TDT4200 Parallel Programming

Bart van Blokland

Lecture 9

&group

Last Time:

Im in ur CPU

Nao in ur GPU

Two important things:

- Differences between a CPU and GPU
- Threads, Warps, and the Streaming Multiprocessor

The CPU can be faster than the GPU.

The CPU can be faster than the GPU.

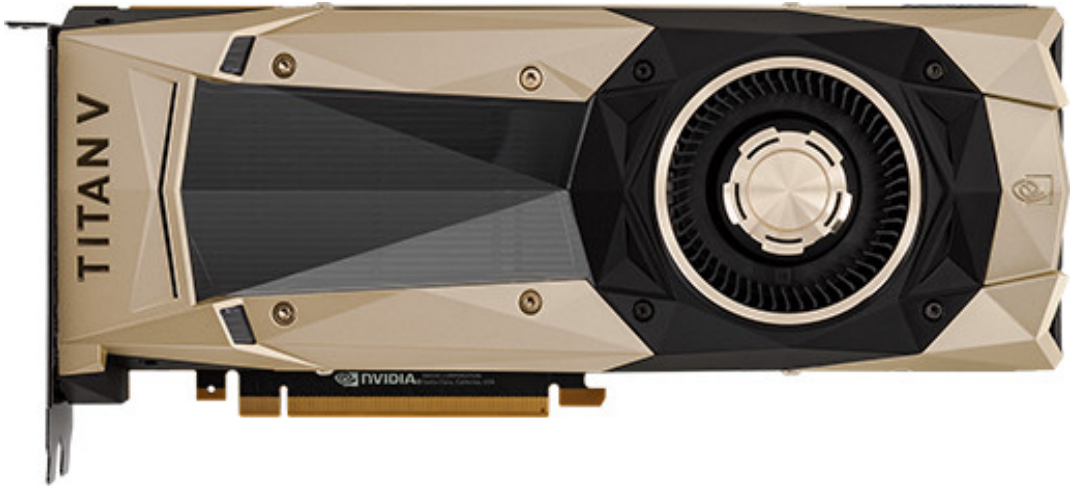
Professional sprinter:

+ Covers small distances super quickly
- Takes a while to run 1000 km

City Marathon:

- Individual people are kind of slow
+ Combined distance of runners sums up to 1000km very quickly

The CPU can be faster than the GPU.

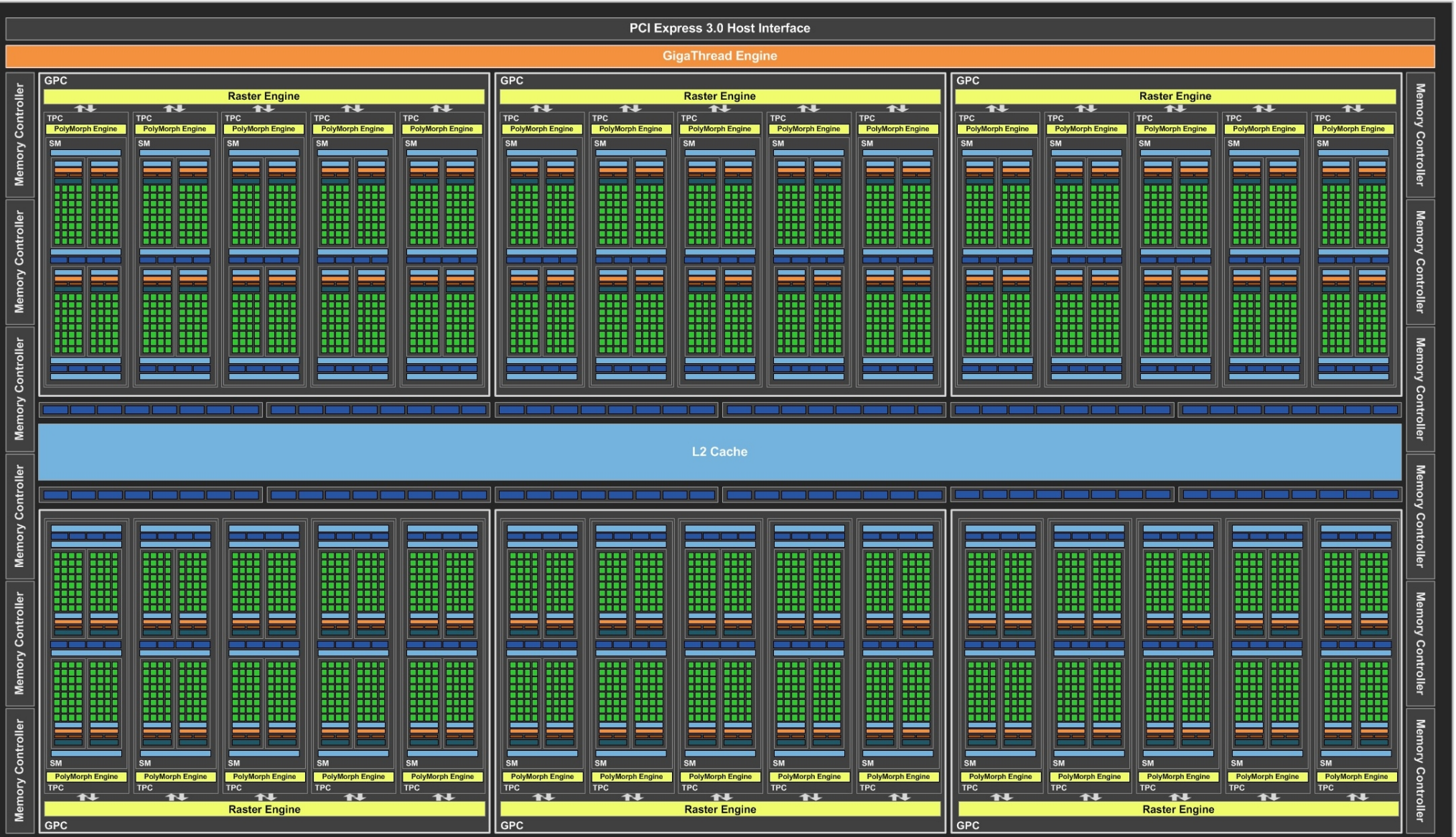


CPU:

+ Single thread performance is amazing
- Throughput could be higher

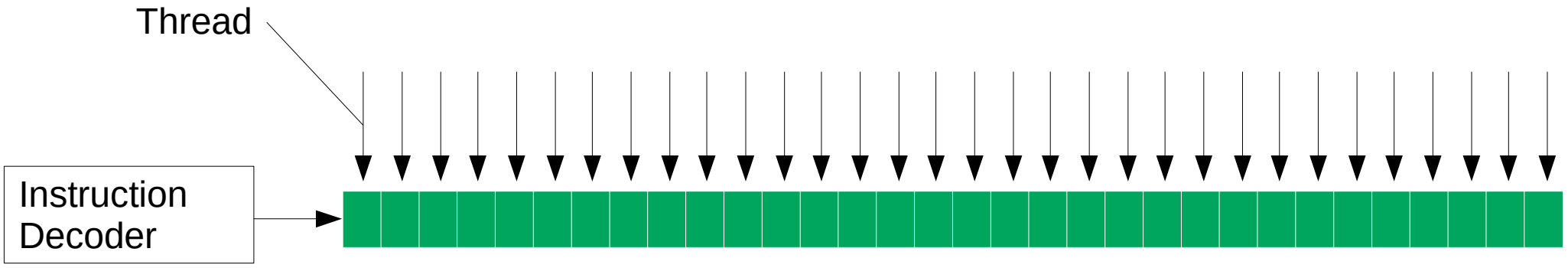
GPU:

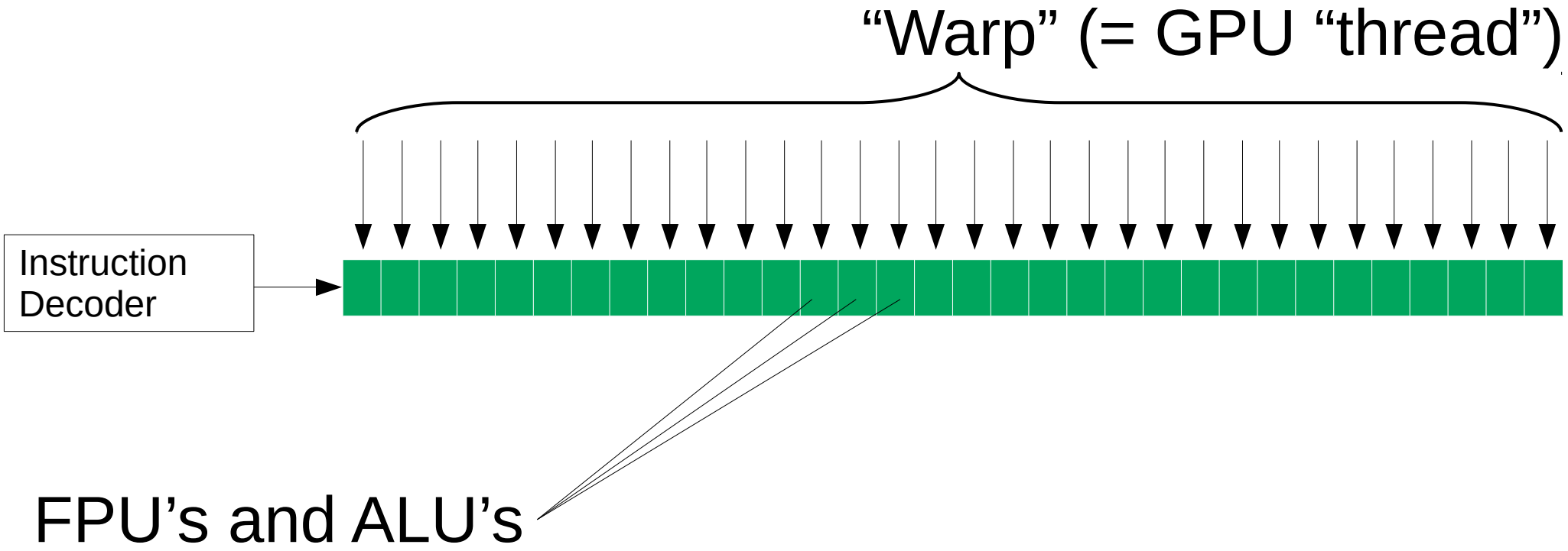
- Single thread performance is terrible
+ High throughput


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
GPC
Raster Engine
TPC
PolyMorph Engine
SM
L2 Cache


SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory

Roughly a single core CPU with a bunch of ALU's and FPU's!


Thread
Instruction
Decoder

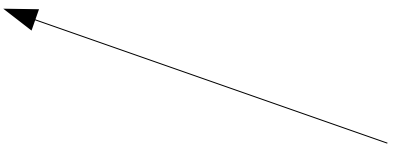

“Warp” (= GPU “thread”)
Instruction Decoder
FPU’s and ALU’s

Today:

Let’s write a CUDA program!

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```



“Kernel”

```
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

The CUDA API distinguishes the CPU from the GPU:

Host: CPU side

Device: GPU side

GPU

```
__device__ void doSomething();

__global__ void mainKernel();
```

CPU

```
__host__ void cpuFunction();
```

Host and device functions cannot call each other.

They live a lonely life.

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Kernel launches are asynchronous

→ CPU can do other stuff after kicking off a job

.. But needs to wait if it requires the results.

```
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Thread Hierarchy

Thread Hierarchy

```
for(int index = 0; index < 1000; index++) {
    bigArray[index] = computeSomething();
}
```

Thread Hierarchy

```
for(int index = 0; index < 1000; index++) {
    bigArray[index] = computeSomething();
}
```

```
int index = threadIndex;
bigArray[index] = computeSomething();
```

x 1000

Done, right?

Note quite.

There is no time to run a scheduling algorithm.

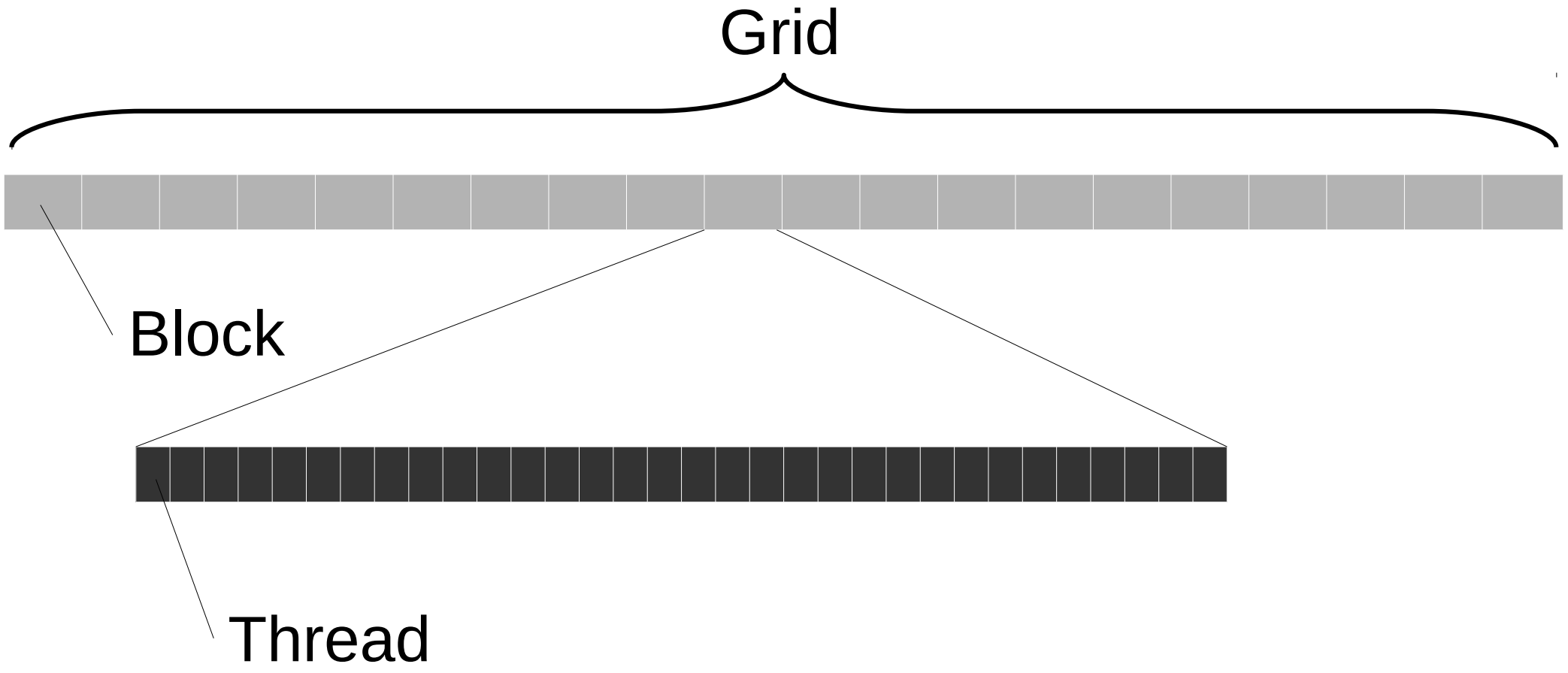
→ All scheduling runs in hardware

There is no time to run a scheduling algorithm.

→ All scheduling runs in hardware

Compromise: threads are grouped in “Blocks”.

