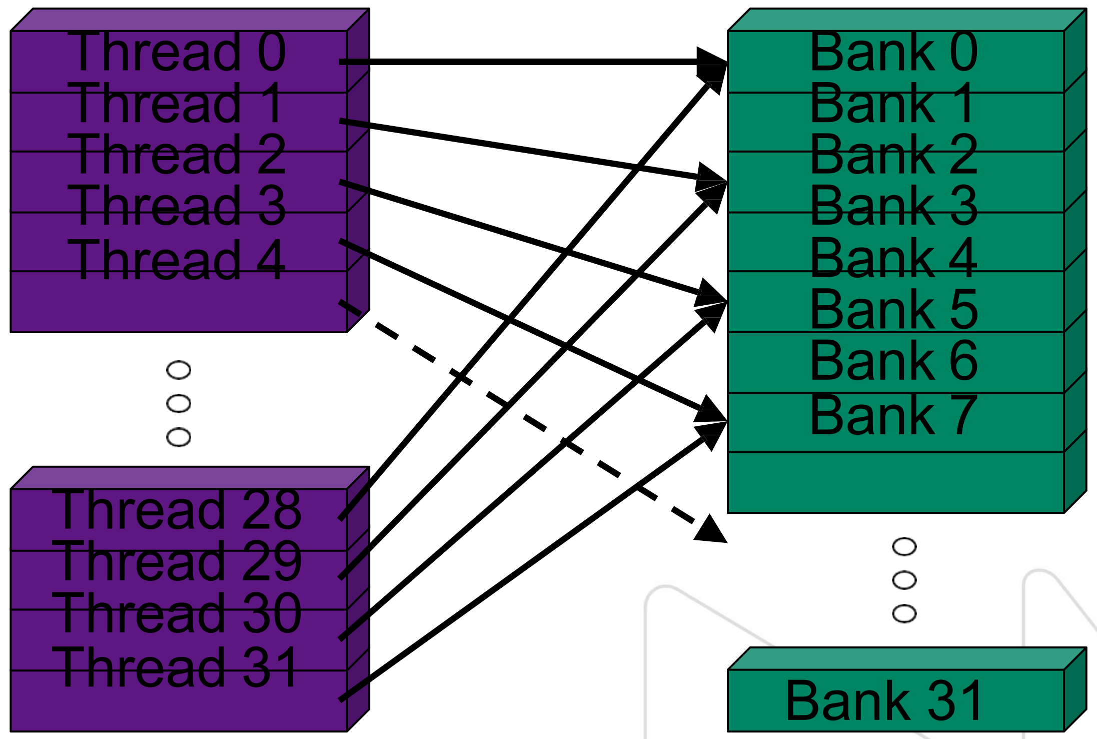


No Bank Conflicts

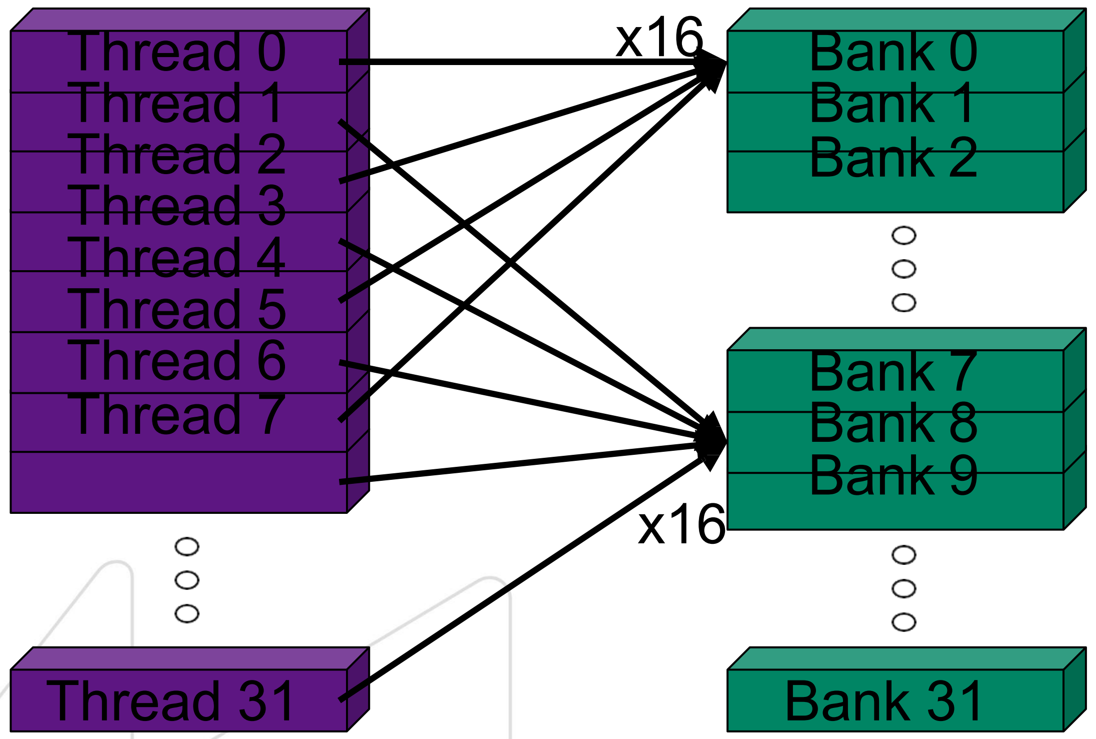


BANK ADDRESSING EXAMPLES

2-way Bank Conflicts



16-way Bank Conflicts


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 31
x16
x16
Bank 0
Bank 1
Bank 2
Bank 7
Bank 8
Bank 9
Bank 31