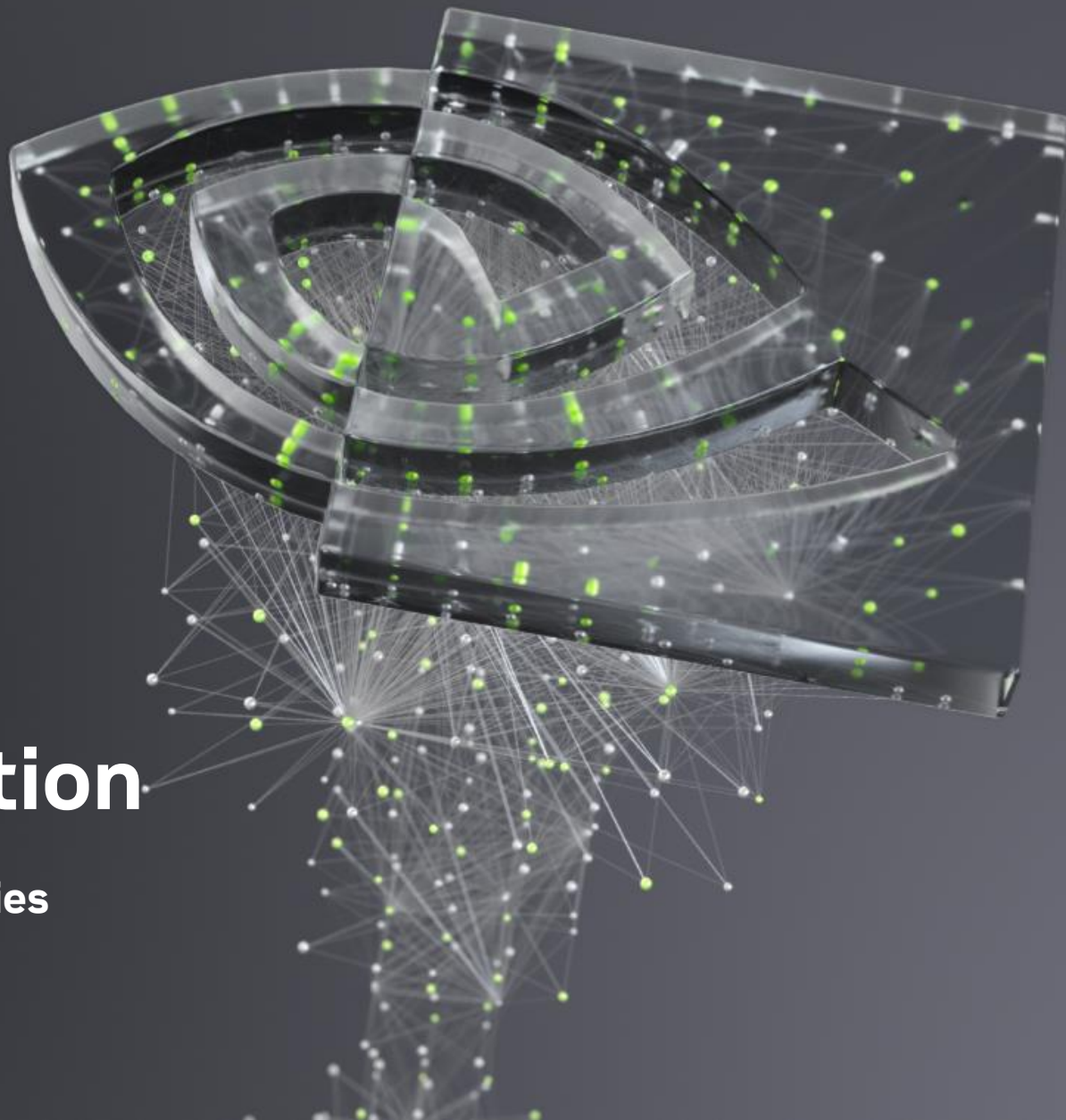




# CUDA/GPU Introduction

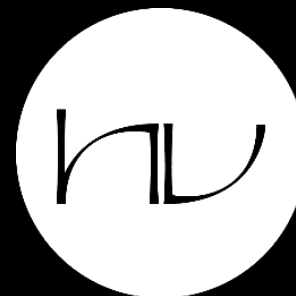
Nicolas Blin – AI DevTech

Michel Migdal – CUDA Math and AI Libraries



# About Us

Nicolas Blin



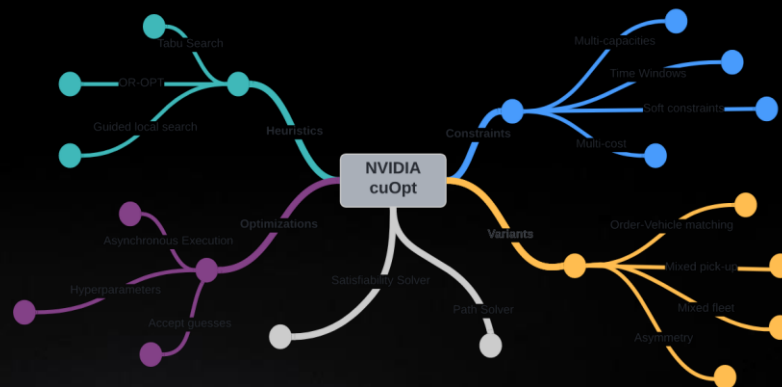
Vice-president of the [Digital Holography Foundation](#)

Teaching Advanced GPU Computing @ NVIDIA - EPITA

DevTech @ NVIDIA

GPU Solver in Pickup & Delivery and Linear Programming

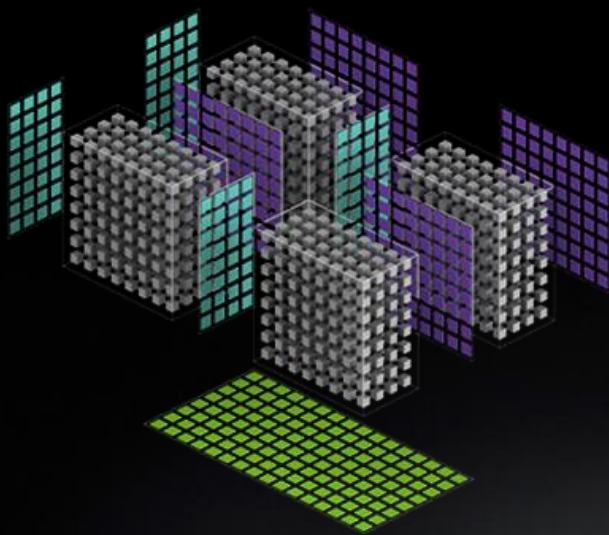
Scientific papers related to GPU computing



# About Us

Michel Migdal

A100 FP16



## Codeplay (now part of Intel)

- Developing SYCL on CUDA ecosystem, Intel/LLVM.
- GPU RPC library: accessing OS file system from GPU

## (Holidays projects)

- Heterogeneous Ethereum miner (free heating for winter)
- GPU Graph runtime

## NVIDIA Image Codecs

- Improving image codecs (for some AI folks)
- Support for Grace-Hopper and embedded systems
- Wrote new “Memory-less decoders” for nvJPEG

## NVIDIA cuBLAS

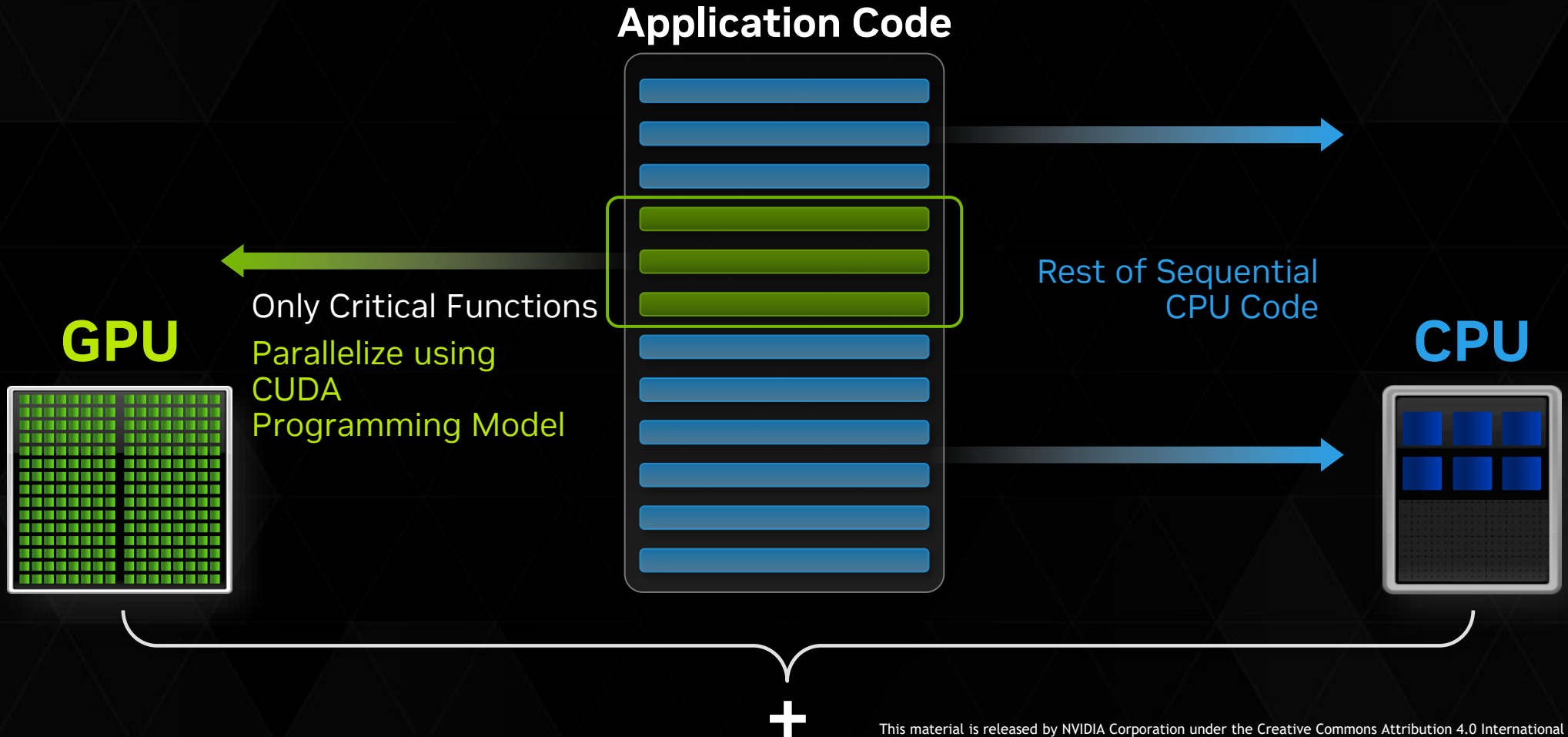
- Mostly not working in cuBLAS
- Hardware fingerprints
- GEMM support and optimizations in DL compilers and kernel generators
- ...