Blocks contain a number of warps, and are scheduled on


Grid
Block
Thread

Blocks and grids can be launched in multiple dimensions:

```
blockIdx.x          threadIdx.x
blockIdx.y          threadIdx.y
blockIdx.z          threadIdx.z
```

Blocks and grids can be launched in multiple dimensions:

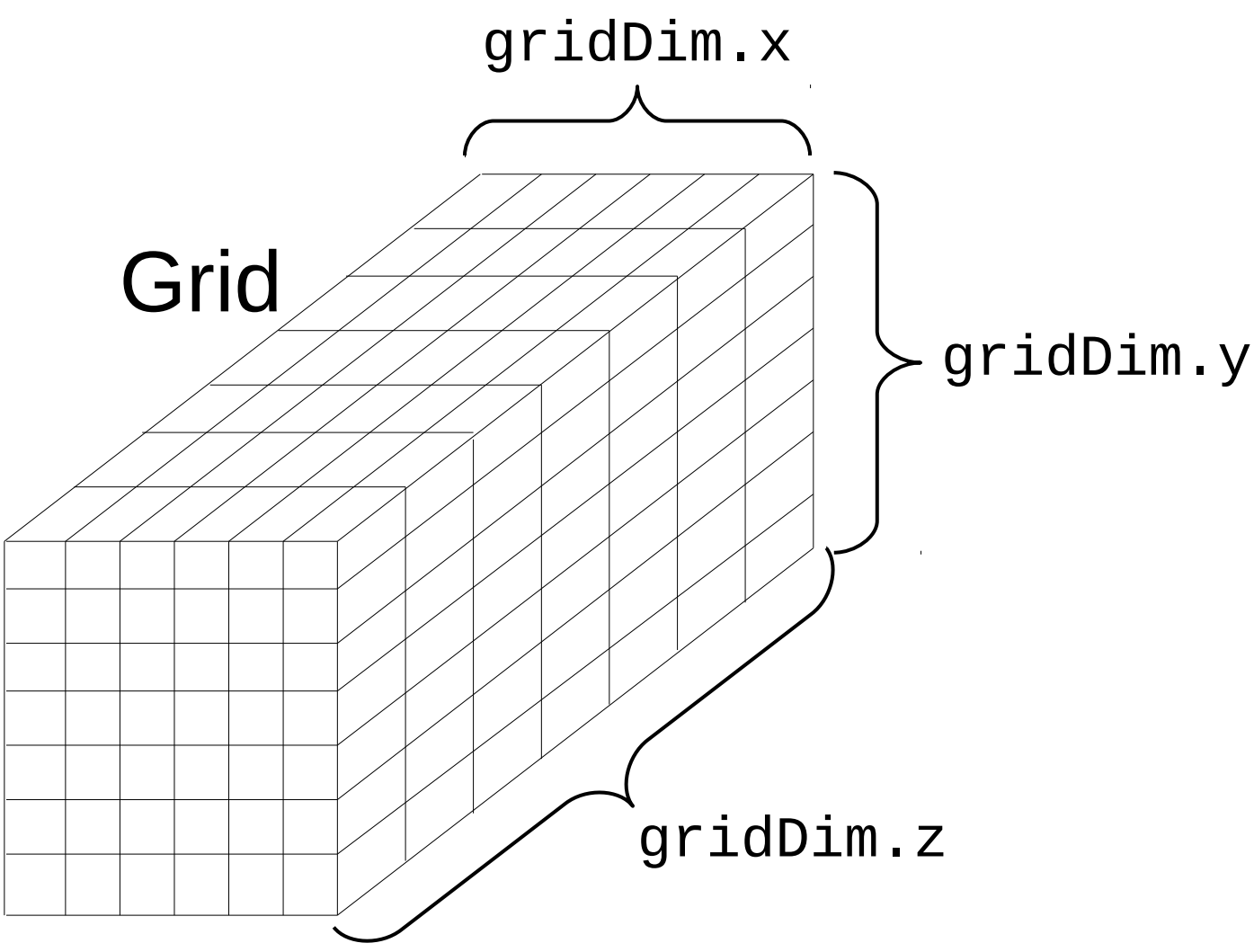
```
blockIdx.x
blockIdx.y
blockIdx.z
```

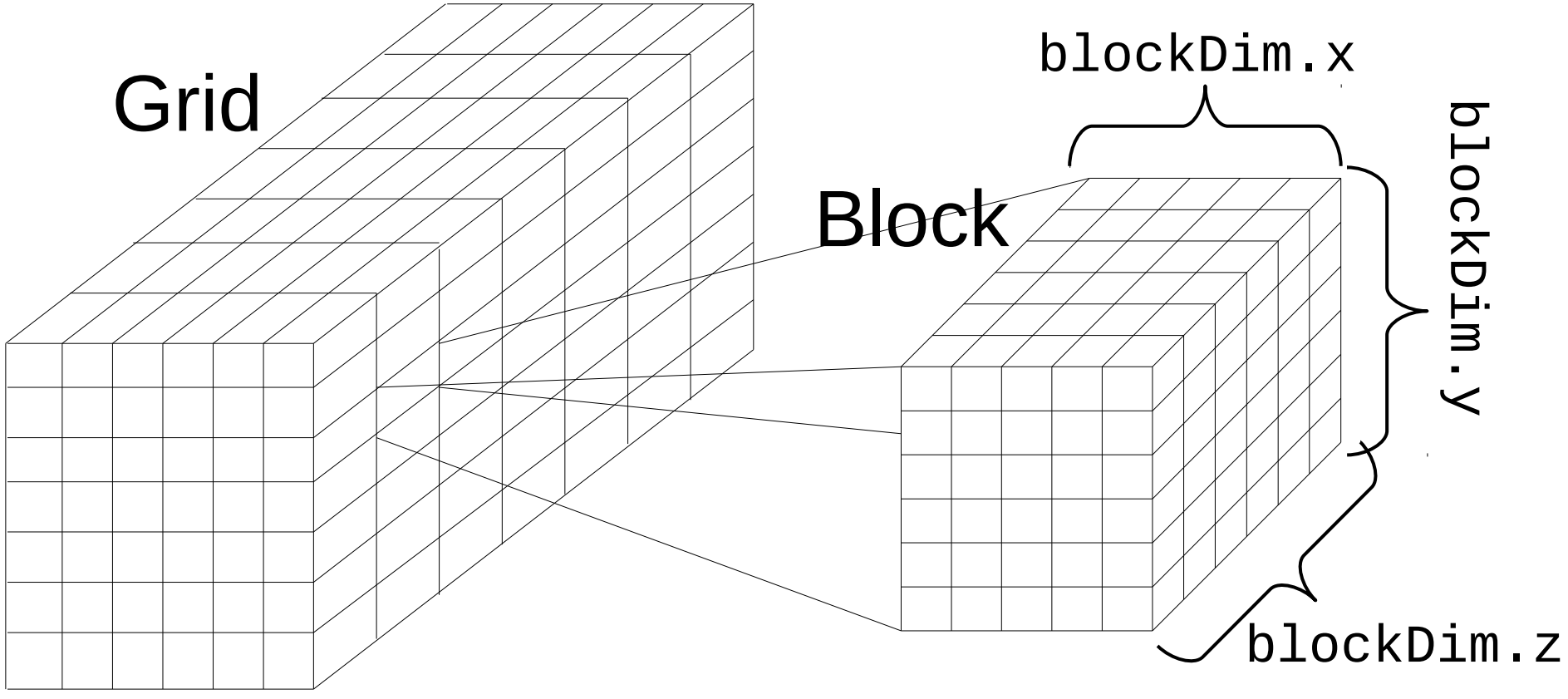
```
threadIdx.x
threadIdx.y
threadIdx.z
```

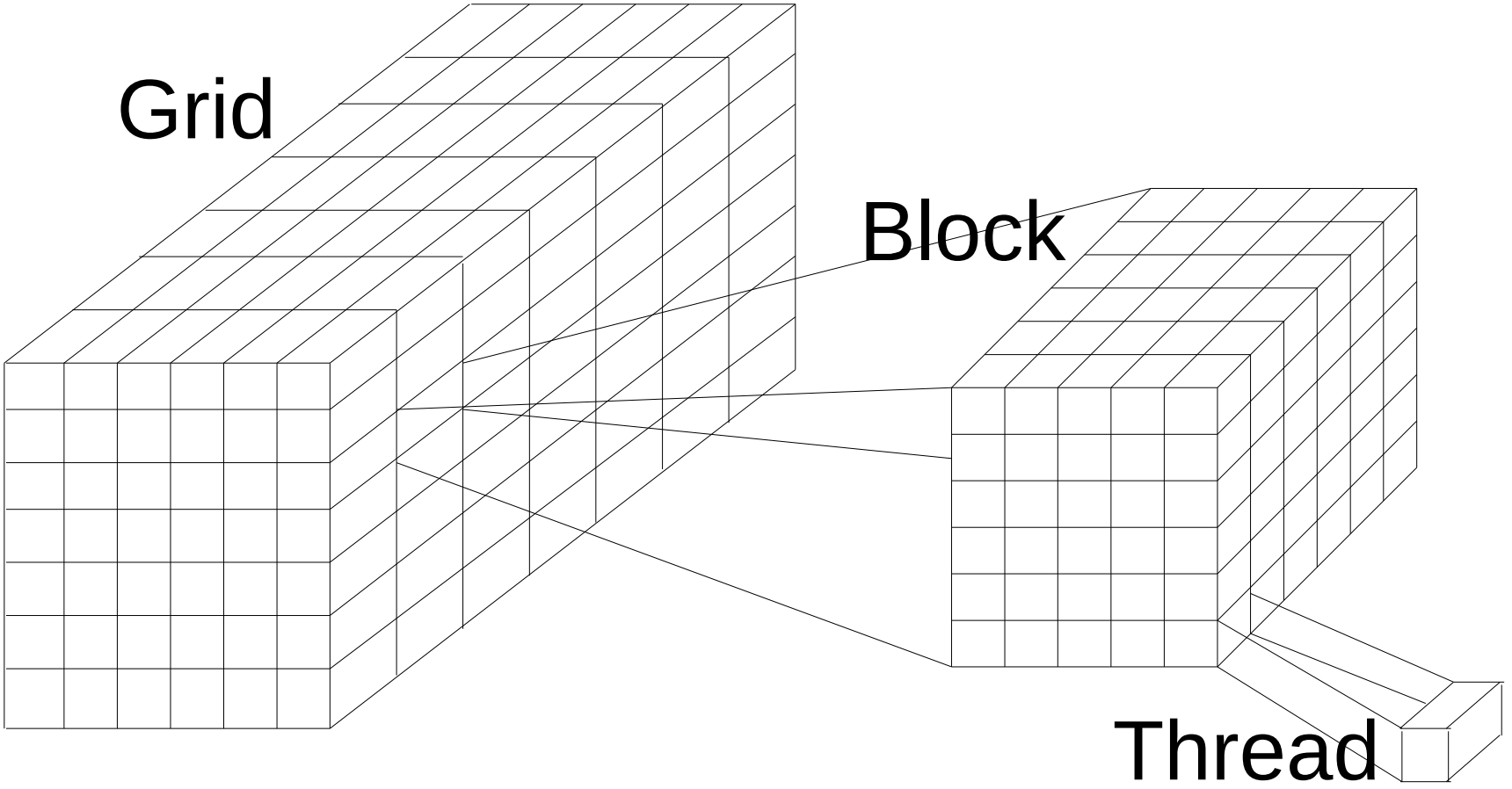
You can request the block and grid sizes at runtime:

```
gridDim.x
gridDim.y
gridDim.z
```

```
blockDim.x
blockDim.y
blockDim.z
```


gridDim.x
Grid
gridDim.y
gridDim.z

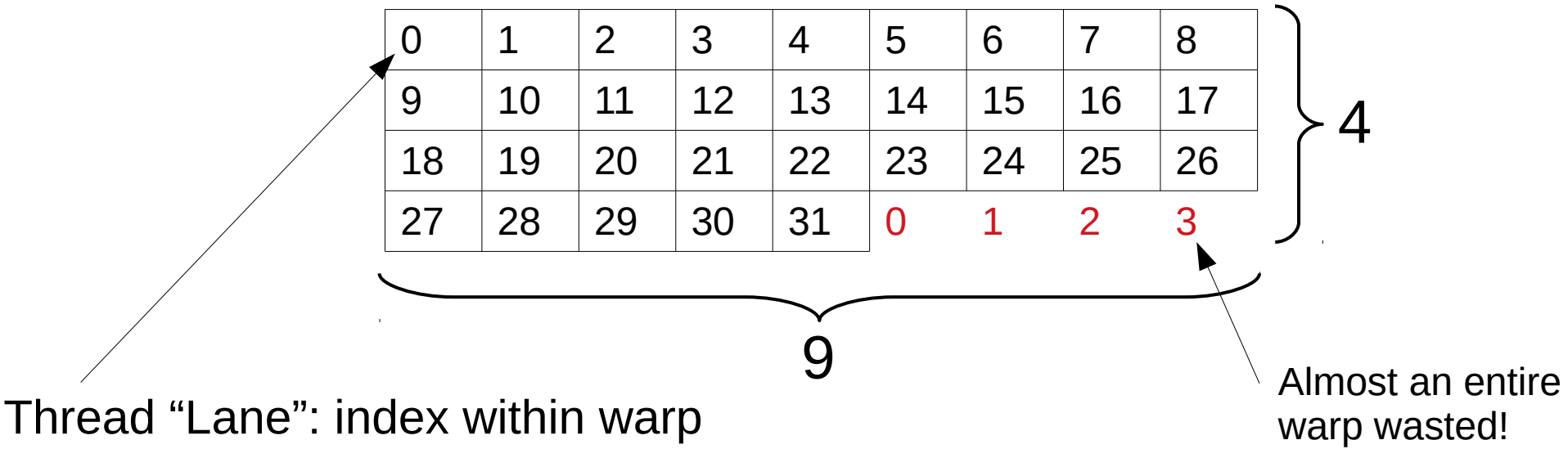

Grid
Block
blockDim.x
blockDim.y
blockDim.z


Grid
Block
Thread

Caveats

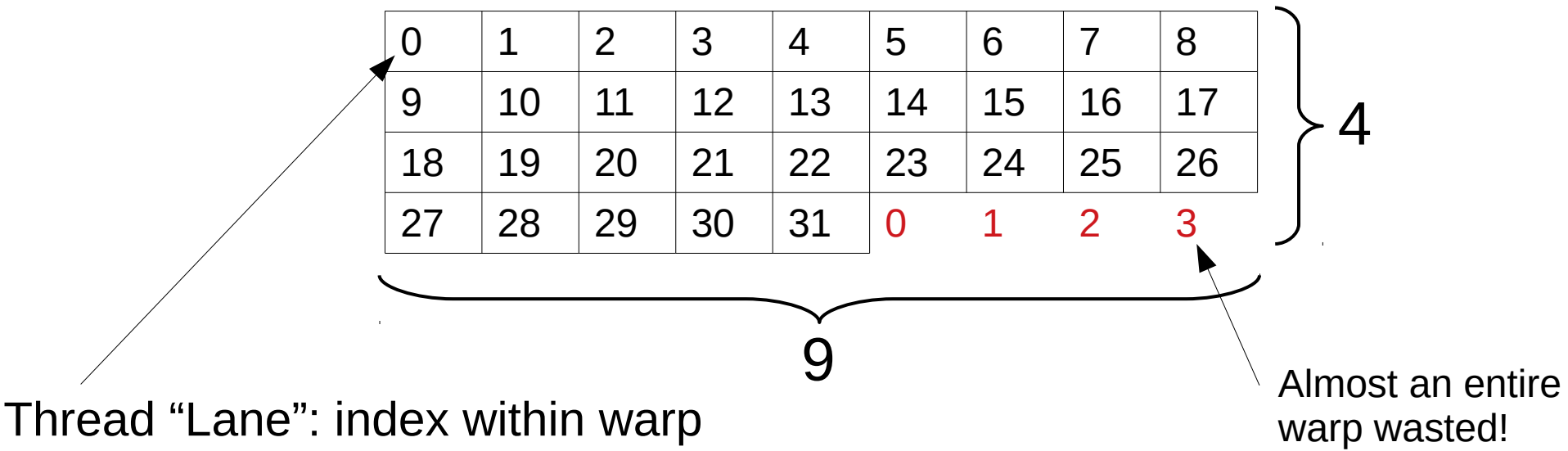
Caveats

Threads in a block are “flattened”, and chopped up into warps.



Caveats

Threads in a block are “flattened”, and chopped up into warps.



blockDim.x * blockDim.y * blockDim.z
should normally be a multiple of 32!

Caveats

Limits

For recent cards:

- Max grid size: $[2^{31} - 1, 65535, 65535]$
- Max block size: [1024, 1024, 64]

- Max threads per block: 1024

- Max number of warps per SM: 64
- Max number of threads per SM: 2048

Launching a kernel launches a grid of blocks

Each block contains threads,
rounded up to the nearest multiple of 32.

Blocks are assigned to SM’s.

Warps execute on SM’s.

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

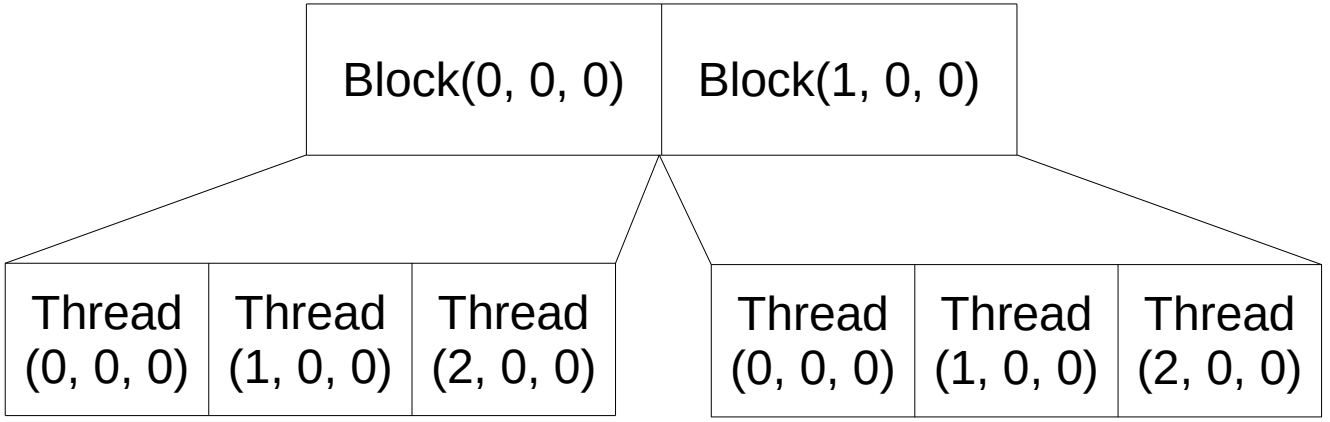
```
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   dim3 grid(2, 1, 1);
   dim3 block(3, 1, 1);
   runMeOnGPU<<<grid, block>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   dim3 grid(2, 1, 1);
   dim3 block(3, 1, 1);
   runMeOnGPU<<<grid, block>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Small problem..

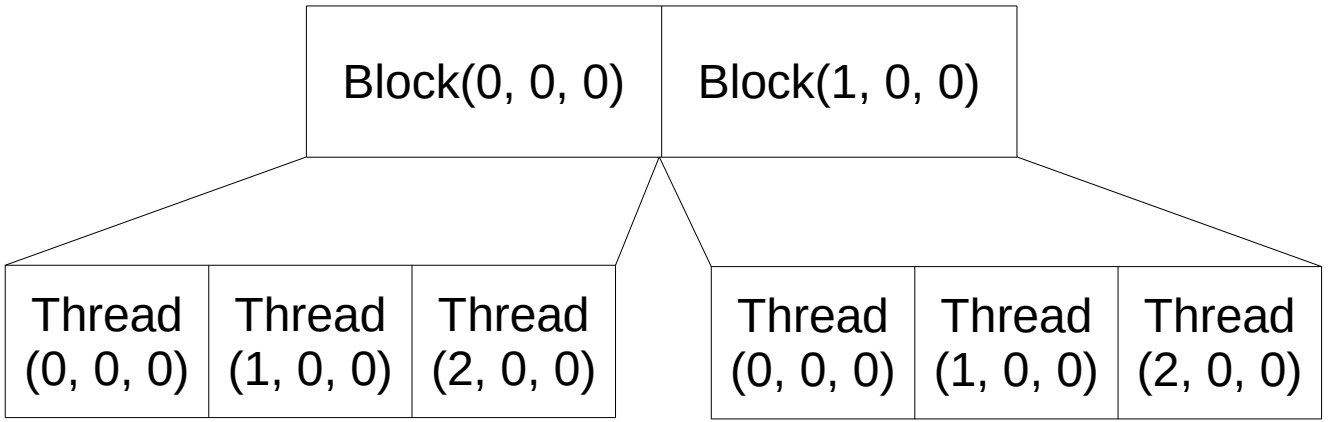
```
I'm thread 0!
I'm thread 1!
I'm thread 2!
I'm thread 0!
I'm thread 1!
I'm thread 2!
```

Fortunately, all blocks have equal sizes




```
dim3 grid(2, 1, 1);
dim3 block(3, 1, 1);
```

Fortunately, all blocks have equal sizes




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

```
dim3 grid(2, 1, 1);
dim3 block(3, 1, 1);
```

Let’s practice this a bit!

For the next problems, determine:

- the launch parameters (block and grid size)

- how each thread computes its index.

You'd like to process a 1D array

What grid and block sizes do you use?
How do you compute the pixel index?

Limits

- Max grid size: $[2^{31} - 1, 65535, 65535]$
- Max block size: [1024, 1024, 64]
- Max threads per block: 1024
- Max number of warps per SM: 64
- Max number of threads per SM: 2048

Additional info

- Each thread processes exactly one element in the array

```
gridDim.x   blockIdx.x          blockDim.x   threadIdx.x
gridDim.y   blockIdx.y          blockDim.y   threadIdx.y
gridDim.z   blockIdx.z          blockDim.z   threadIdx.z
```

You'd like to process a 2D image of an arbitrary size.

What grid and block sizes do you use?
How do you compute the pixel index?

Limits

- Max grid size: [$2^{31} - 1$, 65535, 65535]
- Max block size: [1024, 1024, 64]
- Max threads per block: 1024
- Max number of warps per SM: 64
- Max number of threads per SM: 2048

Additional info

- Each thread processes exactly one pixel

```
gridDim.x   blockIdx.x          blockDim.x   threadIdx.x
gridDim.y   blockIdx.y          blockDim.y   threadIdx.y
gridDim.z   blockIdx.z          blockDim.z   threadIdx.z
```

You'd like to process a sequence of 16x16 pixel images
What grid and block sizes do you use?
How do you compute the image index and pixel coord?

Limits

- Max grid size: [$2^{31} - 1$, 65535, 65535]
- Max block size: [1024, 1024, 64]
- Max threads per block: 1024
- Max number of warps per SM: 64
- Max number of threads per SM: 2048

Additional info

- Each thread processes exactly one pixel
- Images are stored sequentially on a row by row basis.

```
gridDim.x   blockIdx.x        blockDim.x   threadIdx.x
gridDim.y   blockIdx.y        blockDim.y   threadIdx.y
gridDim.z   blockIdx.z        blockDim.z   threadIdx.z
```

You'd like to process a sequence of 14x14 pixel images
What grid and block sizes do you use?
How do you compute the image index and pixel coord?

Limits

- Max grid size: [$2^{31} - 1$, 65535, 65535]
- Max block size: [1024, 1024, 64]
- Max threads per block: 1024
- Max number of warps per SM: 64
- Max number of threads per SM: 2048

Additional info

- Each thread processes exactly one pixel
- Images are stored sequentially on a row by row basis.

```
gridDim.x   blockIdx.x        blockDim.x   threadIdx.x
gridDim.y   blockIdx.y        blockDim.y   threadIdx.y
gridDim.z   blockIdx.z        blockDim.z   threadIdx.z
```

```
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

But how do we compile it?