# CUDA C++

## What to expect?

- Introduction to GPU Architecture fundamentals
- GPU Memory and Programming Model
- CUDA C++ programming

# GPU COMPUTING





# CUDA ARCHITECTURE PROGRAMMING MODEL

# CUDA KERNELS

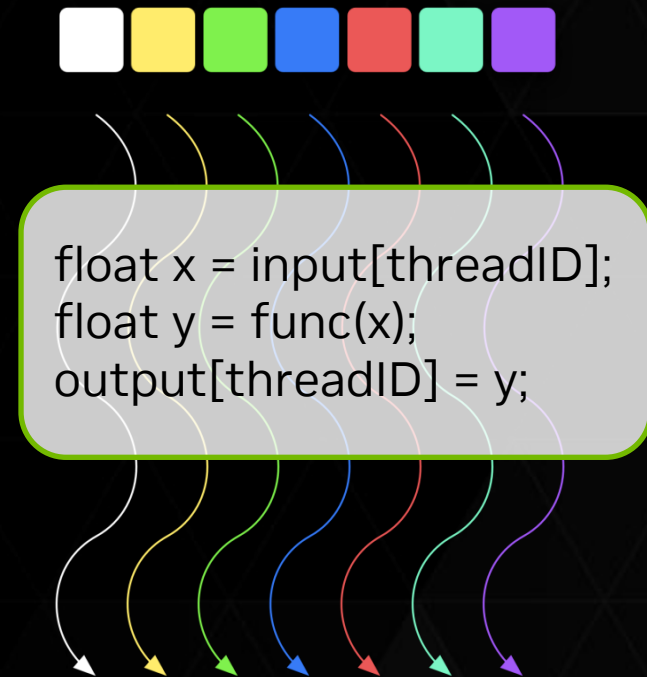
(called programmable shaders in graphics world)

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads (unless your kernel does not fill the GPU)
- CUDA threads:
  - Lightweight
  - Fast switching (thanks to many registers, opposed to CPU context switch)
  - Tens of thousands execute simultaneously, and up to 300 000 resident at any time

CPU	Host	Executes functions
GPU	Device	Executes kernels

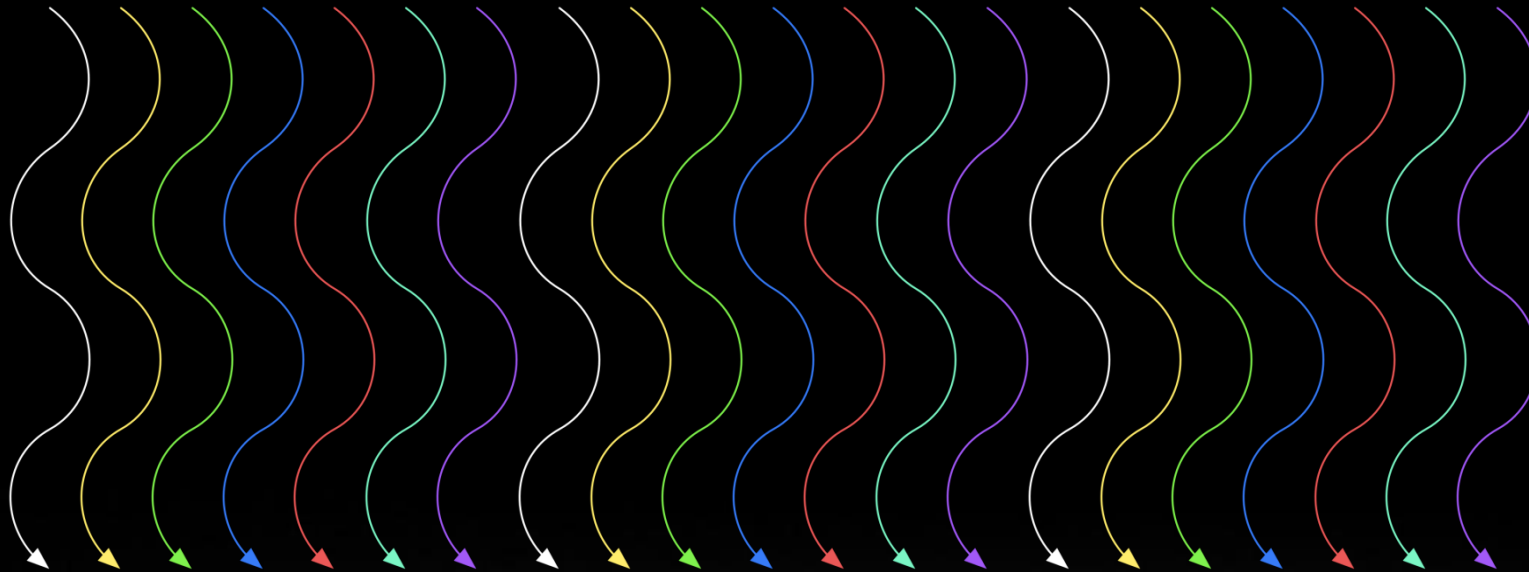
# CUDA KERNELS: PARALLEL THREADS

- A **kernel** is a function executed on the GPU
  - Array of threads, in parallel
- All threads execute the same program, can take different paths
  - Each thread has an ID (accessed through magic registers)
  - Select input/output data
  - Control decisions