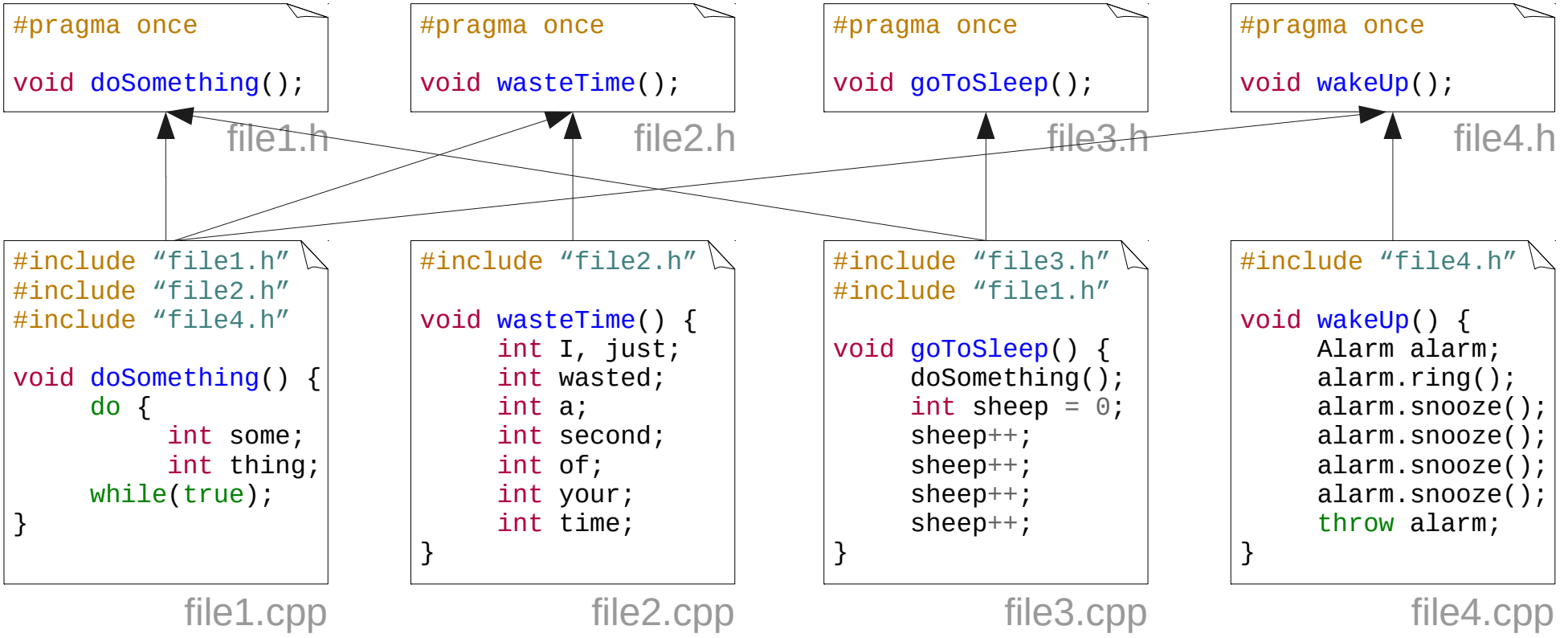
```
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

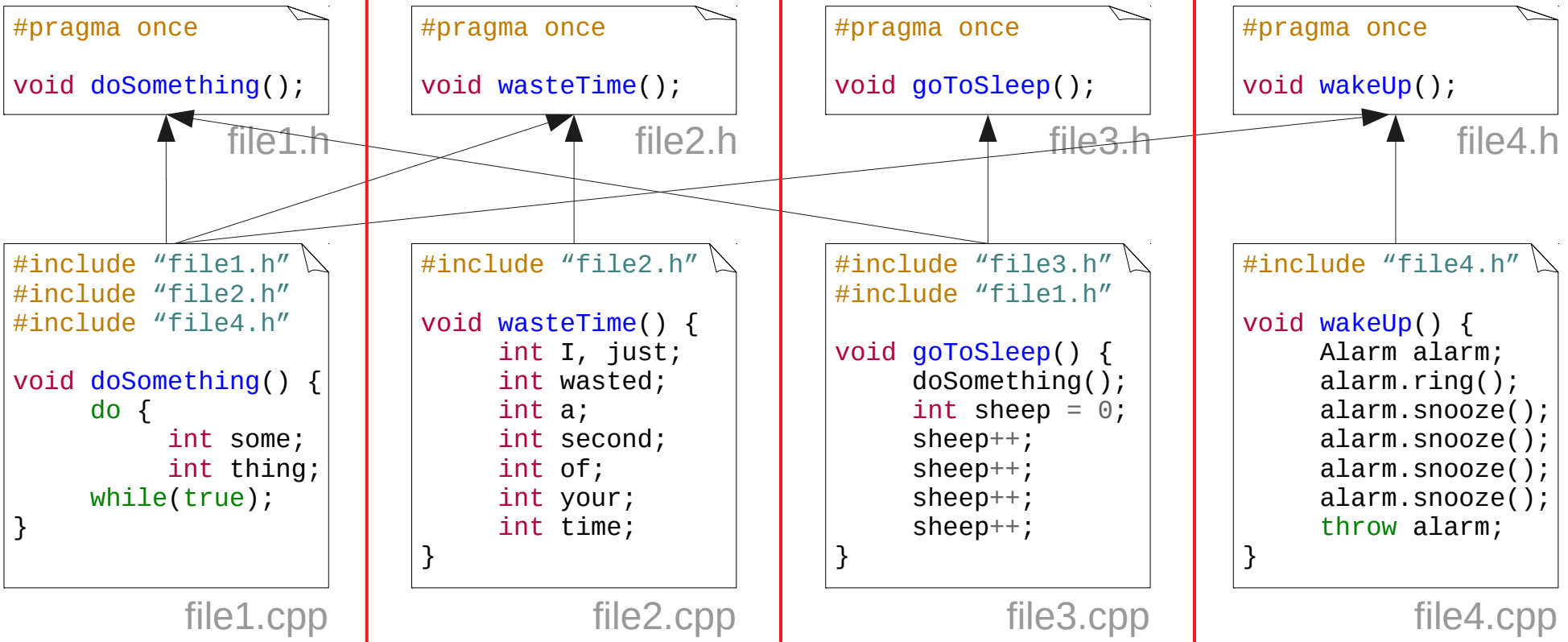
NVCC (the CUDA Compiler)

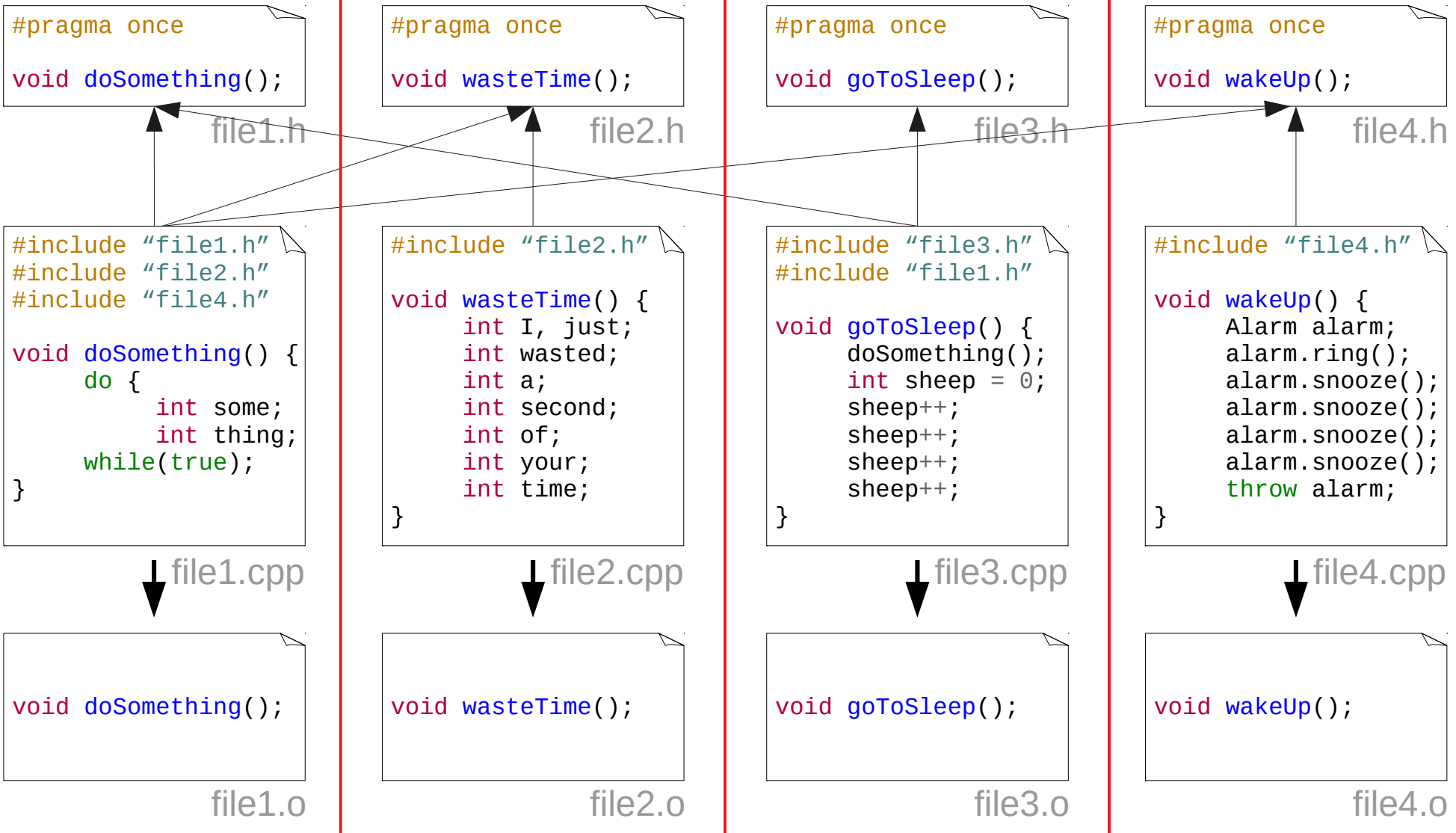
```
nvcc sample.cu -o runongpu
```

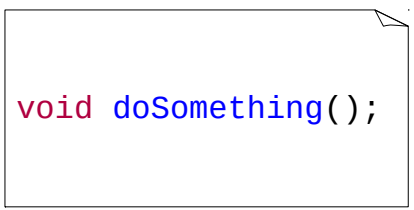
```
nvcc sample.cu -o runongpu
```

What does GCC
have to do with that?

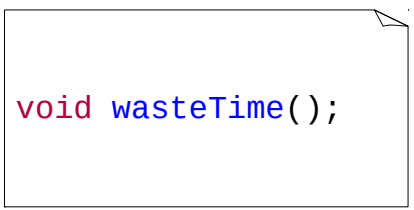

#pragma once
void doSomething();
file1.h
#pragma once
void wasteTime();
file2.h
#pragma once
void goToSleep();
file3.h
#pragma once
void wakeUp();
file4.h
#include “file1.h”
#include “file2.h”
#include “file4.h”
void doSomething() {
do {
int some;
int thing;
while(true);
}
file1.cpp
#include “file2.h”
void wasteTime() {
int I, just;
int wasted;
int a;
int second;
int of;
int your;
int time;
}
file2.cpp
#include “file3.h”
#include “file1.h”
void goToSleep() {
doSomething();
int sheep = 0;
sheep++;
sheep++;
sheep++;
sheep++;
}
file3.cpp
#include “file4.h”
void wakeUp() {
Alarm alarm;
alarm.ring();
alarm.snooze();
alarm.snooze();
alarm.snooze();
alarm.snooze();
throw alarm;
}
file4.cpp


#pragma once
void doSomething();
file1.h
#pragma once
void wasteTime();
file2.h
#pragma once
void goToSleep();
file3.h
#pragma once
void wakeUp();
file4.h
#include “file1.h”
#include “file2.h”
#include “file4.h”
void doSomething() {
do {
int some;
int thing;
while(true);
}
file1.cpp
#include “file2.h”
void wasteTime() {
int I, just;
int wasted;
int a;
int second;
int of;
int your;
int time;
}
file2.cpp
#include “file3.h”
#include “file1.h”
void goToSleep() {
doSomething();
int sheep = 0;
sheep++;
sheep++;
sheep++;
sheep++;
}
file3.cpp
#include “file4.h”
void wakeUp() {
Alarm alarm;
alarm.ring();
alarm.snooze();
alarm.snooze();
alarm.snooze();
alarm.snooze();
throw alarm;
}
file4.cpp

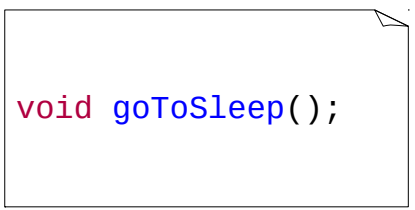

#pragma once
void doSomething();
file1.h
#pragma once
void wasteTime();
file2.h
#pragma once
void goToSleep();
file3.h
#pragma once
void wakeUp();
file4.h
#include “file1.h”
#include “file2.h”
#include “file4.h”
void doSomething() {
do {
int some;
int thing;
while(true);
}
file1.cpp
#include “file2.h”
void wasteTime() {
int I, just;
int wasted;
int a;
int second;
int of;
int your;
int time;
}
file2.cpp
#include “file3.h”
#include “file1.h”
void goToSleep() {
doSomething();
int sheep = 0;
sheep++;
sheep++;
sheep++;
sheep++;
}
file3.cpp
#include “file4.h”
void wakeUp() {
Alarm alarm;
alarm.ring();
alarm.snooze();
alarm.snooze();
alarm.snooze();
alarm.snooze();
throw alarm;
}
file4.cpp
void doSomething();
file1.o
void wasteTime();
file2.o
void goToSleep();
file3.o
void wakeUp();
file4.o



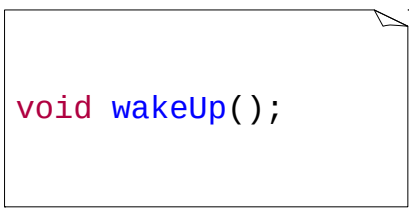
file1.o



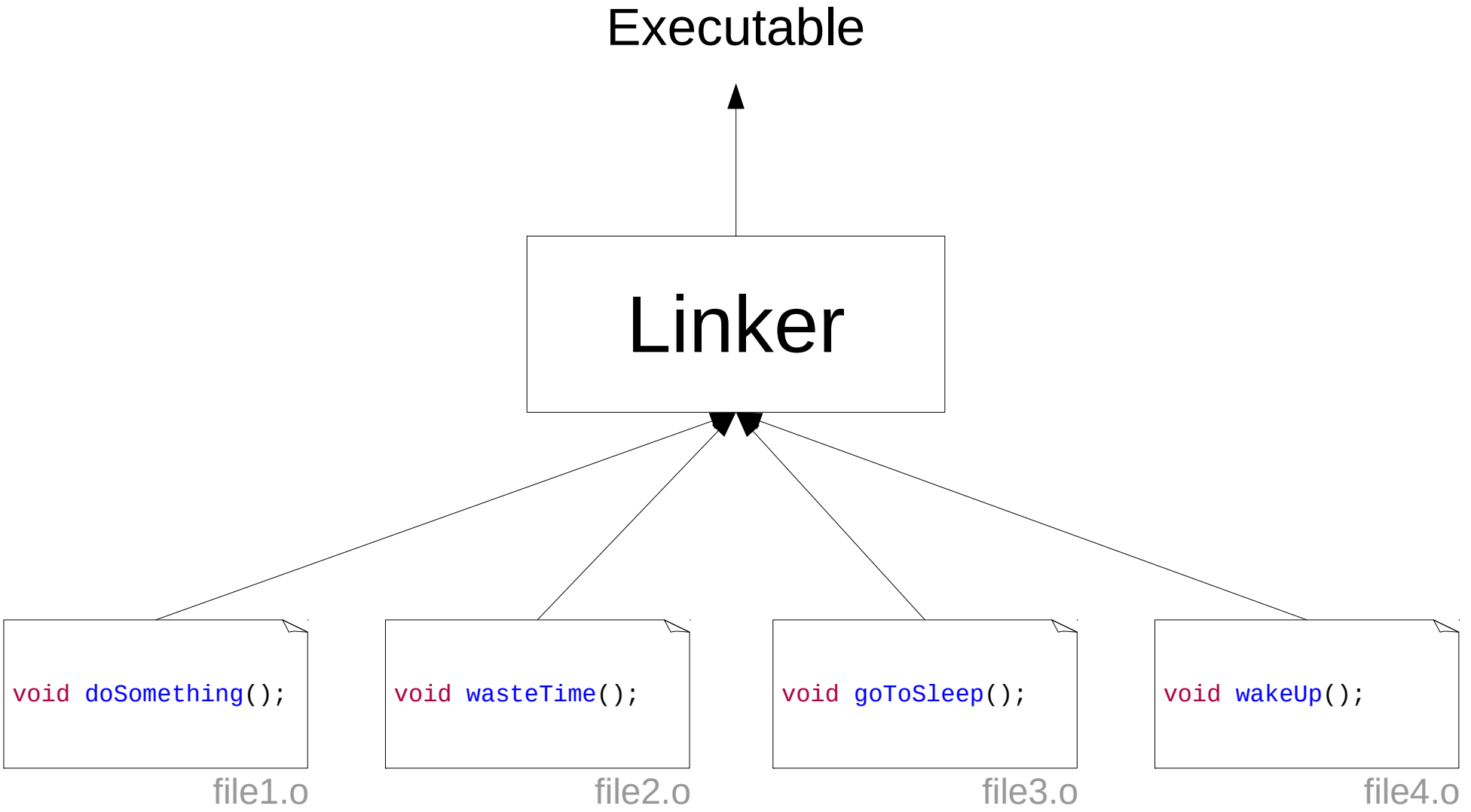
file2.o



file3.o



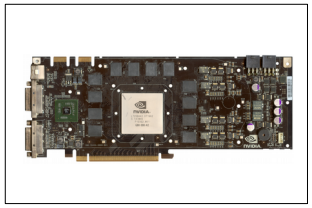
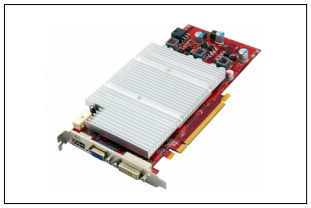
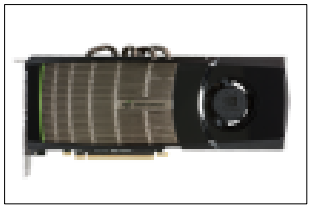
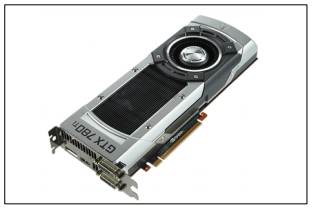
file4.o

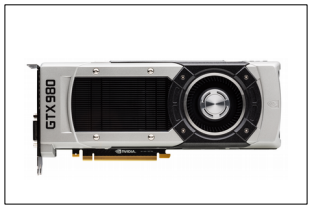

Executable
Linker
void doSomething();
void wasteTime();
void goToSleep();
void wakeUp();
file1.o
file2.o
file3.o
file4.o

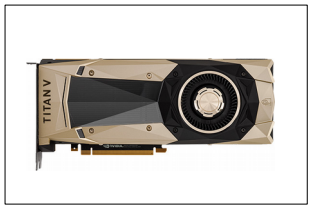
```cpp
#include <stdio.h>


__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}


int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Additional problems a GPU compiler has:

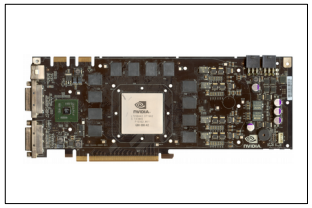
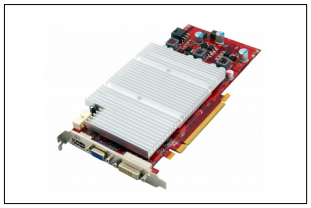
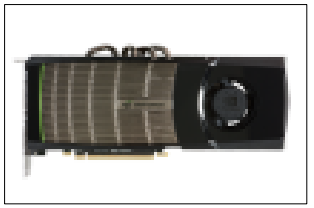
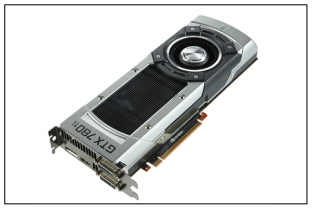
- Same executable may run on different generations of GPU hardware
- Exact hardware is only known at runtime
- Instruction set can vary wildly between generations.

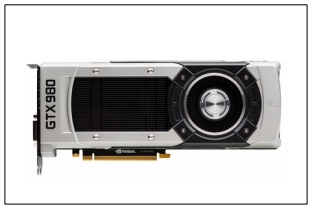





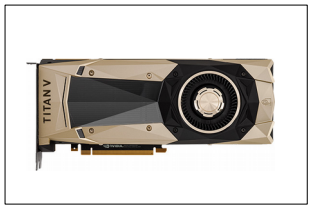




- Same executable may run on different generations of GPU hardware

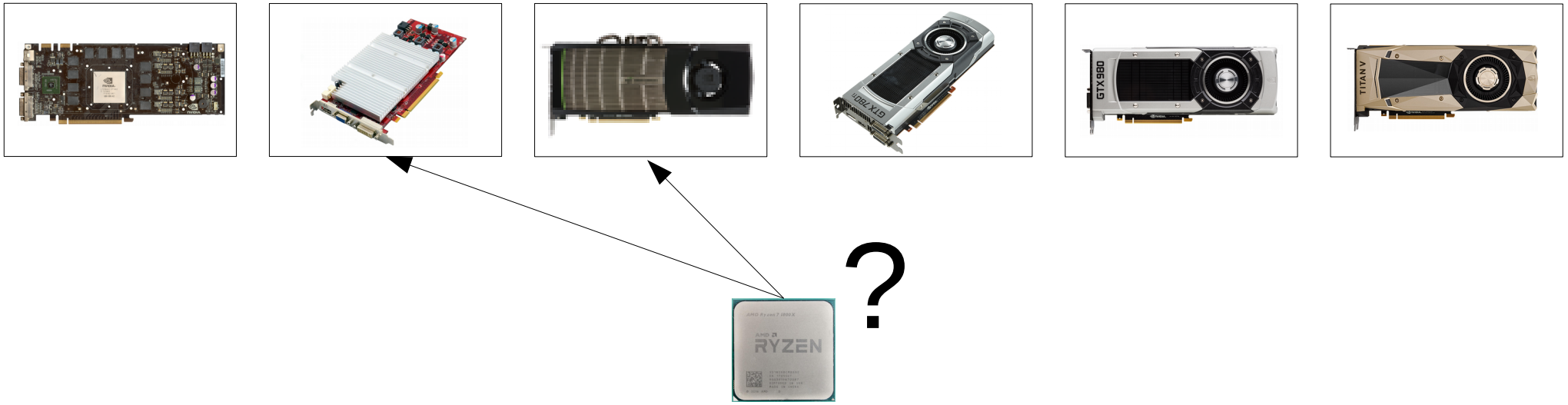









?

- Exact hardware is only known at runtime



- Exact hardware is only known at runtime

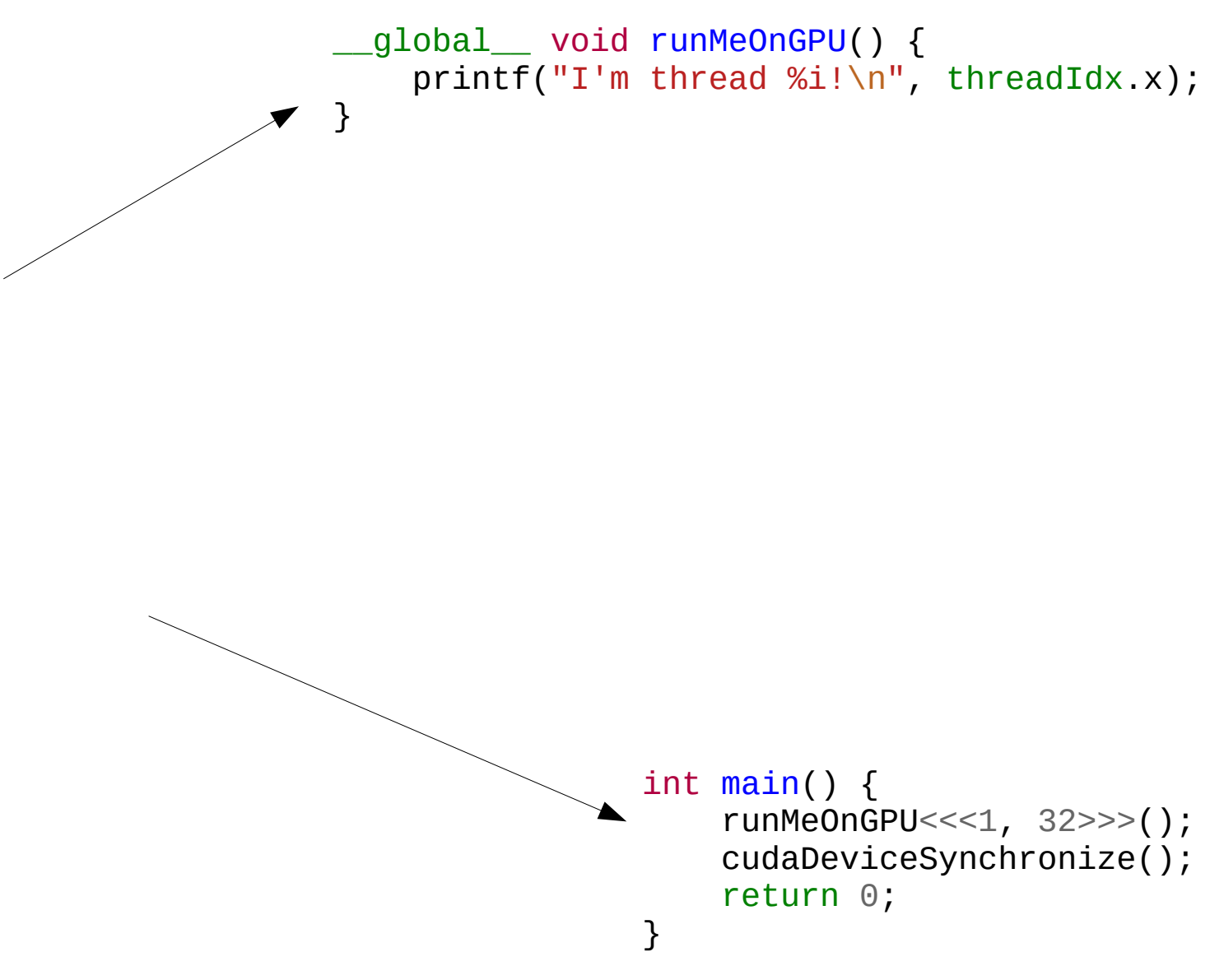
Additional problems a GPU compiler has:

- Same executable may run on different generations of GPU hardware
- Exact hardware is only known at runtime
- **Instruction set can vary wildly between generations.**

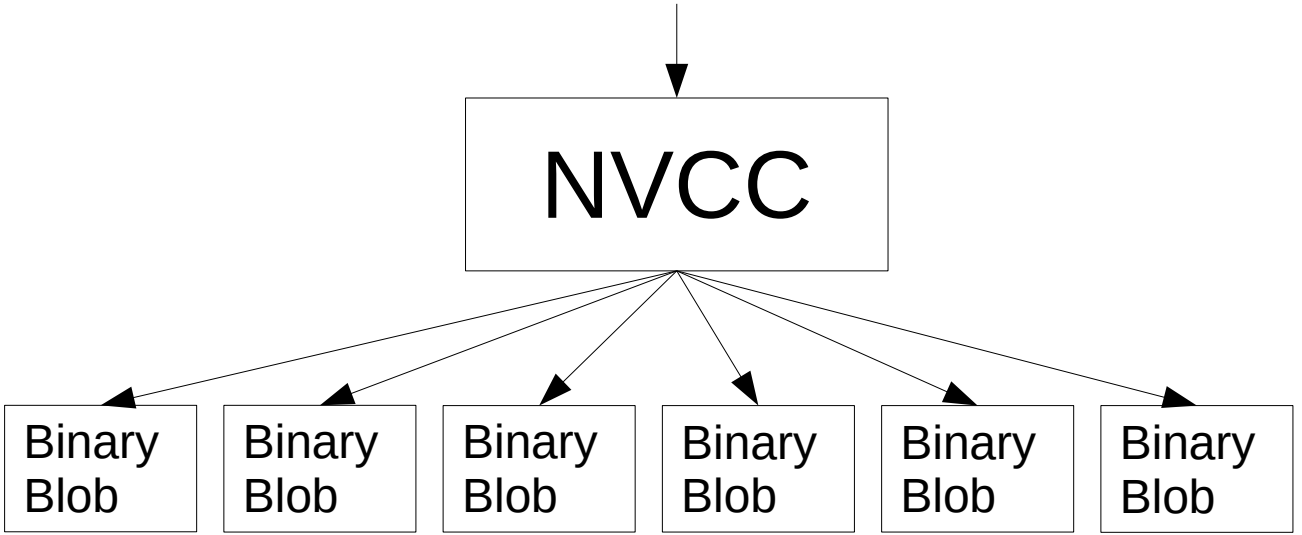
```
#include <stdio.h>

__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

GPU Kernel

CPU Code

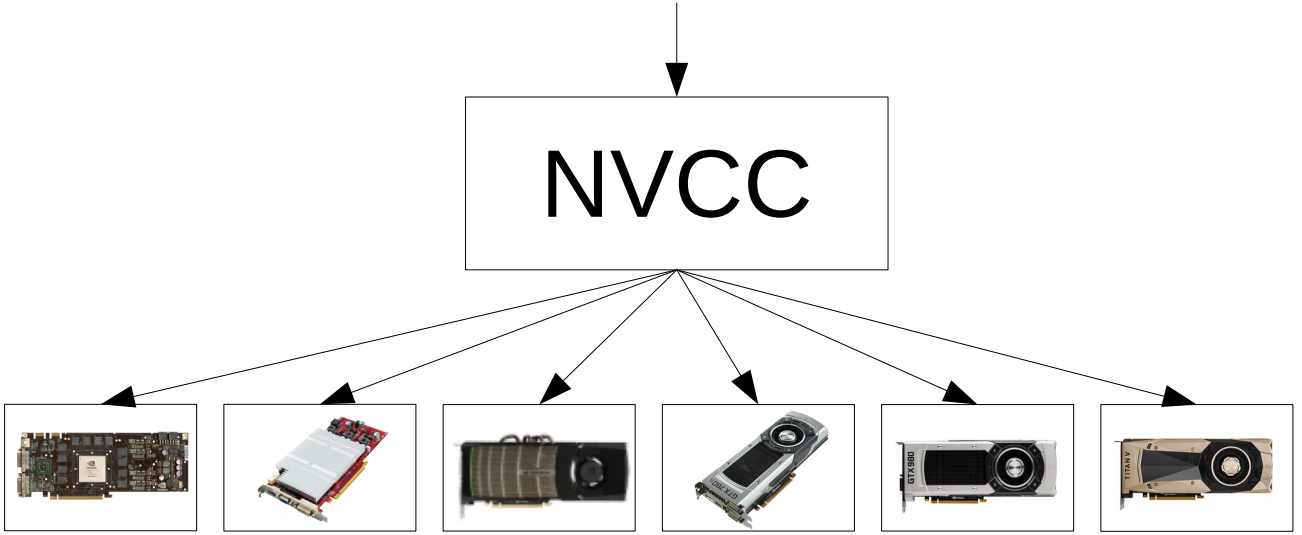

__global__ void runMeOnGPU() {
printf("I'm thread %i!\n", threadIdx.x);
}
int main() {
runMeOnGPU<<<1, 32>>>();
cudaDeviceSynchronize();
return 0;
}

```cpp
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```



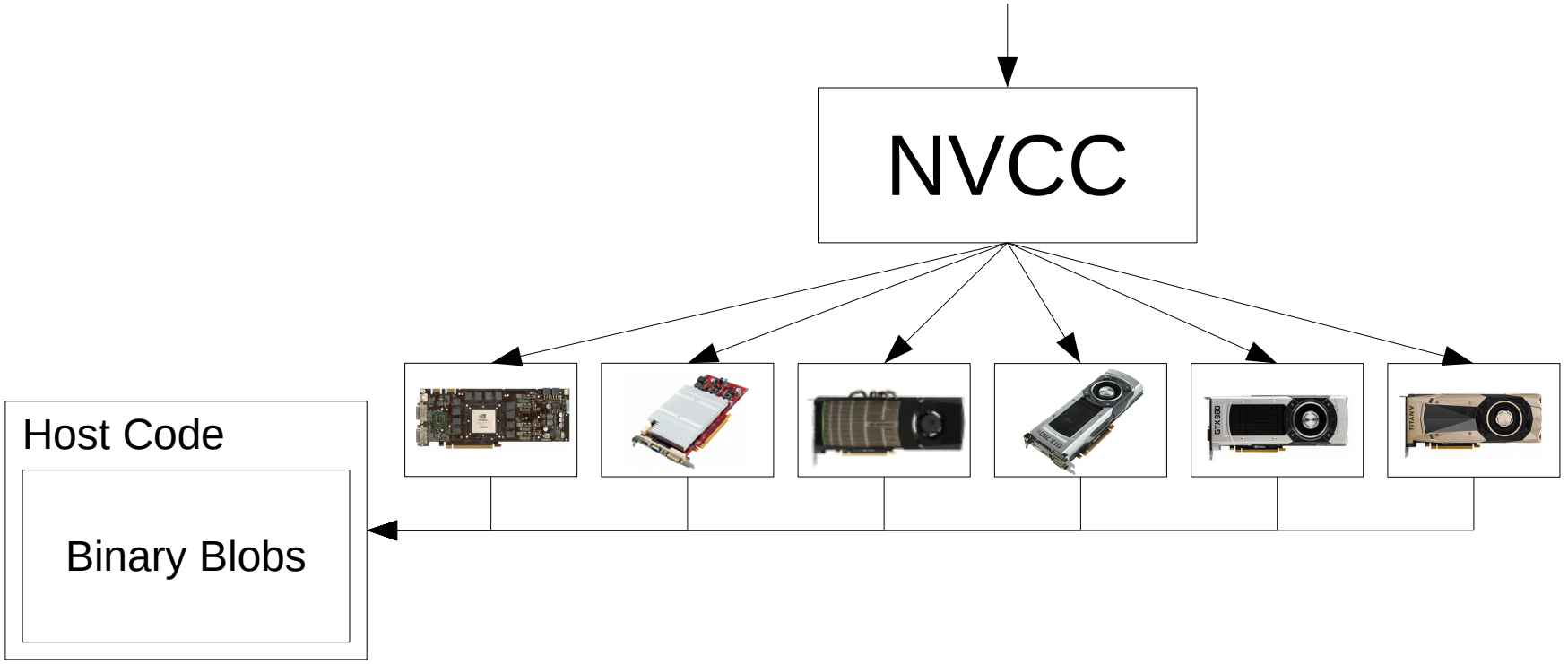

```cpp
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```cpp
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```



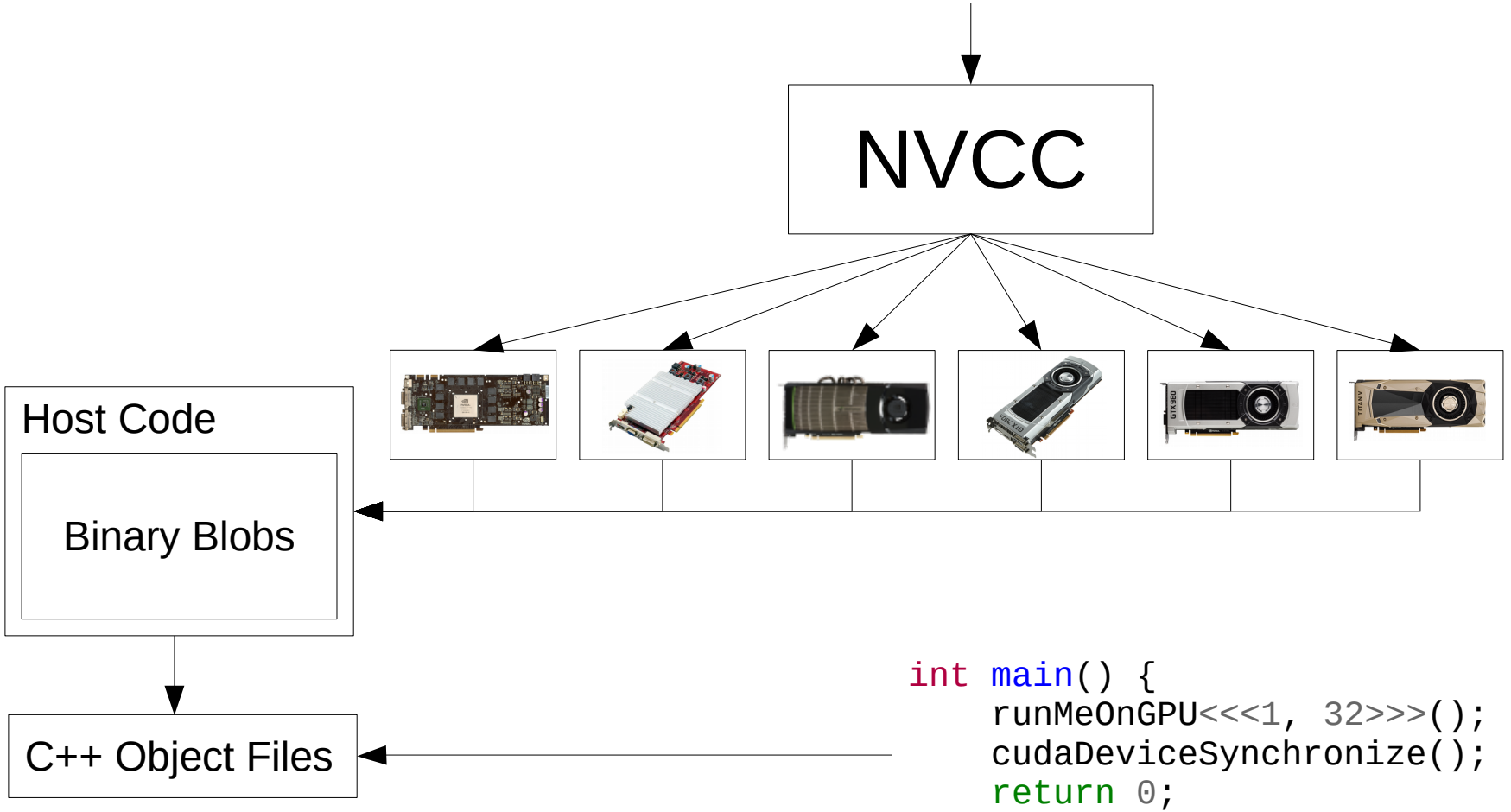