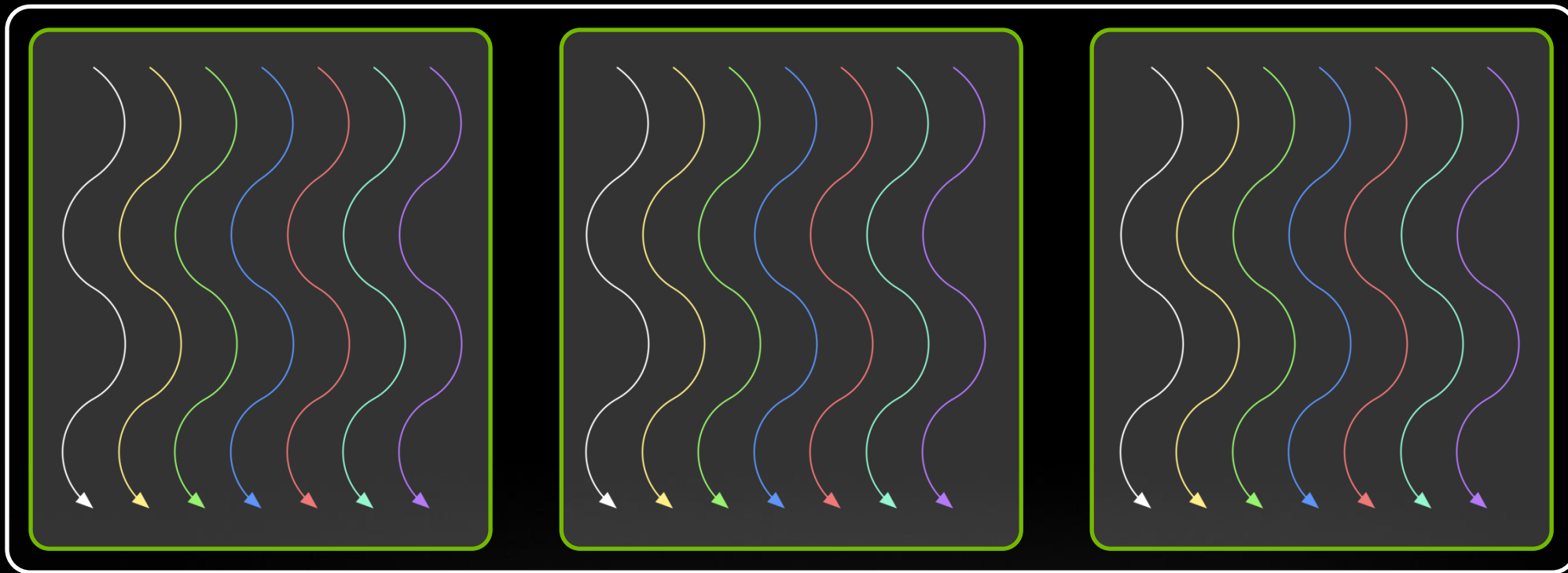




# CUDA KERNELS: SUBDIVIDE INTO BLOCKS

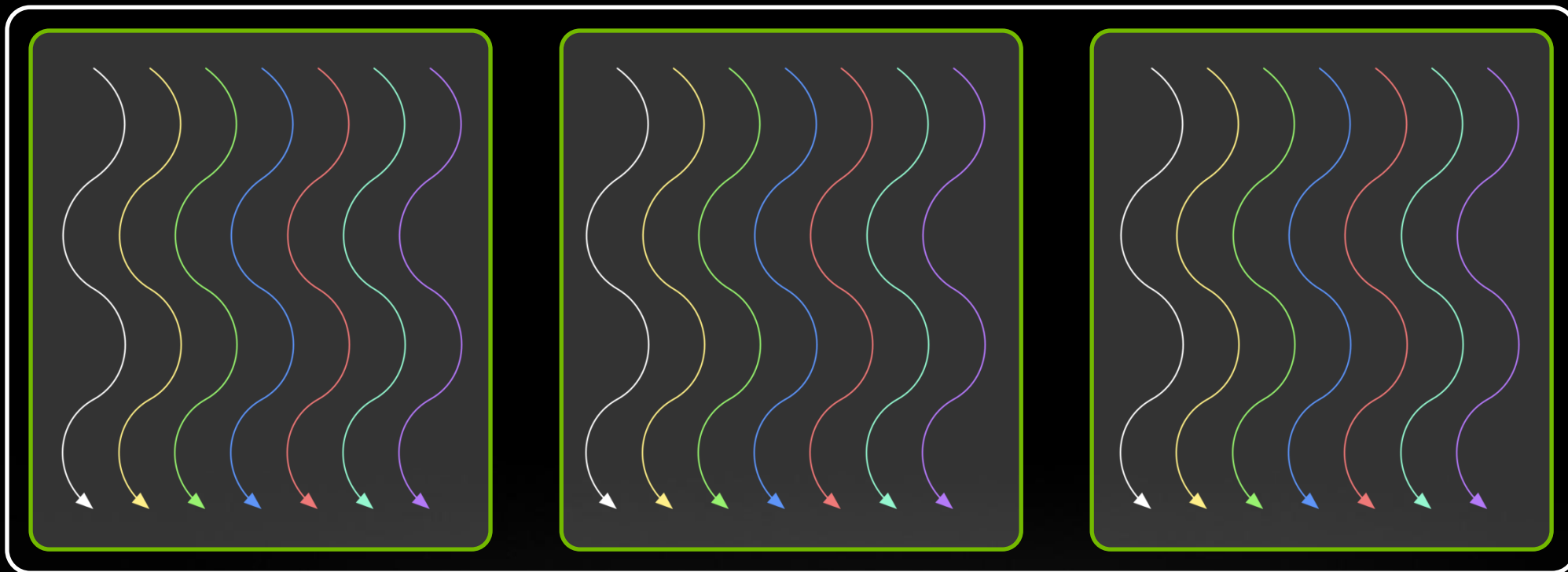


# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



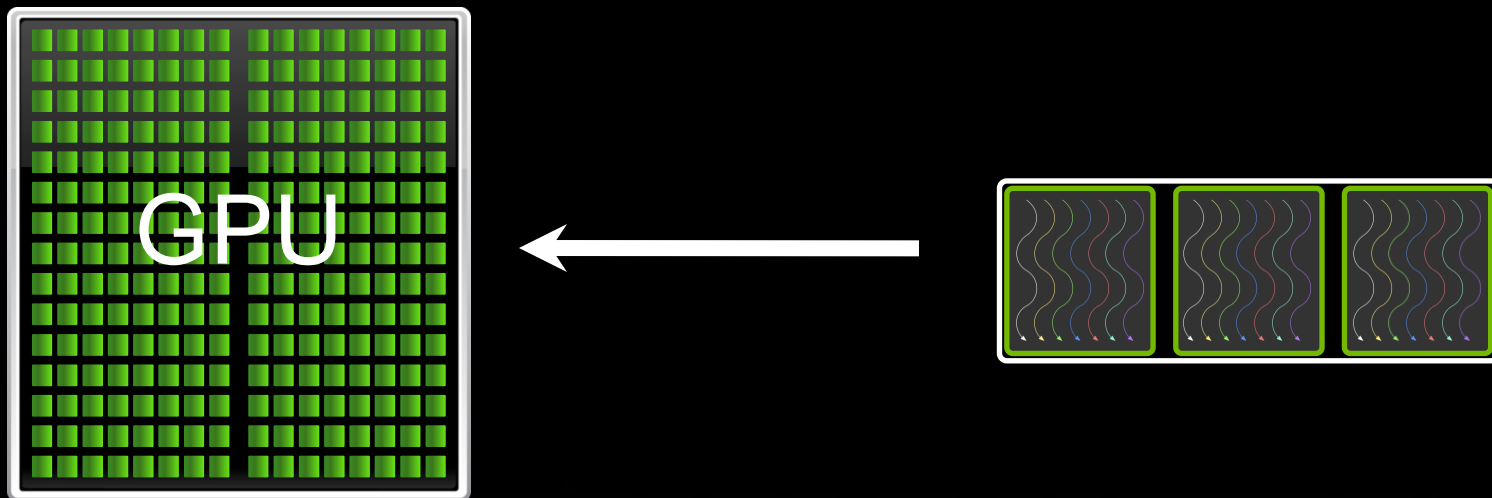
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



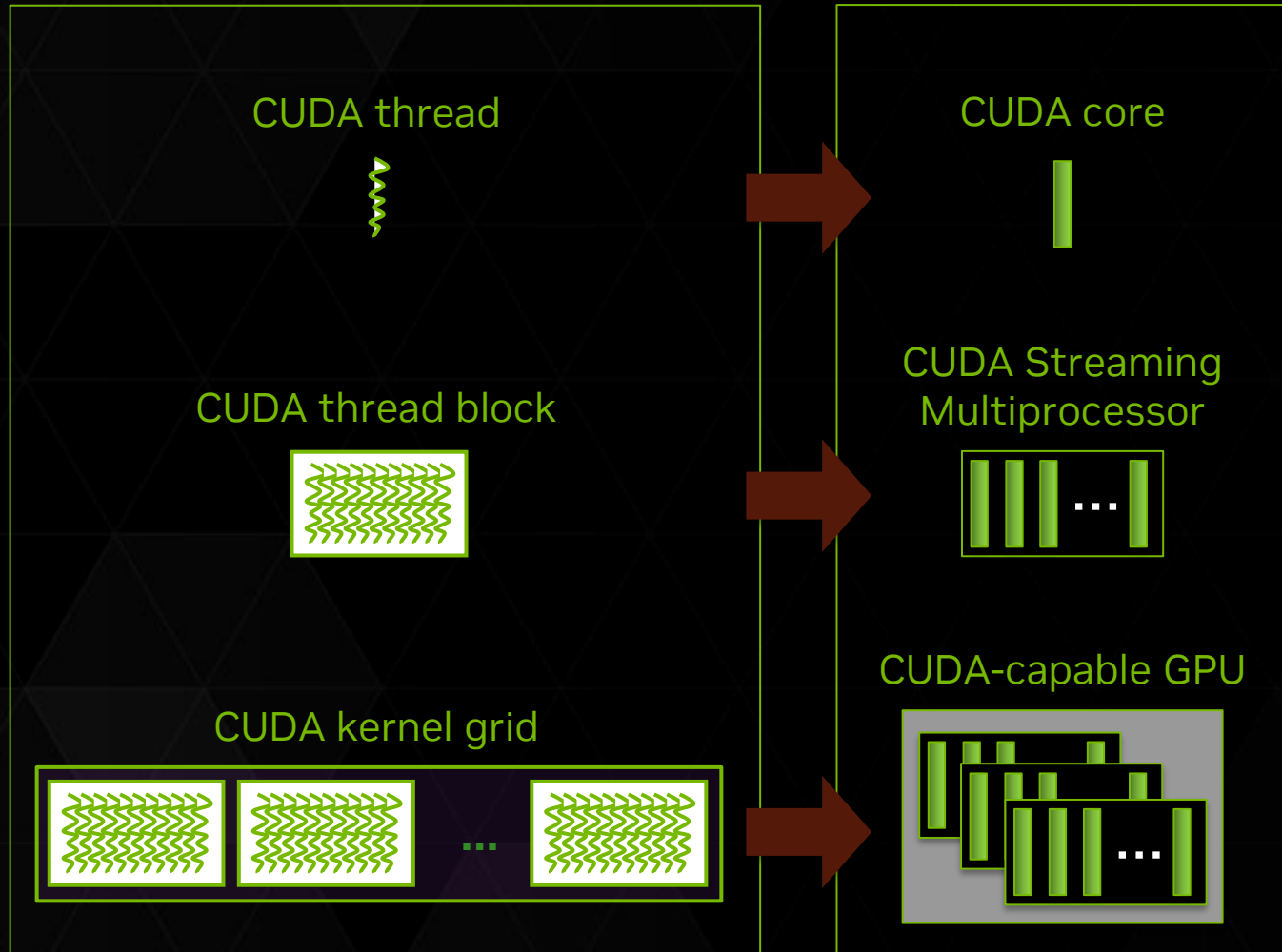
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



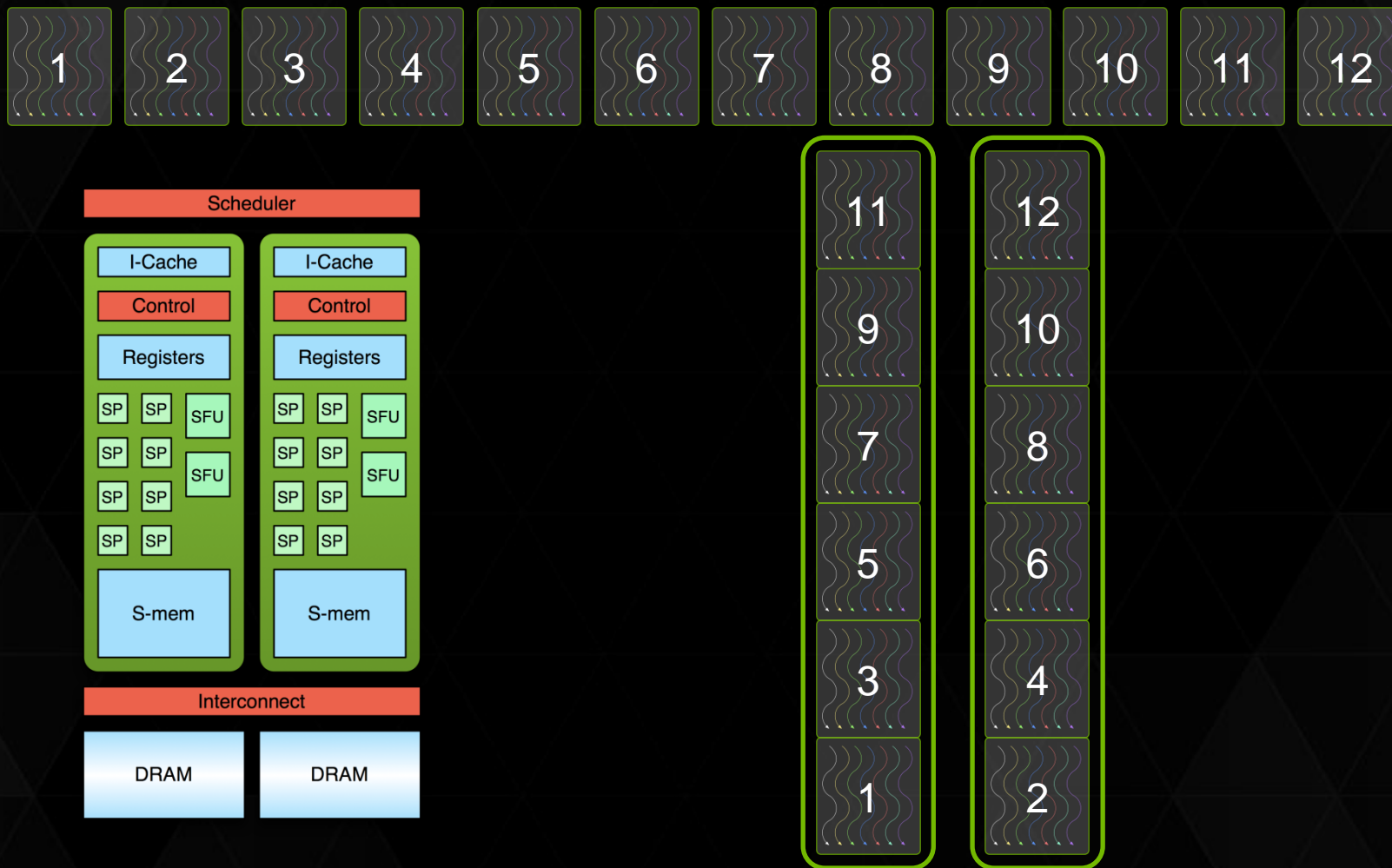
- Threads are grouped into **blocks**
- **Blocks** are grouped into a **grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