```cpp
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```cpp
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```



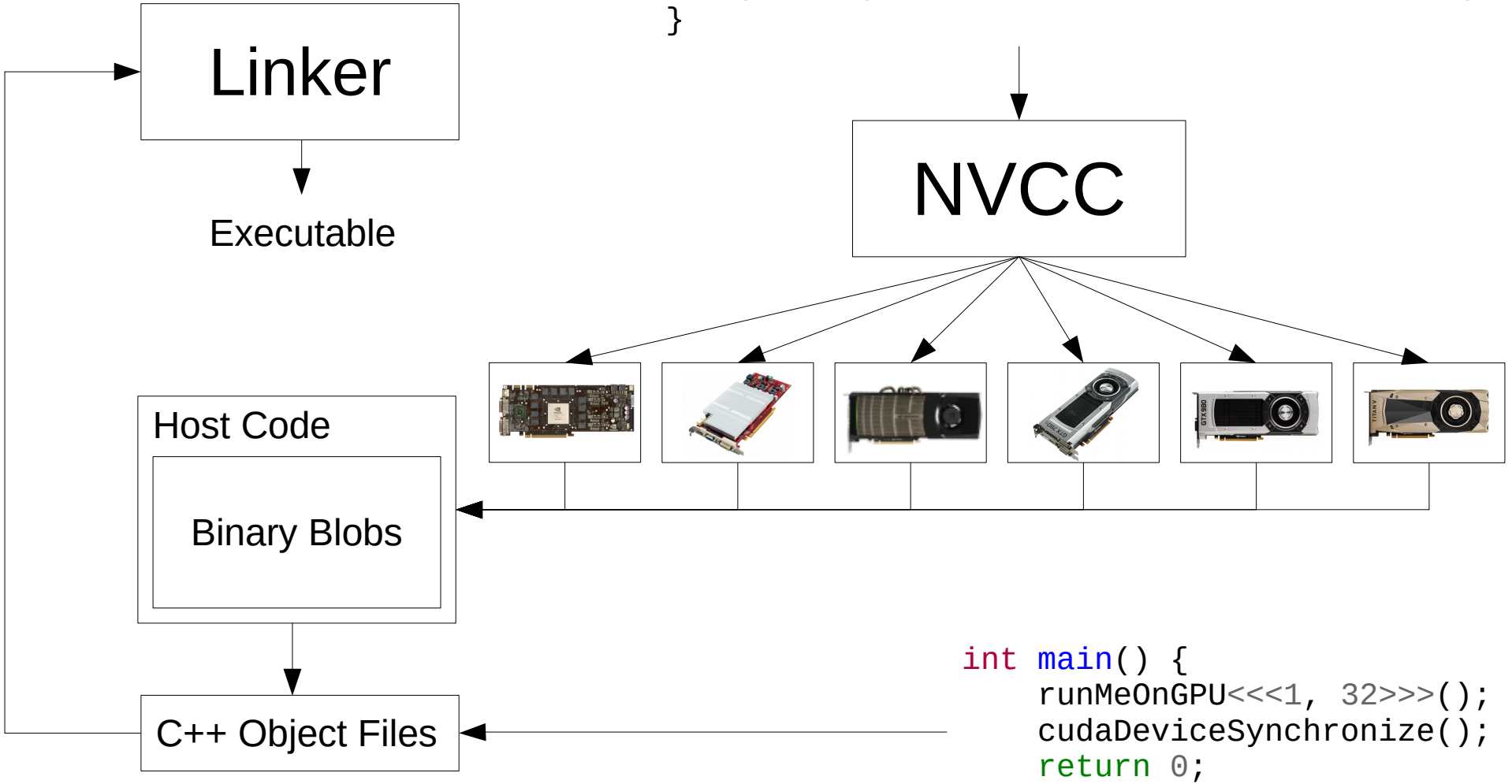

```cpp
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```




```
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```cpp
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```




```cpp
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

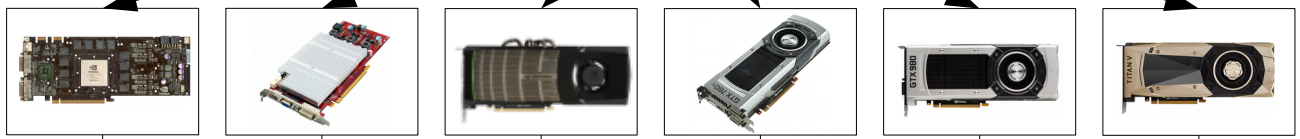
GCC, MSVC, etc

Linker

Executable

```
__global__ void runMeOnGPU() {
    printf("I'm thread %i!\n", threadIdx.x);
}
```

NVCC



Host Code

Binary Blobs

C++ Object Files

```
int main() {
    runMeOnGPU<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Why this matters:

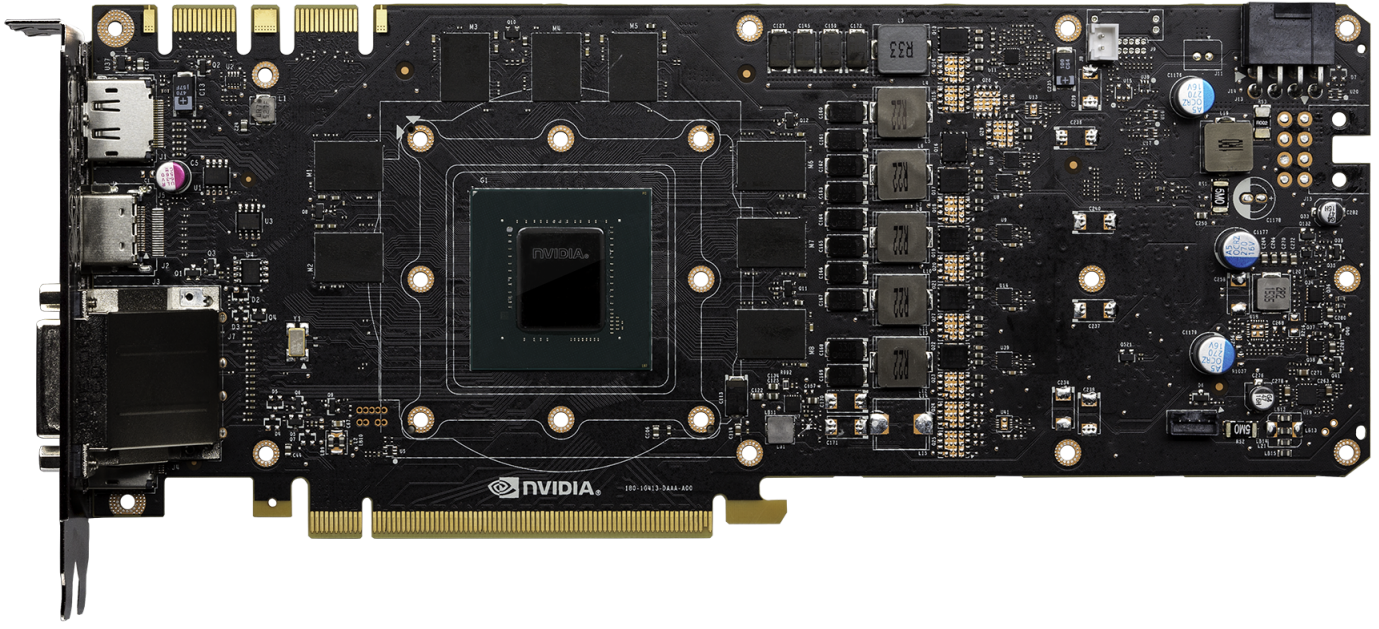
- CUDA is not C++
- NVCC has to support a particular compiler to compile the “host” code
- You may need to specify any architecture(s) you’d like to compile your kernels for.

```cpp
#include <stdio.h>

__global__ void runMeOnGPU() {
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   runMeOnGPU<<<1, 32>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

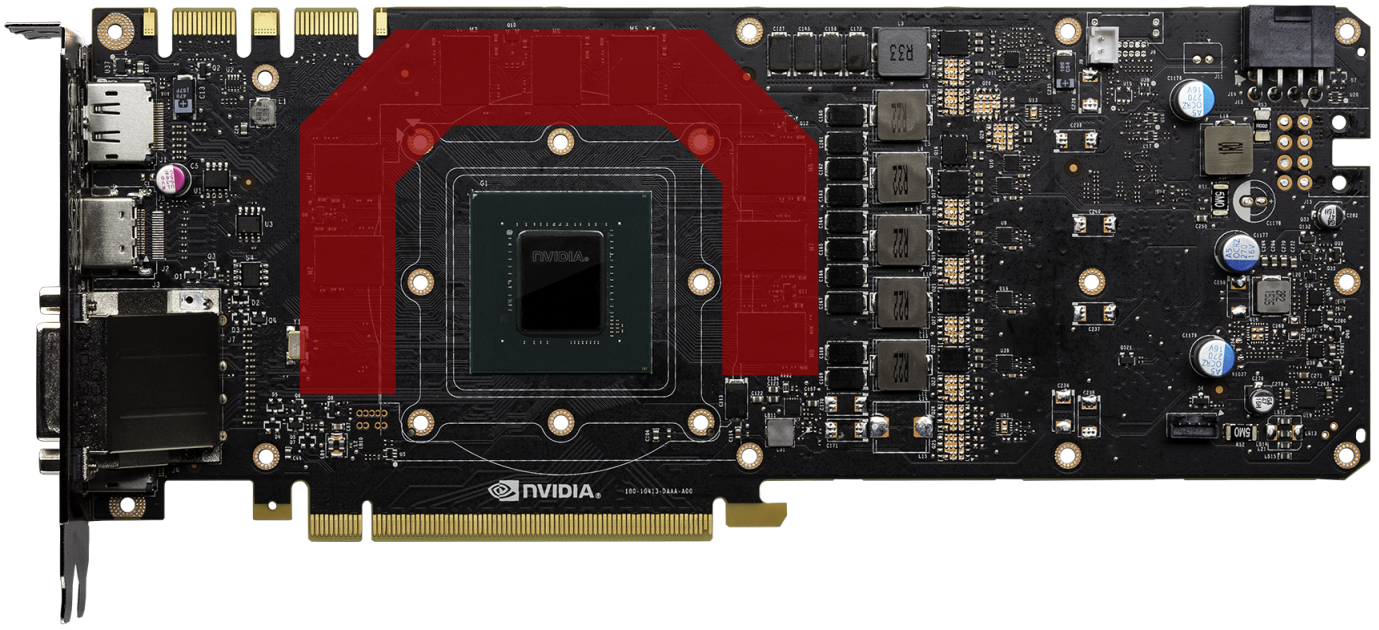
What about larger arrays?

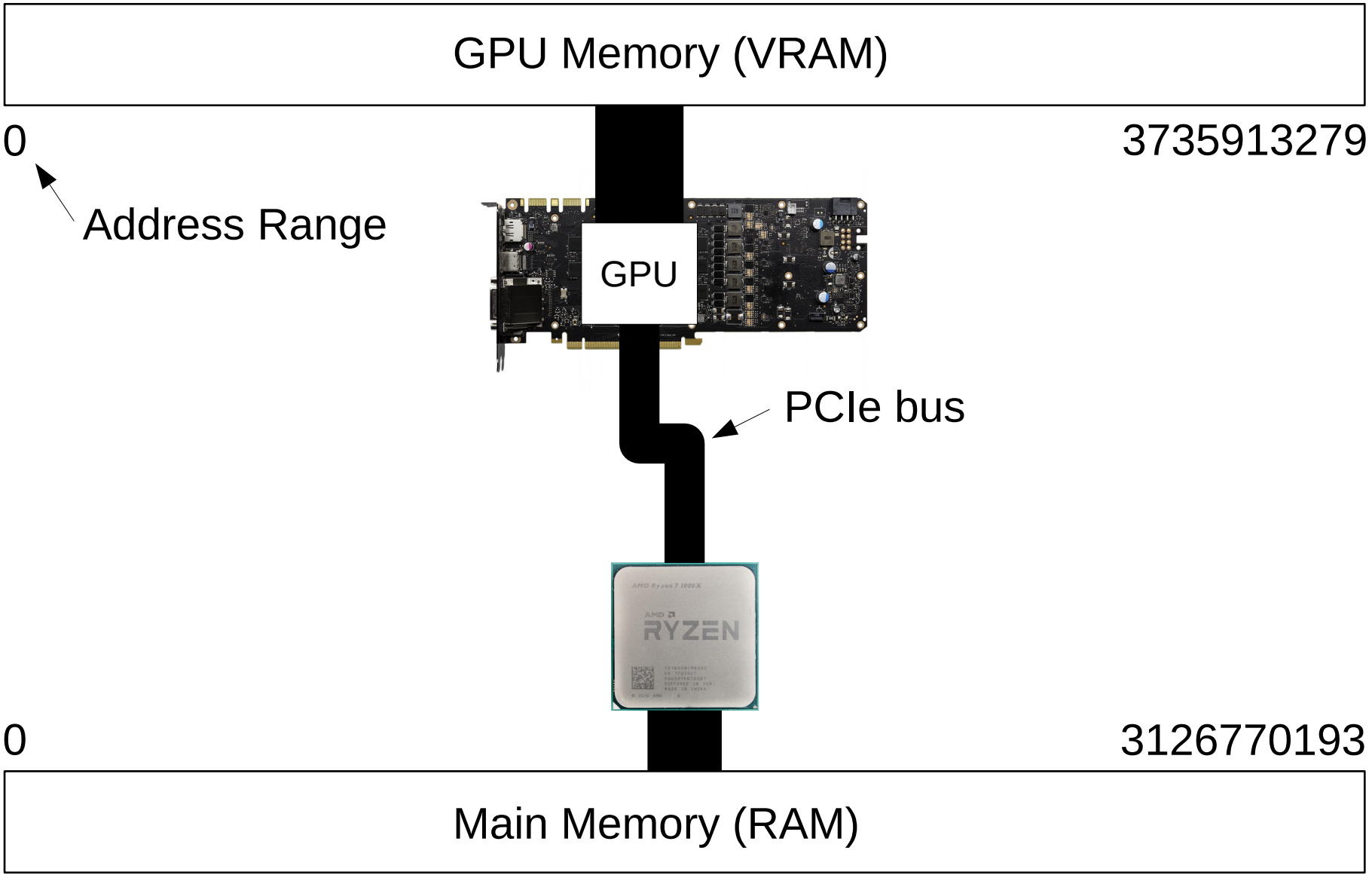
```cpp
#include <stdio.h>