# KERNEL EXECUTION



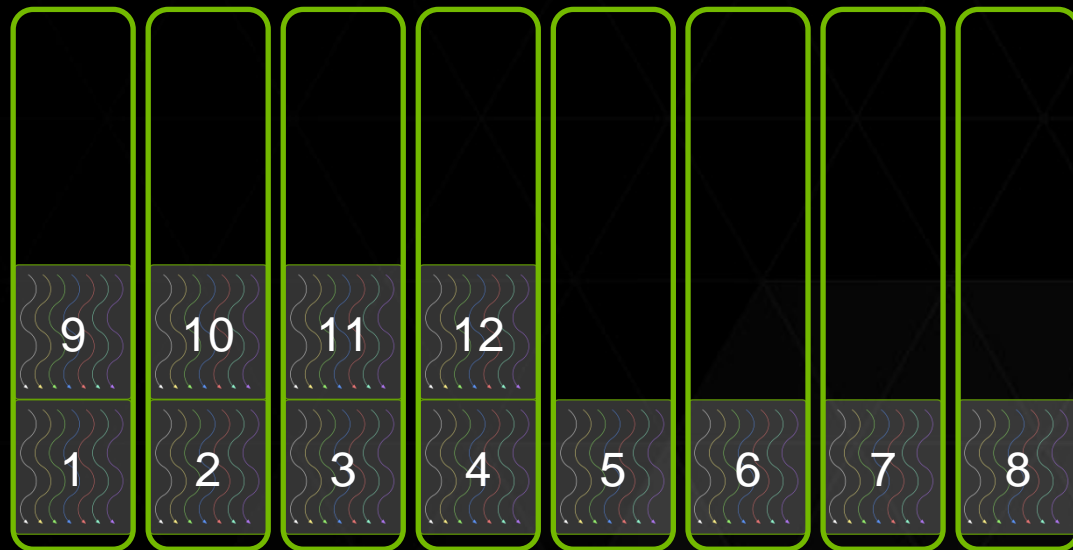
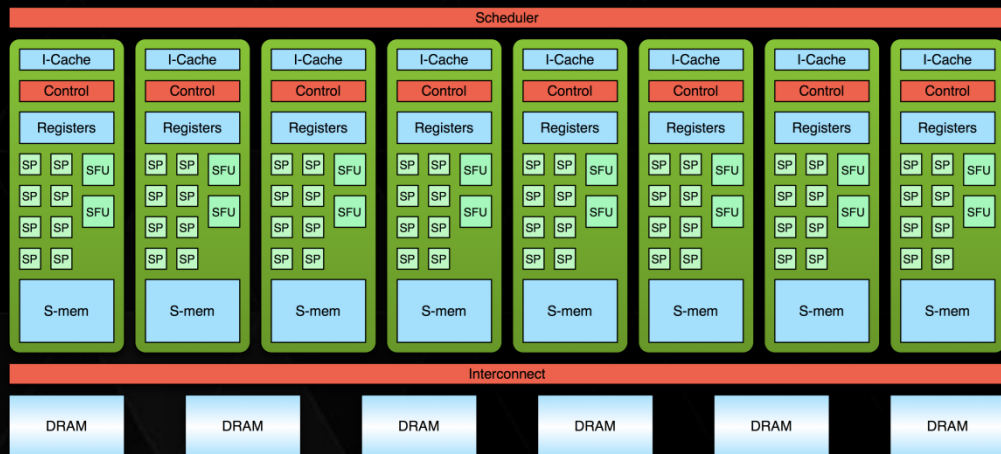
- Each thread is executed by a core
- A core executes many threads
- Each block is executed by one SM and does not migrate (if you check the ISA you realize it can migrate)
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

# TRANSPARENT SCALABILITY

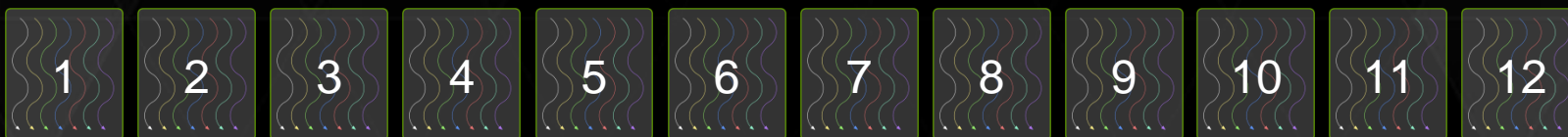


# TRANSPARENT SCALABILITY

(and some grid quantization)



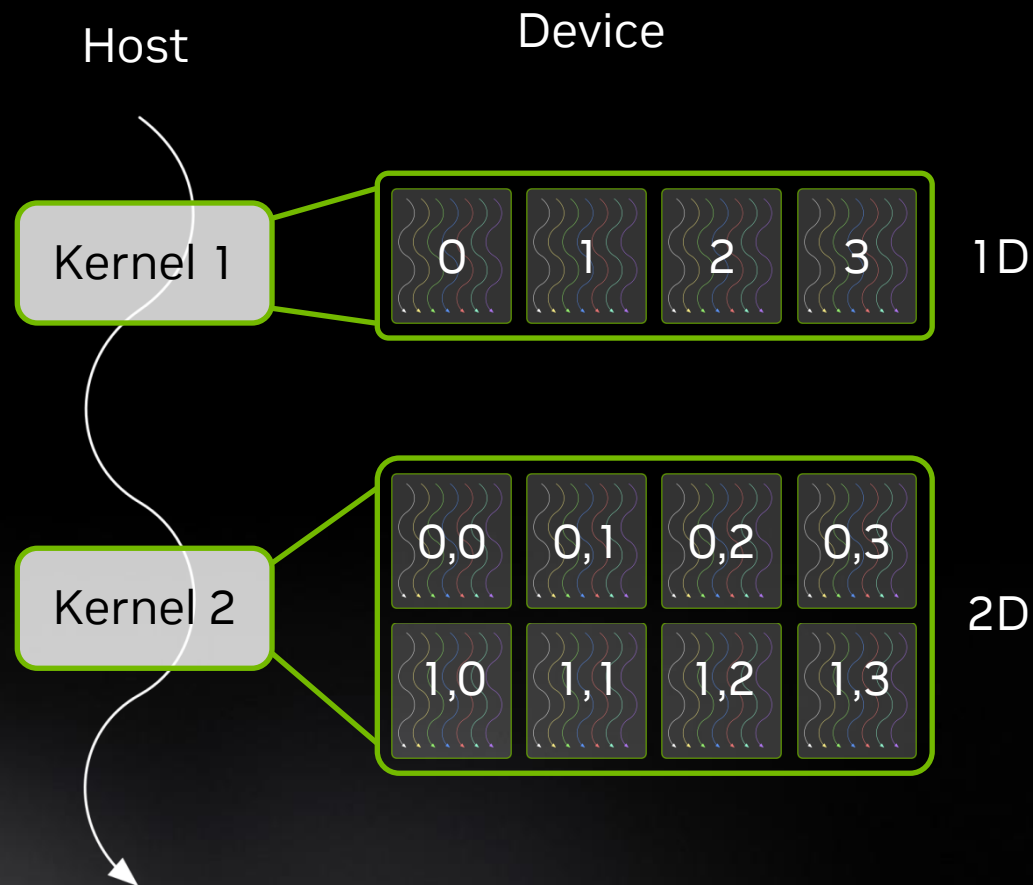
# TRANSPARENT SCALABILITY –





# CUDA PROGRAMMING MODEL - SUMMARY

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID





# CUDA ARCHITECTURE MEMORY MODEL

# GPU ARCHITECTURE

Two Main components

## Global memory

Analogous to RAM in a CPU server

Accessible by both GPU and CPU

48GB with bandwidth currently up to 1 TB/s

## Streaming Multiprocessors (SMs)

SMs perform the actual computations

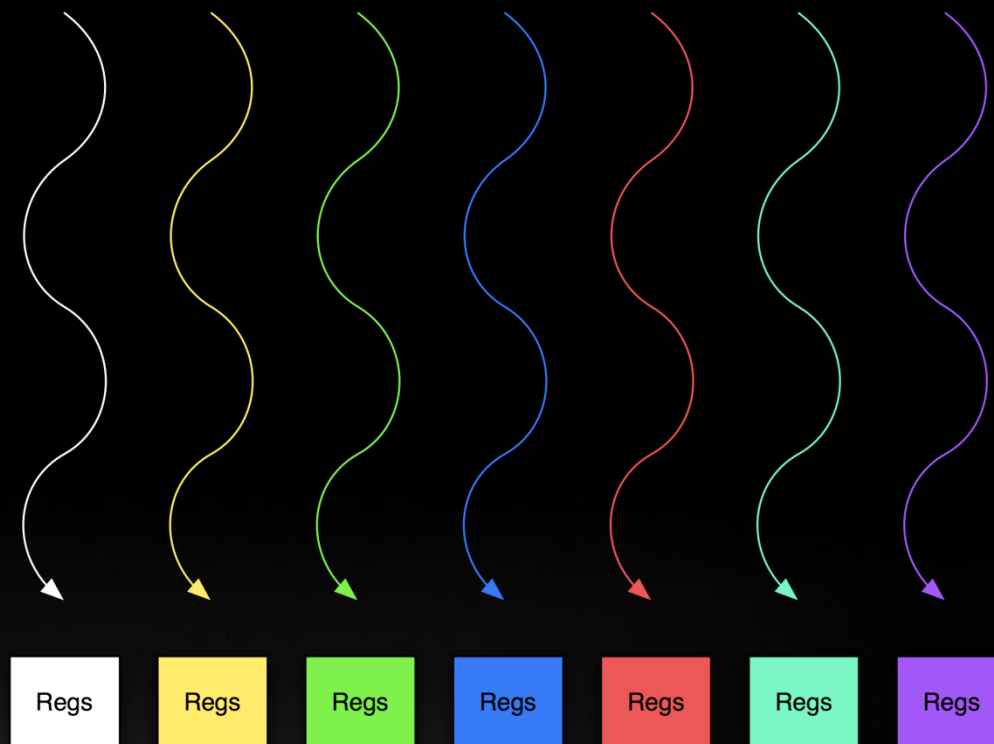
Each SM has its own:

Control units, registers, execution pipelines, caches



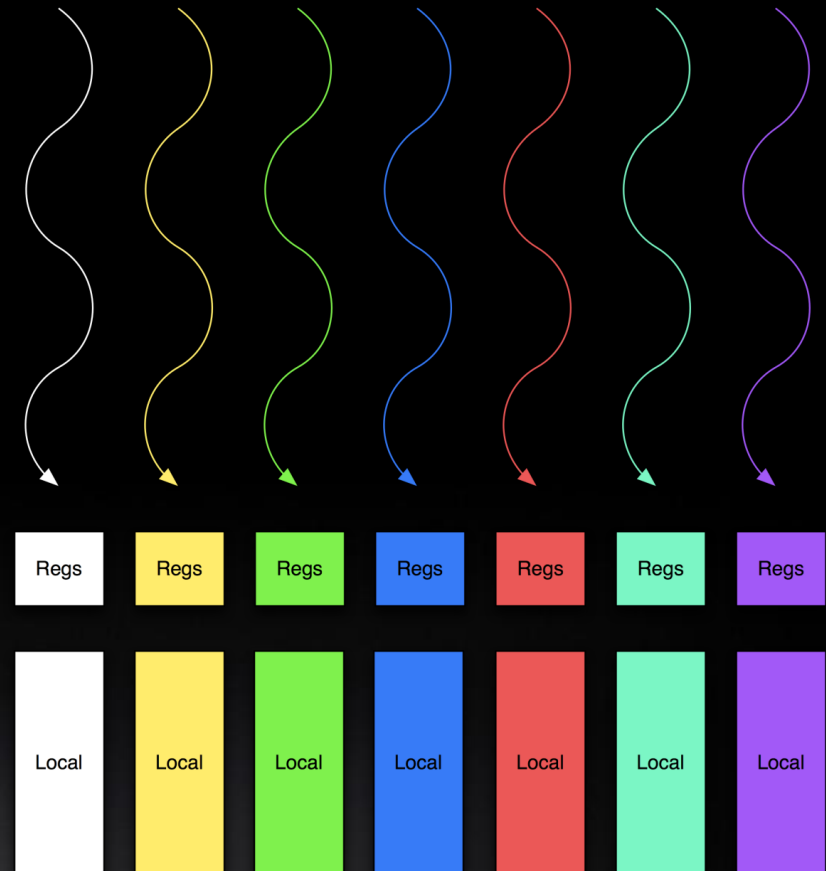
# MEMORY HIERARCHY

- Thread
  - Registers (“non addressable”)



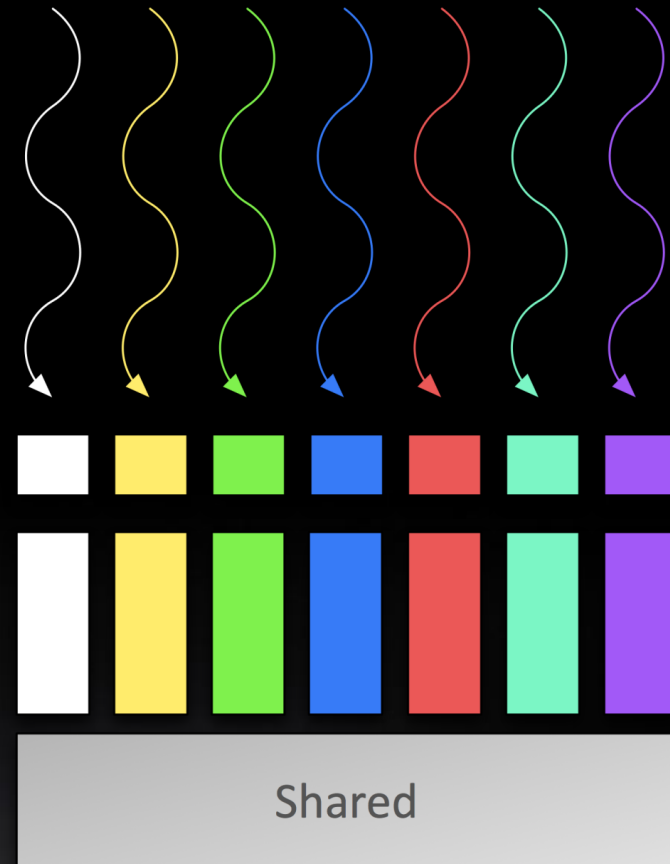
# MEMORY HIERARCHY

- Thread
  - **Registers** (“non addressable”)
  - **Local** memory (stack, register spills)



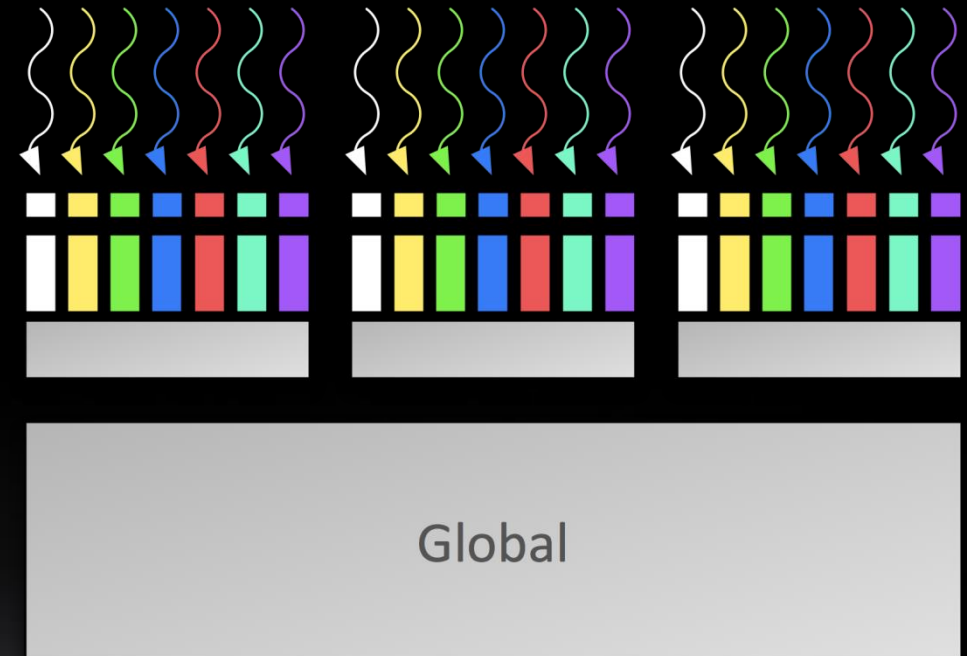
# MEMORY HIERARCHY

- Thread
  - Registers (“non addressable”)
  - Local memory
- Block of threads
  - Shared memory



# MEMORY HIERARCHY

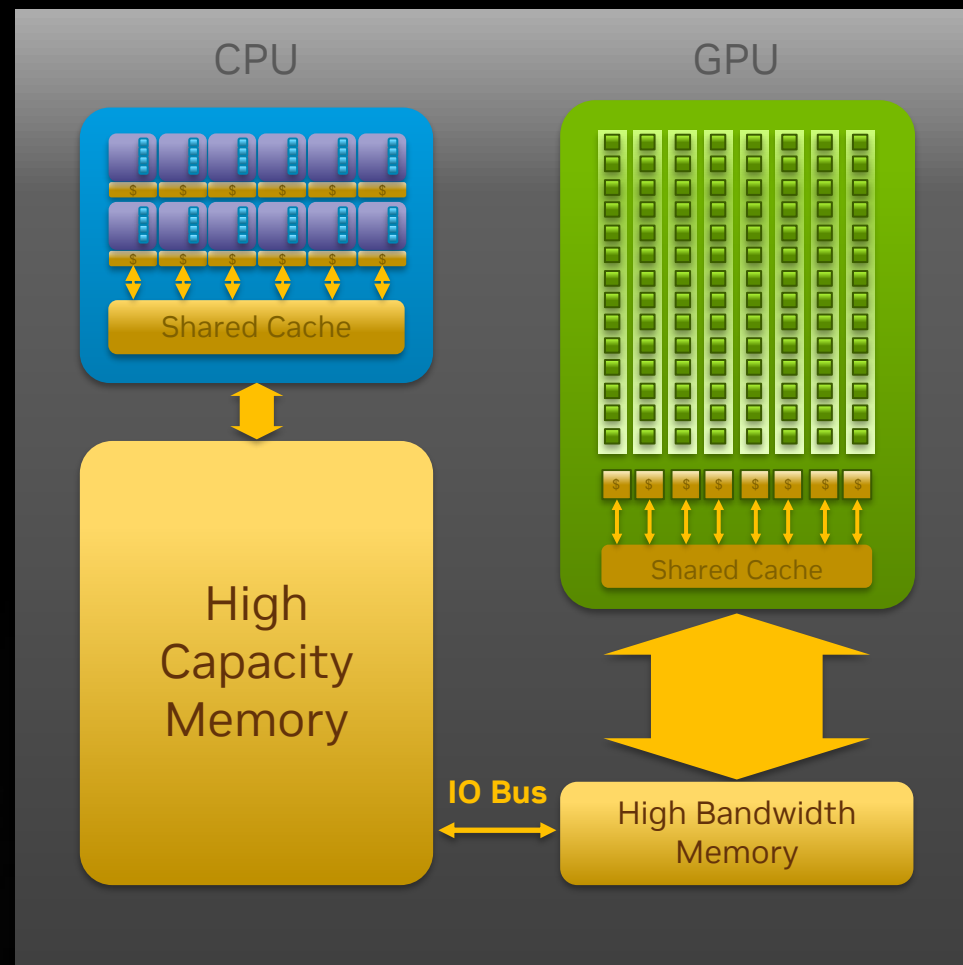
- Thread
  - **Registers** (“non addressable”)
  - **Local** memory
- Block of threads
  - **Shared** memory
  - **Uniform** registers
- All blocks
  - **Global** memory
  - **Host-Pinned** memory
  - **Constant** memory



# CPU + GPU

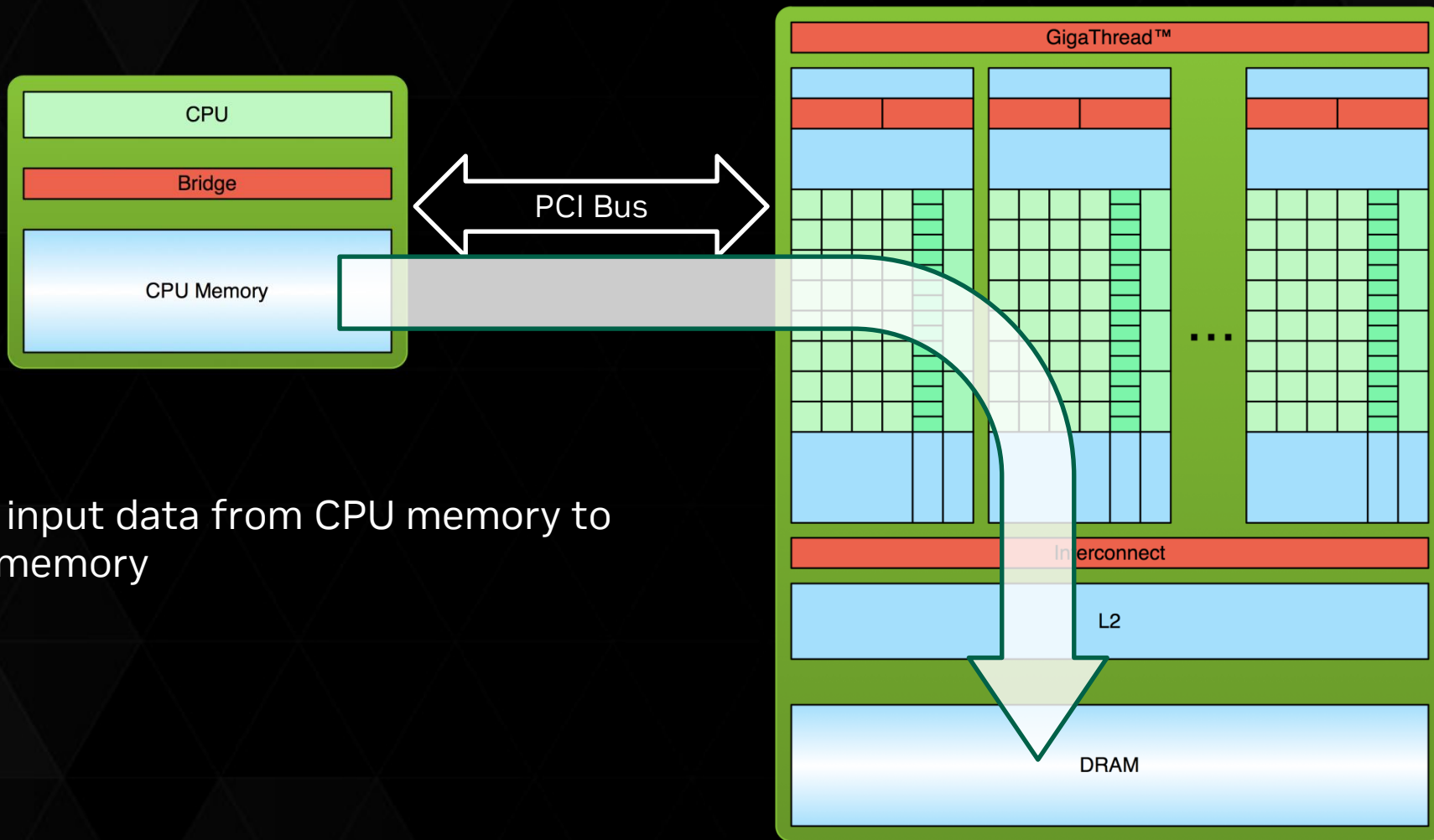
## Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCIe)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth. Grace-Hopper solves this :)
- The GPU cannot perform computation until the data is within its memory