__global__ void runMeOnGPU(int* array) {
   array[threadIdx.x]++;
   printf("I'm thread %i!\n", threadIdx.x);
}

int main() {
   int* largeArray = new int[1000000000];
   runMeOnGPU<<<1, 32>>>(largeArray);
   cudaDeviceSynchronize();
   return 0;
}
```

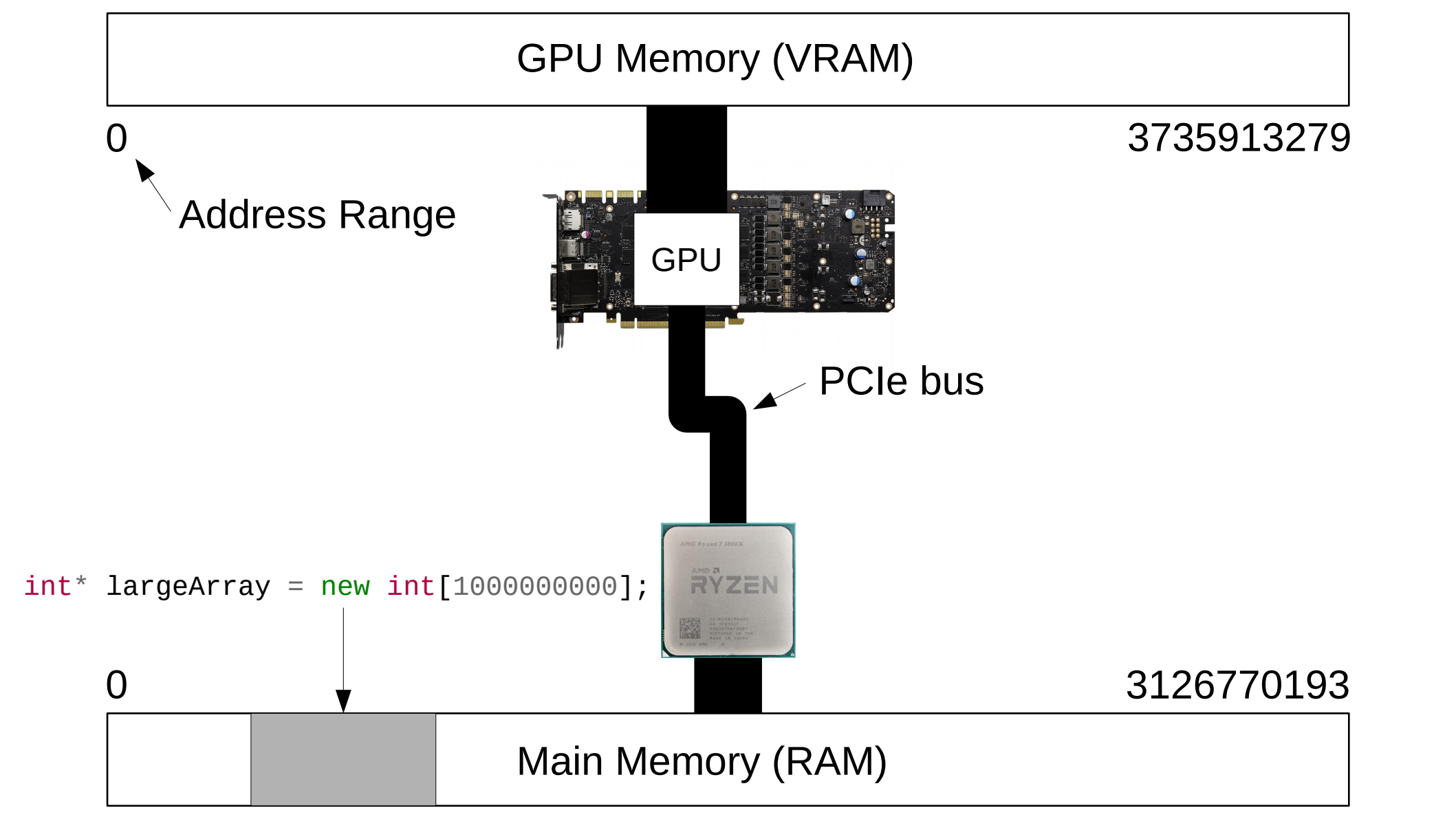

#include <stdio.h>
__global__ vo nMeOnGPU(i rray) {
array[thre]++;
printf("I th %i!\n", th dIdx.x);
}
int main()
int* larg ray = new 1000 0000];
runMeOnGPU 1, 32>>>(la);
cudaDeviceS ronize();
return 0;
}

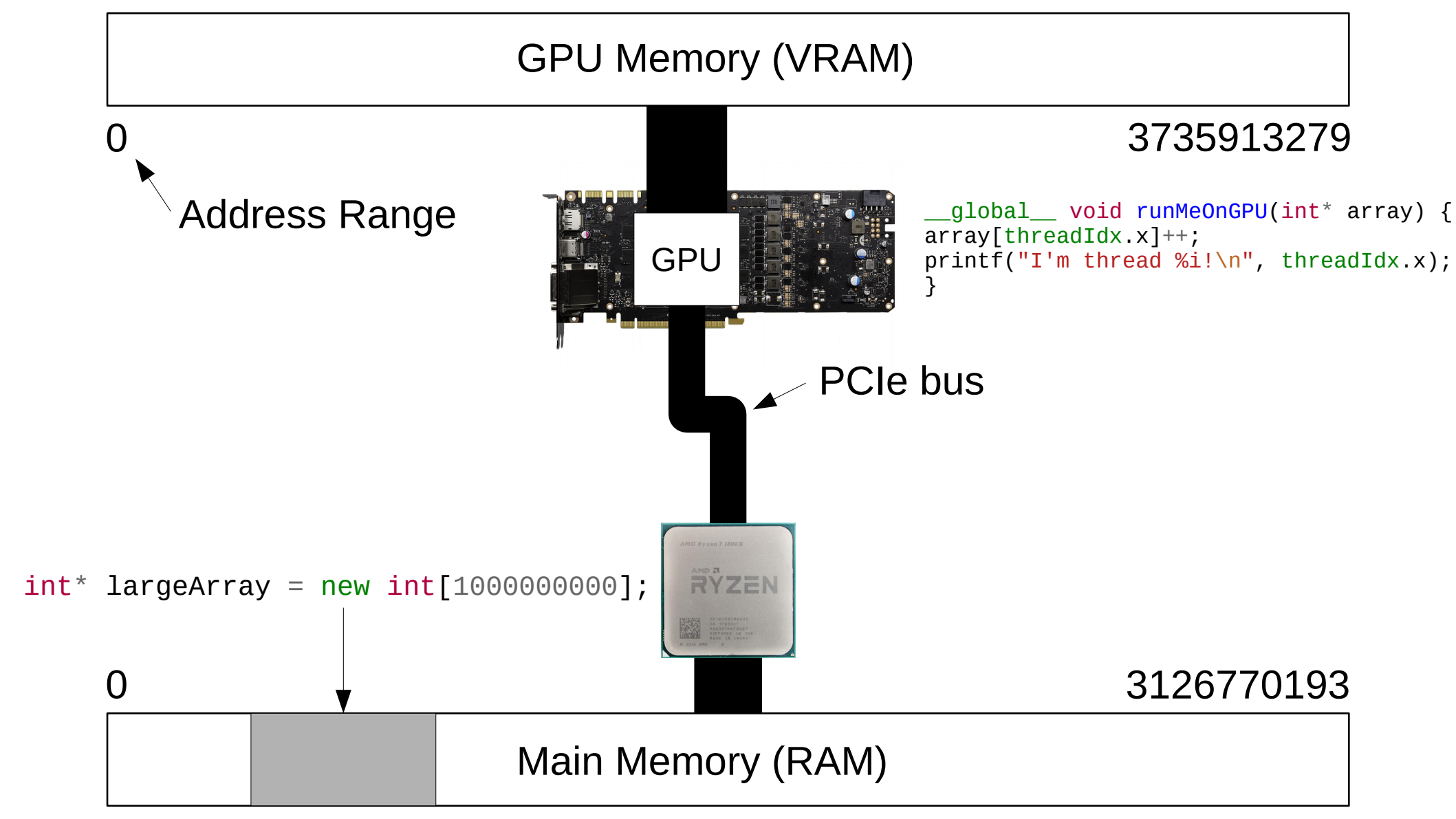

NVIDIA
NVIDIA

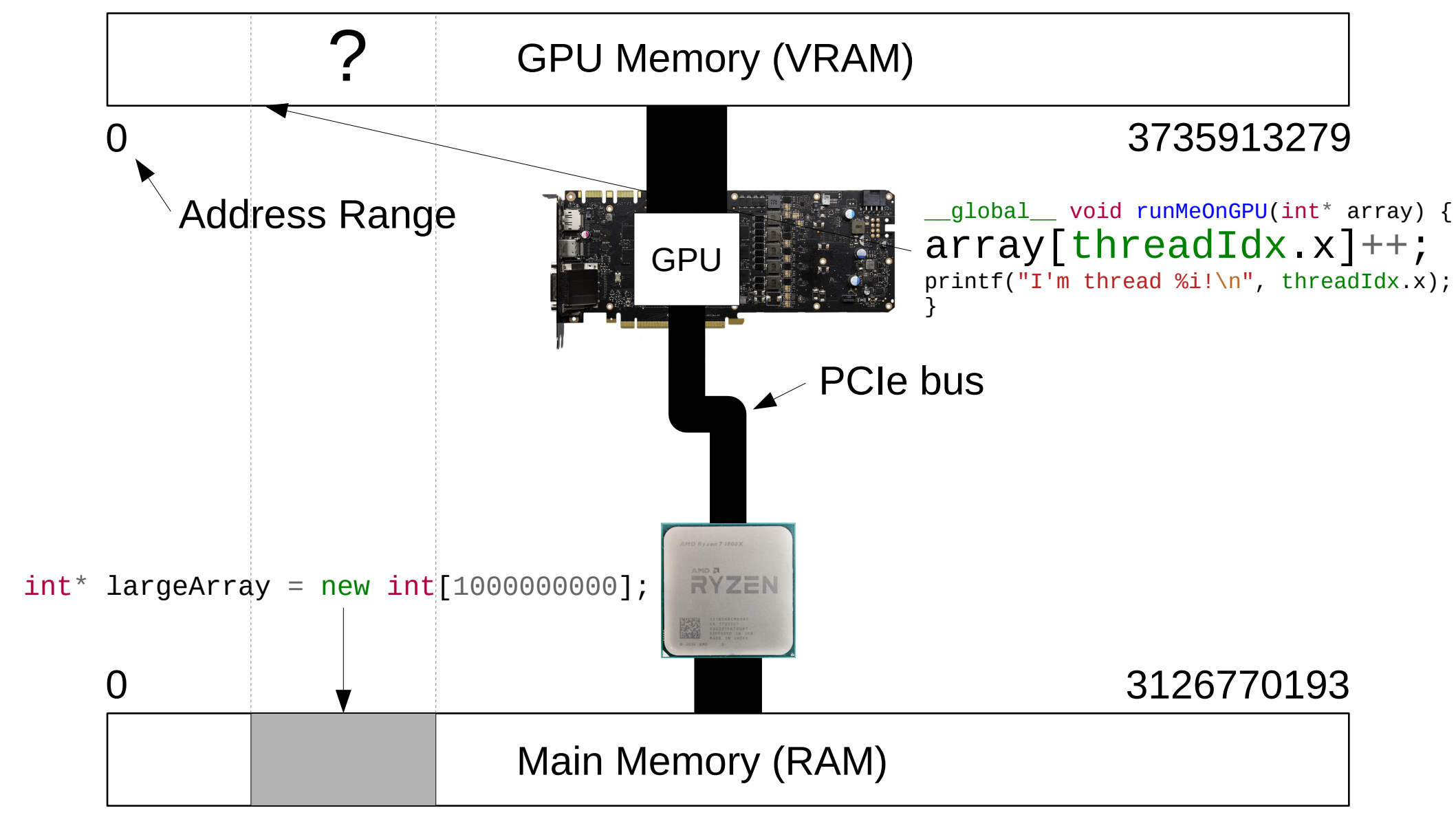
The GPU has its own memory banks!



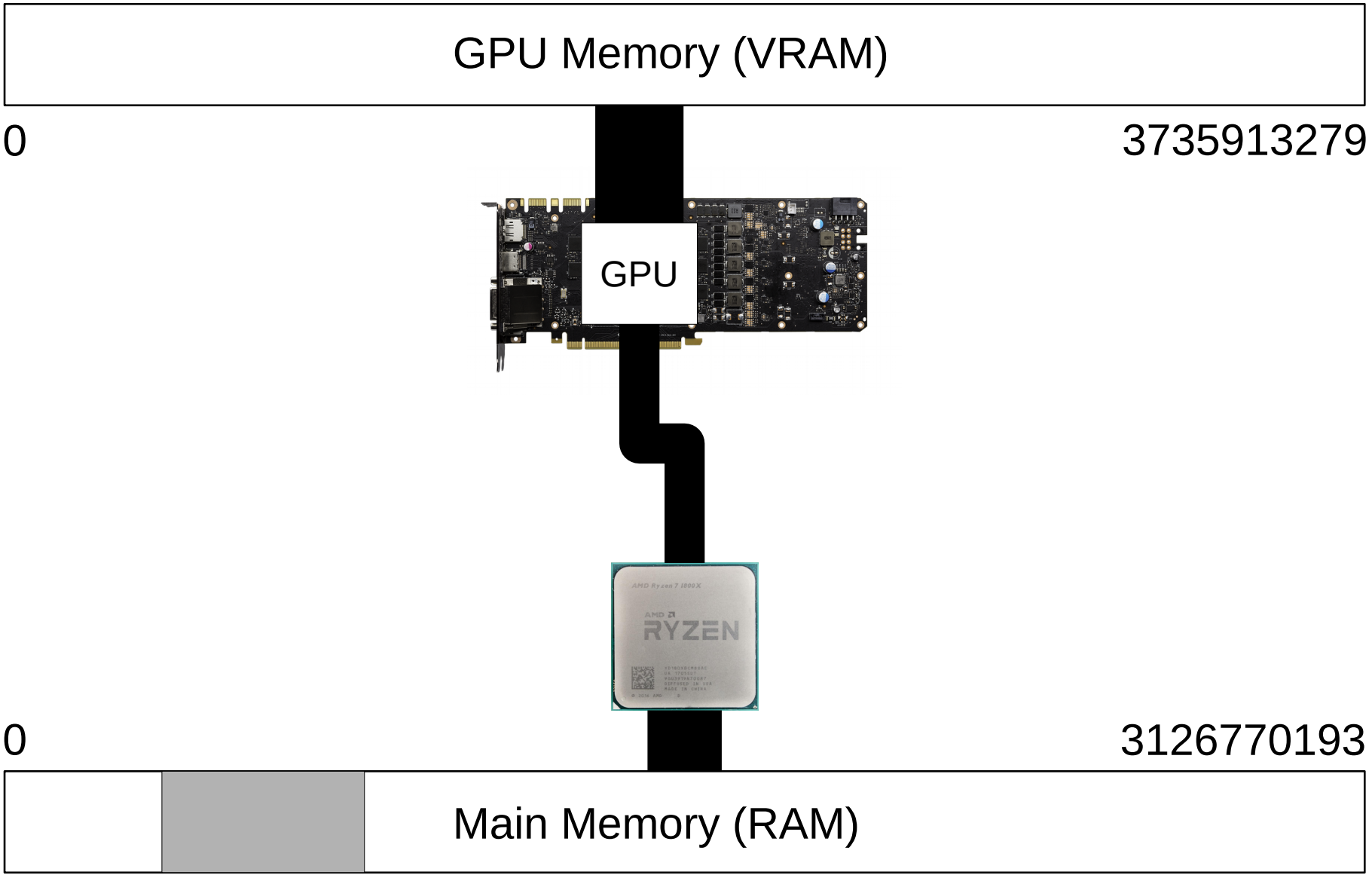

GPU Memory (VRAM)
0
3735913279
Address Range
GPU
PCIe bus
0
3126770193
Main Memory (RAM)


GPU Memory (VRAM)
0
3735913279
Address Range
GPU
PCIe bus
int* largeArray = new int[1000000000];
0
3126770193
Main Memory (RAM)


GPU Memory (VRAM)
0
3735913279
Address Range
GPU
__global__ void runMeOnGPU(int* array) {
array[threadIdx.x]++;
printf("I'm thread %i!\n", threadIdx.x);
}
PCIe bus
int* largeArray = new int[1000000000];
0
3126770193
Main Memory (RAM)


?
GPU Memory (VRAM)
0
3735913279
Address Range
GPU
__global__ void runMeOnGPU(int* array) {
array[threadIdx.x]++;
printf("I'm thread %i!\n", threadIdx.x);
}
PCIe bus
int* largeArray = new int[1000000000];
0
3126770193
Main Memory (RAM)

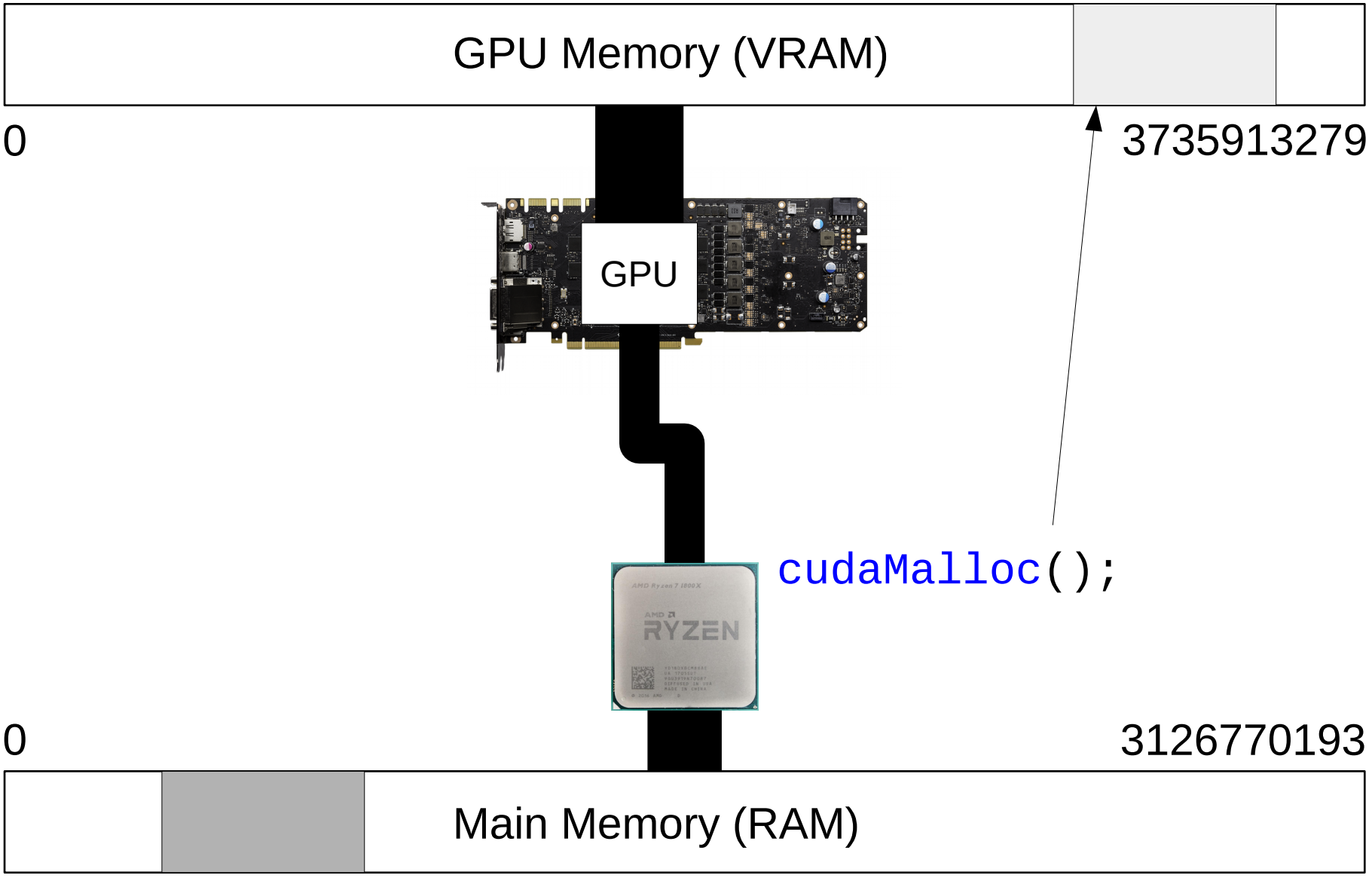
Memory needs to be allocated and copied to the GPU explicitly


GPU Memory (VRAM)
0
3735913279
GPU
AMD Ryzen 7 1800X
AMD
RYZEN
0
3126770193
Main Memory (RAM)

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

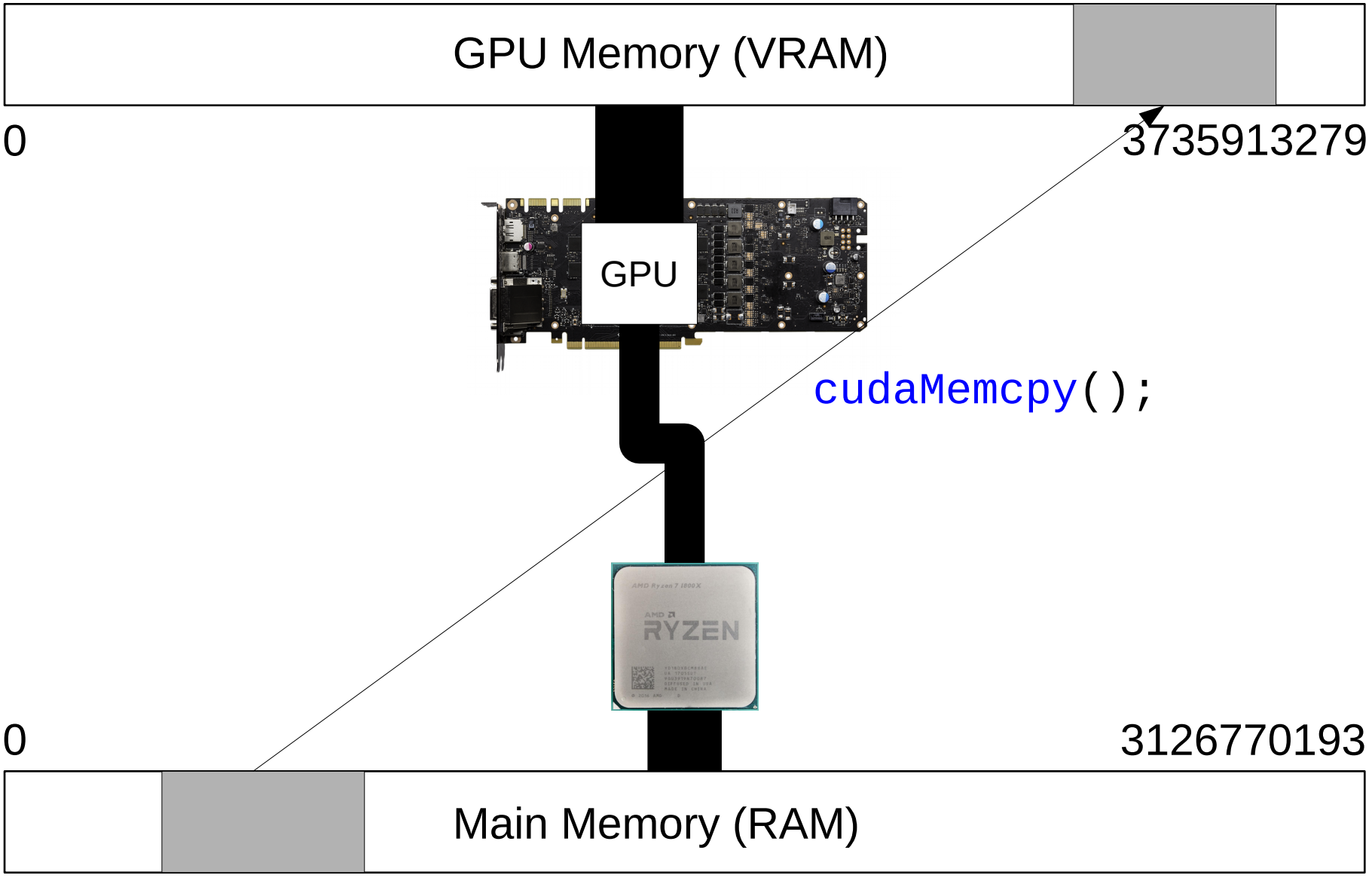
```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

```
float* device_bigArray;
cudaMalloc(&device_bigArray, sizeof(float) * length);
```