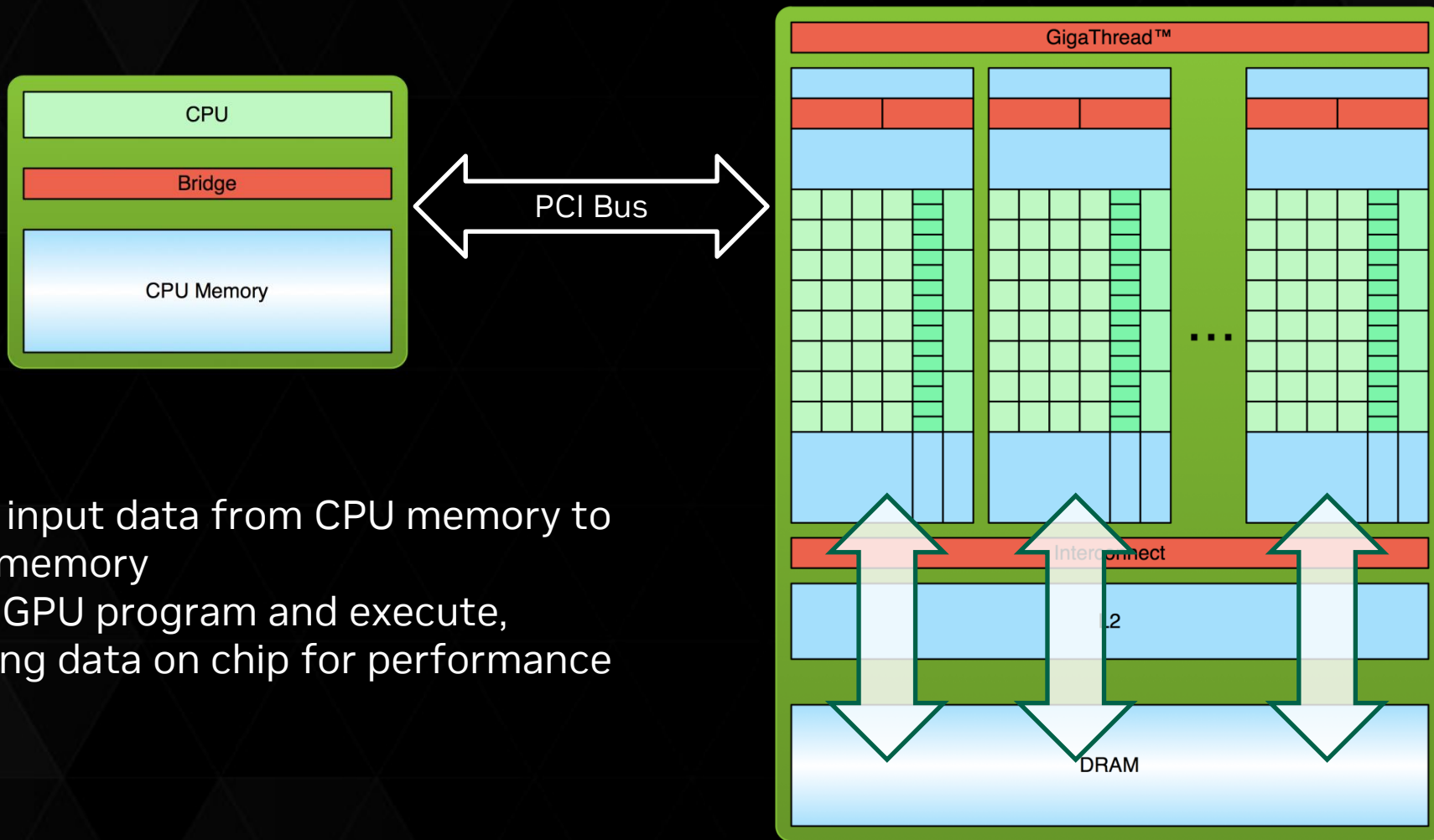


# PROCESSING FLOW – STEP 1

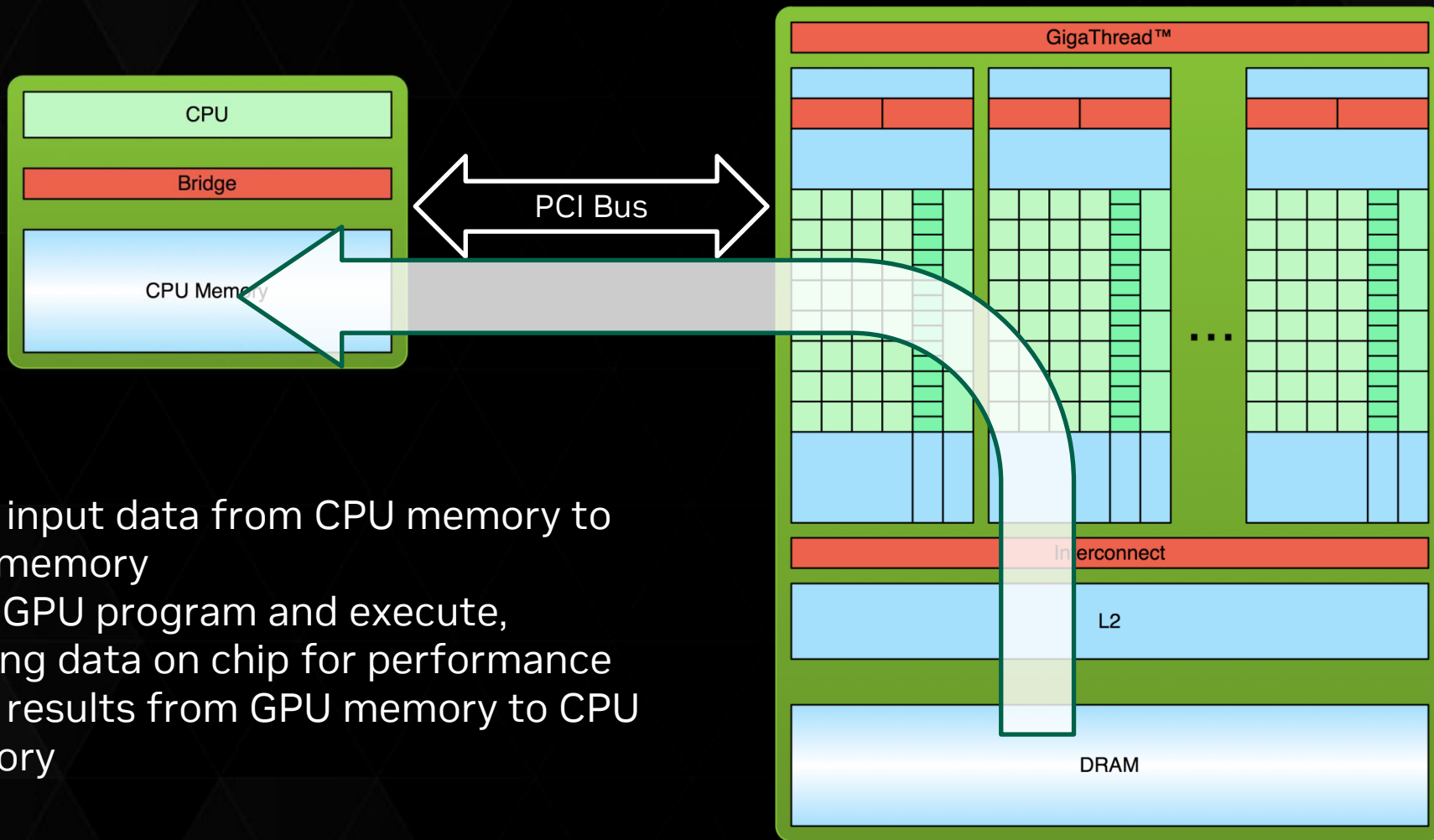


1. Copy input data from CPU memory to GPU memory

# PROCESSING FLOW – STEP 2



# PROCESSING FLOW – STEP 3



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



# CUDA-C++ INTRODUCTION

# WHAT IS CUDA?

- CUDA Architecture

- Expose general-purpose GPU computing as first-class capability
- Retain traditional DirectX/OpenGL graphics performance

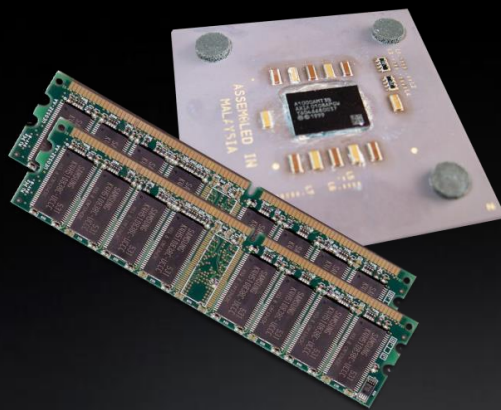
- CUDA C++

- Based on industry-standard C++
- A handful of language extensions to allow heterogeneous programs
- Straightforward APIs to manage devices, memory, etc.

# CUDA C++: THE BASICS

- Terminology
  - *Host* – The CPU and its memory (host memory)
  - *Device* – The GPU and its memory (device memory)

Host



Device



# HELLO, WORLD!

```
int main( void ) {  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- This basic program is just standard C++ that runs on the *host*
- NVIDIA's compiler (`nvcc`) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C++ is just C++!

# HELLO, WORLD! WITH DEVICE CODE

```
__global__ void kernel( void ) {  
    printf( "Hello, World!\n" );  
}  
  
int main( void ) {  
    kernel<<<1,1>>>();  
    cudaDeviceSynchronize();  
    return 0;  
}
```

- Two notable additions to the original “Hello, World!”



# HELLO, WORLD! WITH DEVICE CODE

```
int main( void ) {  
    kernel<<< 1, 1 >>>();  
    cudaDeviceSynchronize();  
    return 0;  
}
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Sometimes called a “kernel launch”
  - We’ll discuss the parameters inside the angle brackets later
- *cudaDeviceSynchronize()* make the CPU wait for the GPU
  - By default, the CPU-GPU relationship is **asynchronous**

# HELLO, WORLD! WITH DEVICE CODE

```
__global__ void kernel( void ) {  
}
```

- Keyword `__global__` in CUDA indicates that a function
  - Runs on the device
  - Called from host code
- `nvcc` splits source file into host and device components
  - NVIDIA's compiler handles device functions like `kernel()`
  - Standard host compiler handles host functions like `main()`
    - `g++`
    - Microsoft Visual C++

# NVIDIA HPC SDK

- Comprehensive suite of compilers, libraries, and tools used to GPU accelerate HPC modeling and simulation application
- The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting CUDA C and Fortran
  - The command to compile C++ code is '**nvc++**'
  - Same compiler will be used for host & device code → **more optimization opportunities**

```
nvc++ main.cu
```

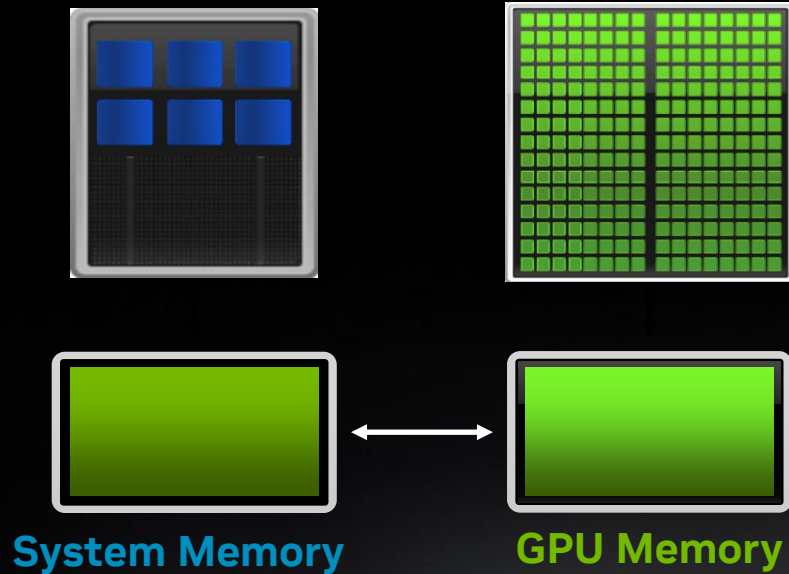
# A MORE COMPLEX EXAMPLE

- A simple kernel to add two integers:
- As before, `__global__` is a CUDA keyword meaning
  - `add ( )` will execute on the device ... so `a`, `b`, and `c` must point to device memory
  - `add ( )` will be called from the host
  - How to handle memory allocation ?

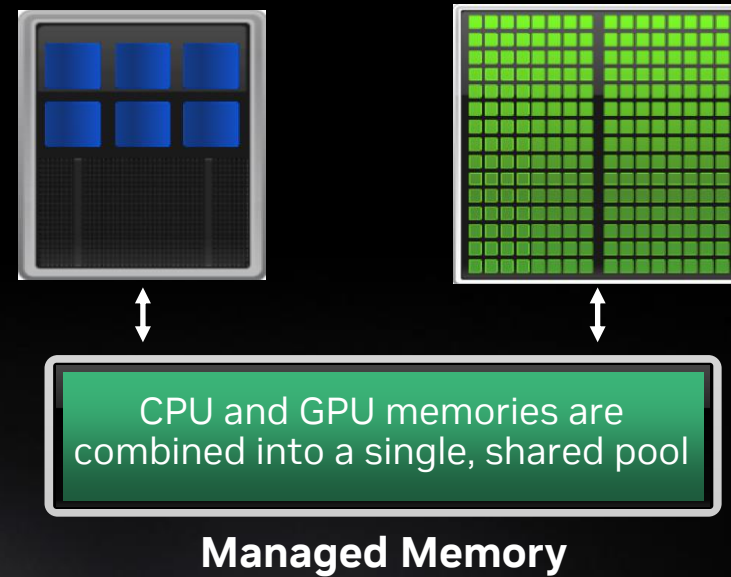
```
__global__ void add( int *a, int *b, int *c ) {  
    *c = *a + *b;  
}
```

# CUDA UNIFIED MEMORY

Simplified Developer Effort



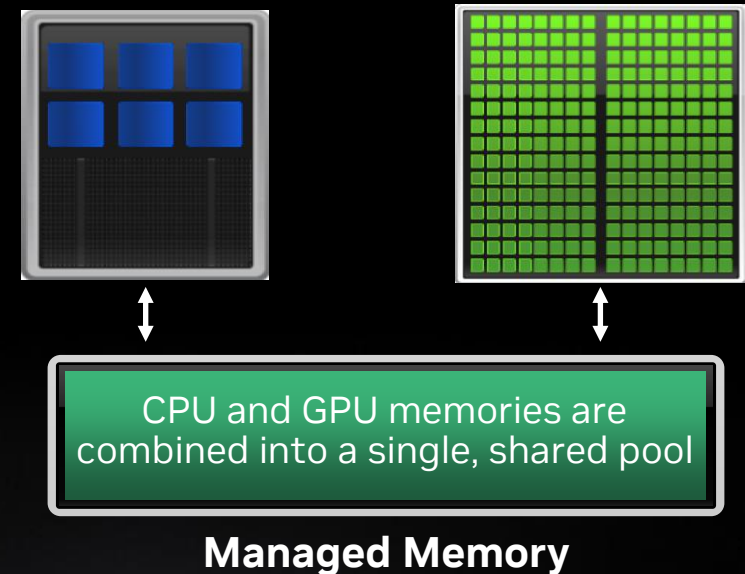
Commonly referred to as  
*"managed memory."*



# MANAGED MEMORY

## Limitations

- Disadvantage:
  - The programmer will almost always be able to get better performance by manually handling data transfers. Just Too Late
  - Memory allocation/deallocation takes longer with managed memory
- Advantage:
  - Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
  - This allows the developer to concentrate on parallelism and think about data movement as an optimization



# MEMORY MANAGEMENT

- Host and device memory are distinct entities
- Basic CUDA API for dealing with explicit device memory management
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`
- CUDA API for using Unified memory is
  - C API: `cudaMallocManaged()`, `cudaFree()`
  - \* For this session we will be making use of Unified Memory

# A MORE COMPLEX EXAMPLE: MAIN ( )

```
int main( void ) {  
    int *a, *b, *c;  
    int size = sizeof( int );  
  
    cudaMallocManaged( &a, size );  
    cudaMallocManaged( &b, size );  
    cudaMallocManaged( &c, size );  
  
    add<<< 1, 1 >>>( a, b, c );  
  
    cudaFree( a ); cudaFree( b );  
    cudaFree( c );  
    return 0;  
}
```



# PARALLEL PROGRAMMING IN CUDA

- But wait...GPU computing is about massive parallelism
- So how do we run code in parallel on the device?
- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( a, b, c );
```

↓ Number of blocks

```
add<<< N, 1 >>>( a, b, c );
```

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

# PARALLEL PROGRAMMING IN CUDA

- With `add()` running in parallel...let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a *block*
- Kernel can refer to its block's index with the variable
  - C: `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:
- By using `blockIdx.x` to index arrays, each block handles different indices

# PARALLEL PROGRAMMING IN CUDA

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

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

This is what runs in parallel on the device

# THREADS

- Terminology: A block can be split into parallel *threads*
- Let's change vector addition to use parallel threads instead of parallel blocks:
- We use `threadIdx.x` instead of `blockIdx.x` in `add()`

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

`add<<<N,1>>>>( a, b, c )`



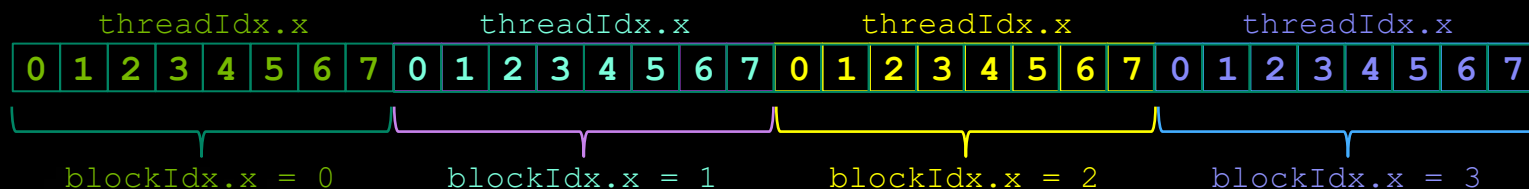
`add<<<1,N>>>>( a, b, c );`

# USING THREADS AND BLOCKS

- We've seen parallel vector addition using
  - Many blocks with 1 thread apiece
  - 1 block with many threads
- Let's adapt vector addition to use lots of *both* blocks and threads
- After using threads and blocks together, we'll talk about *why* threads
- First let's discuss data indexing...

# INDEXING ARRAYS WITH THREADS AND BLOCKS

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices
- To index array with 1 thread per entry (using 8 threads/block)



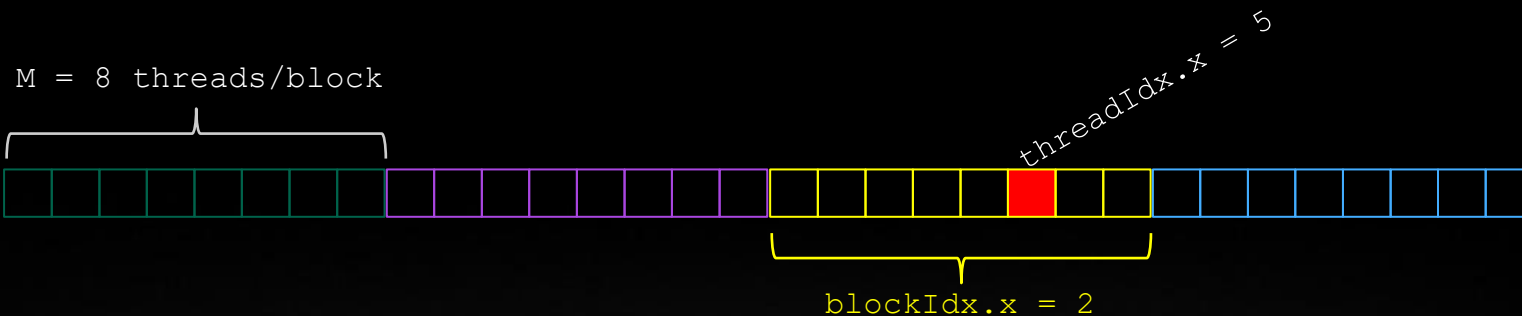
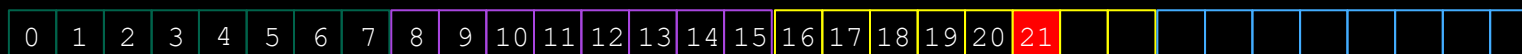
- If we have M threads/block, a unique array index for each entry given by

```
int index = threadIdx.x + blockIdx.x * M;
```

int index =           x           +           y           \* width;

# INDEXING ARRAYS: EXAMPLE

- In this example, the red entry would have an index of 21:



```
int index = threadIdx.x + blockIdx.x * M;  
  
          =      5      +      2      * 8;  
  
          = 21;
```

# ADDITION WITH THREADS AND BLOCKS

- The `blockDim.x` is a built-in variable for threads per block:

```
int index= threadIdx.x + blockIdx.x * blockDim.x
```

- So what changes in `main()` when we use both blocks and threads?

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



# PARALLEL ADDITION (BLOCKS/THREADS): MAIN ( )

```
#define N    (2048*2048)

#define THREADS_PER_BLOCK 512

add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( a, b, c );
```

# CUDA MANAGED MEMORY

## Usefulness

- Use modern C++17 parallel algorithm seamlessly on the GPU:
  - Do all allocations using managed memory
  - Call to all C++ algorithms (*reduce, scan, transform...*) with *std::par*
  - Compile with `nvc++` and the correct flags

```
nvc++ -stdpar=gpu program.cu -o program
```

# CUDA MANAGED MEMORY

## Usefulness

```
nvc++ -stdpar=gpu program.cu -o program
```

```
int main(int argc, char* argv[]) {  
    std::vector<int> vec(...) // Allocated on the CPU  
    std::reduce(std::par, vec.begin() , vec.end()); // Automatically copied and computed on the GPU  
  
    cpu_computation(vec); // Rest of the complex CPU code kept unchanged  
}
```

# CUDA SPEEDUP : cuOpt

Traveling Salesman Problem : **shortest path to visit a set of points**

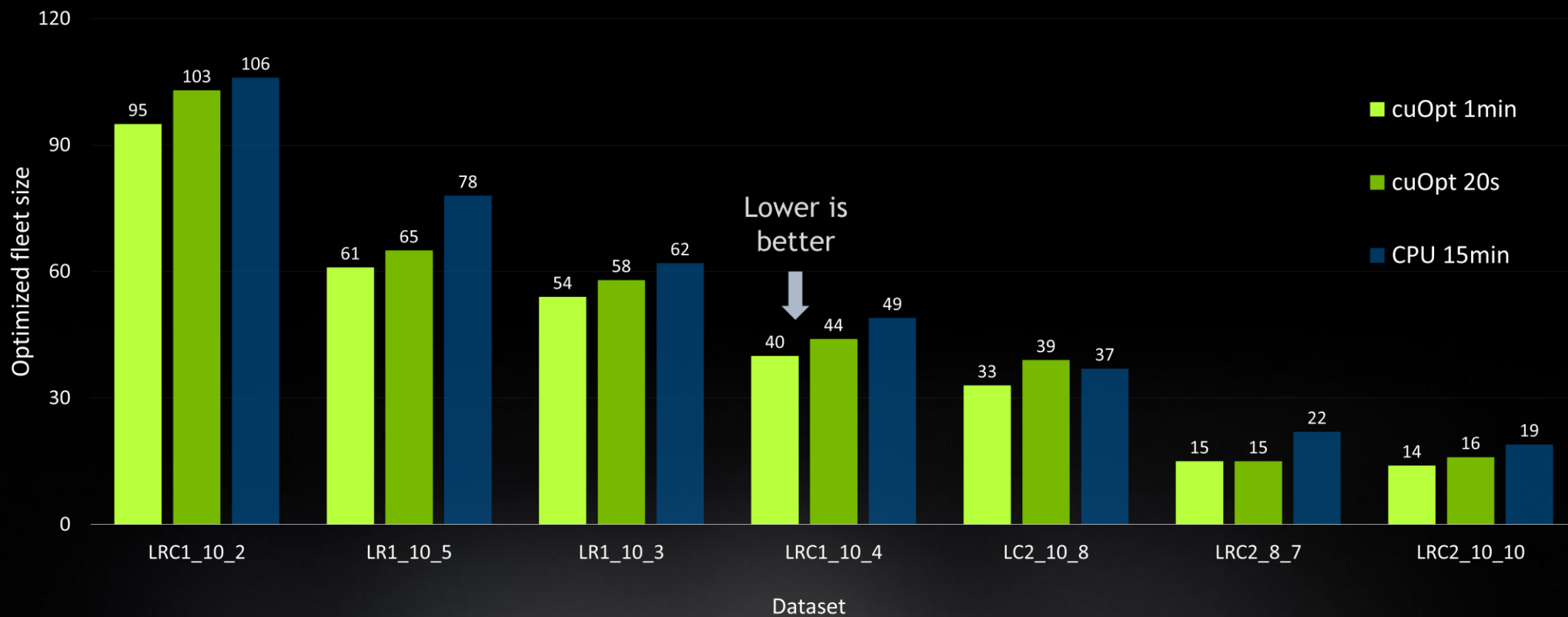
Vehicle routing problem (VRP) : multiple vehicles, weight & time constraints...

NP-Hard problem with factorial complexity :

**100 cities → more paths to test than the number of atoms in the universe**



# CUDA SPEEDUP



# REFERENCES

[Blog | Accelerating standard c with GPUs using stdpar](#)

[Blog | An even easier introduction CUDA/](#)

# Have fun practicing with CUDA

Free AI Technical Training A Few Clicks Away

Elevate your skills with free AI technical training. For a limited time, we are offering a complimentary self-paced training course to new members of our [NVIDIA Developer Program](#).

DLI course will allow you to **practice easily and without needing a GPU**.

Getting Started with Accelerated Computing in CUDA C/C++

Fundamentals of Accelerated Computing With CUDA Python

[Self-paced course | An even easier introduction to CUDA](#)



# THANK YOU

Any questions?