GPU Memory (VRAM)
0
3735913279
GPU
cudaMalloc();
0
3126770193
Main Memory (RAM)

```
cudaError_t cudaMemcpy(void* dest, const void* src,
                       size_t count, cudaMemcpyKind kind);
```

```
cudaError_t cudaMemcpy(void* dest, const void* src,
                       size_t count, cudaMemcpyKind kind);


cudaMemcpy(device_bigArray, bigArray,
   sizeof(float) * length, cudaMemcpyHostToDevice);

cudaMemcpy(bigArray, device_bigArray,
   sizeof(float) * length, cudaMemcpyDeviceToHost);
```


GPU Memory (VRAM)
0
3735913279
GPU
cudaMemcpy();
0
3126770193
Main Memory (RAM)

Remember that std::vector is a two-part structure!

```
cudaError_t cudaFree(void** devicePointer);
```

That’s how we copy memory back and forth.

But using memory inside a kernel is slow.

What else can we do?

Shared Memory

Shared Memory

- Blocks can share memory

- Works as a “programmable cache”

- Allows communication and cooperation between warps

~10x faster than main memory
~10x slower than registers

Last topics:

- What about race conditions?

- Thread divergence

- Debugging CUDA code

- Practical notes

What about race conditions?

Bad news:

They still exist

But we have atomic operations!

```
int atomicSub(int* address, int val);
int atomicExch(int* address, int val);
int atomicMin(int* address, int val);
int atomicMax(int* address, int val);
unsigned int atomicInc(..);
unsigned int atomicDec(..);
int atomicCAS(int* address, int compare, int val);
int atomicAnd(int* address, int val);
int atomicOr(int* address, int val);
int atomicXor(int* address, int val);
```

But no mutexes!

(can create one using atomicCAS, but it’s prone to deadlocks on anything but brand new hardware)

Debugging CUDA code

- Can do stepwise debugging, though may need alternate GPU to draw the screen

Debugging CUDA code

- Can do stepwise debugging, though may need alternate GPU to draw the screen
- In my experience:
  - Use assertions
  - Use printf() to find anomalous values
  - Worst case: write debug values into a buffer, dump buffer into excel sheet (lot of work).

Practical notes:

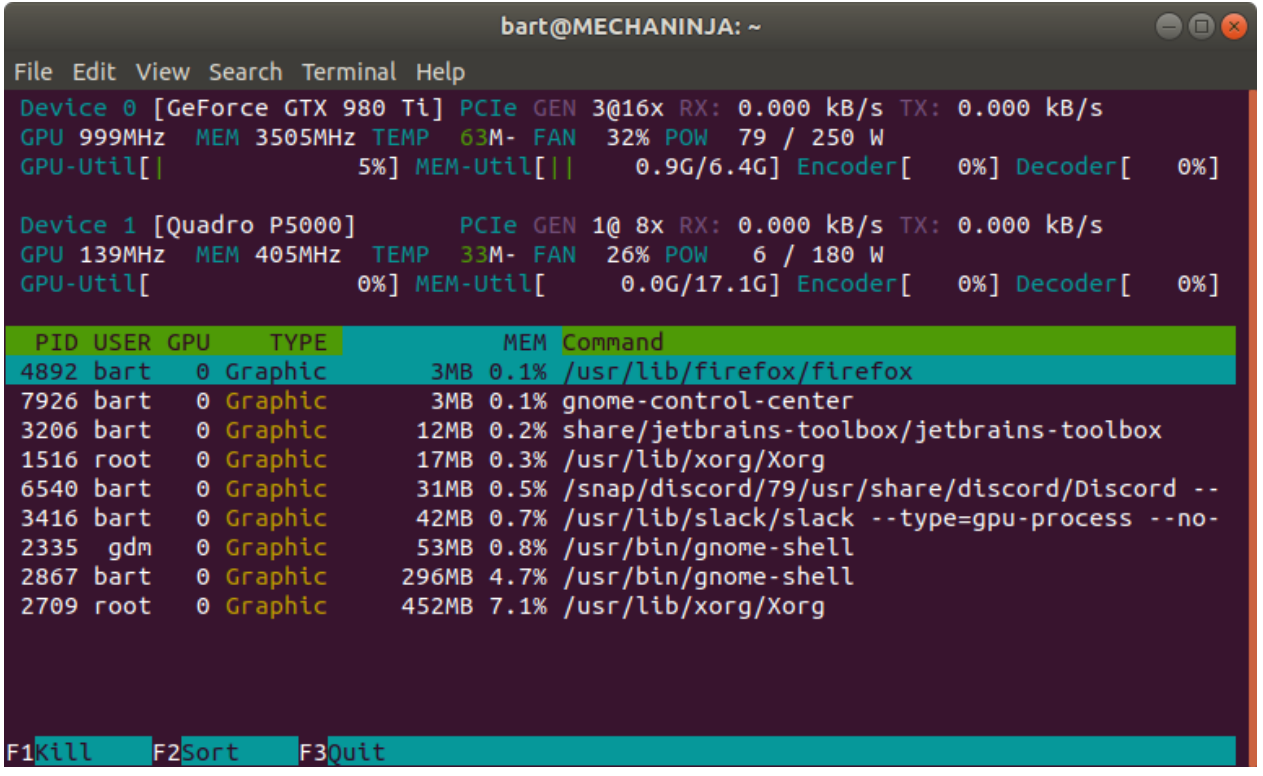
CUDA is not C++

Practical notes:

Windows and the WDDM

Pratical notes:

Nvtop: https://github.com/Syllo/nvtop



Next week