CS516: Parallelization of Programs

Introduction GPUs and CUDA Programming

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

2023-24 W

Course Outline

- Introduction
- Overview of Parallel Architectures
- Performance
- Parallel Programming
  - GPUs and CUDA programming
  - CUDA thread organization
  - Instruction execution
  - GPU memories
  - Synchronization
  - Unified memory
- Case studies
- Extracting Parallelism from Sequential Programs Automatically

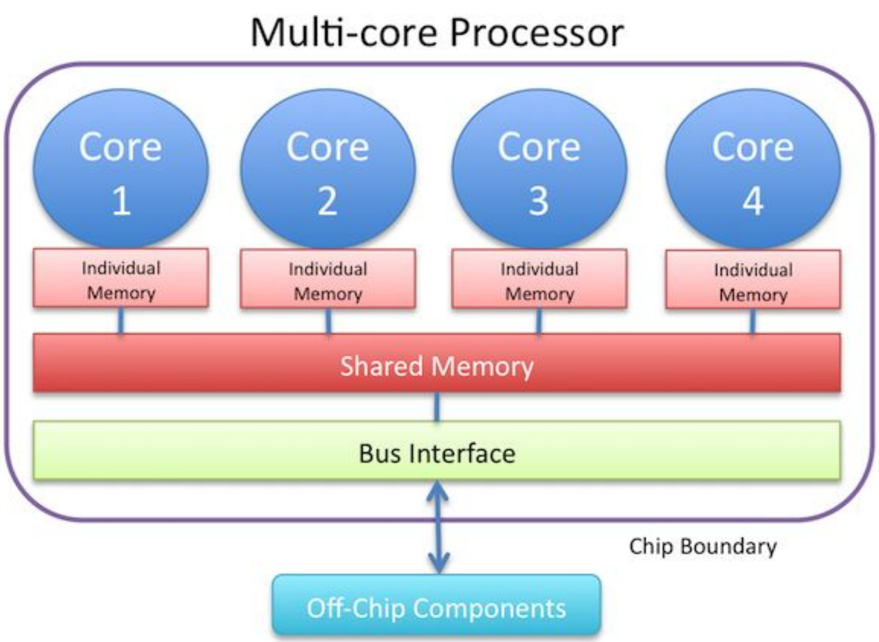
Outline

- GPUs and CUDA Programming Demos

Motivation

- For many decades, the single core processors were popular
  - Instruction-level parallelism
  - Core clock frequency
  - Moore’s law
- Mid-to late-1990s - power wall
  - Power constraints
  - Heat dissipation
- Multicore processors, accelerators, such as GPUs.

Why GPUs?

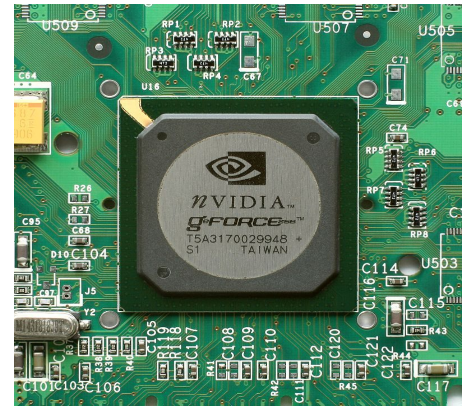


- Multicore processors
  - Task level parallelism
  - Graphics rendering is computationally expensive
  - Not efficient for graphics applications

Images Source: Internet

Graphics Processing Units

- The early GPU designs
  - Specialized for graphics processing only
  - Exhibit SIMD execution
  - Less programmable

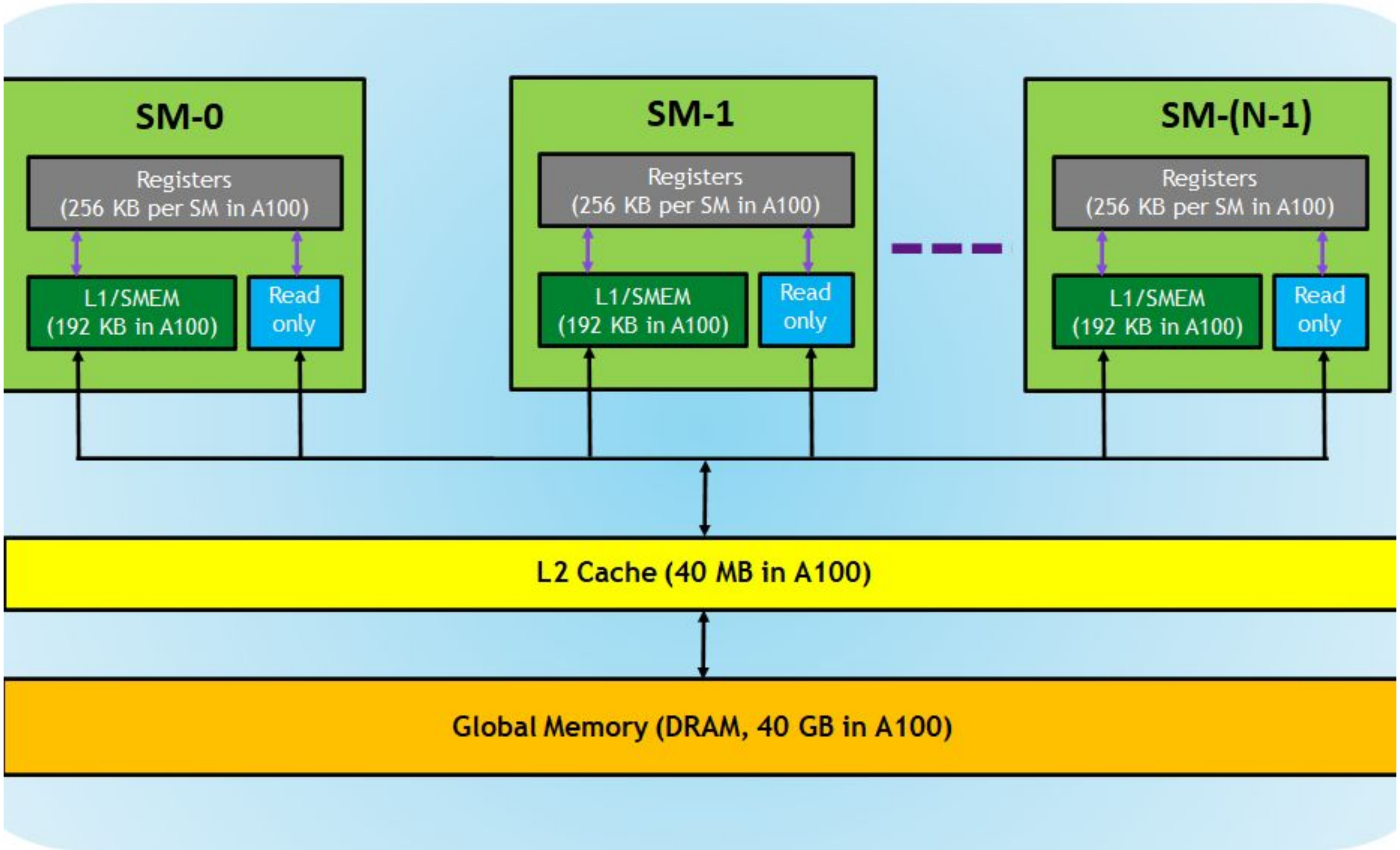


NVIDIA GeForce 256

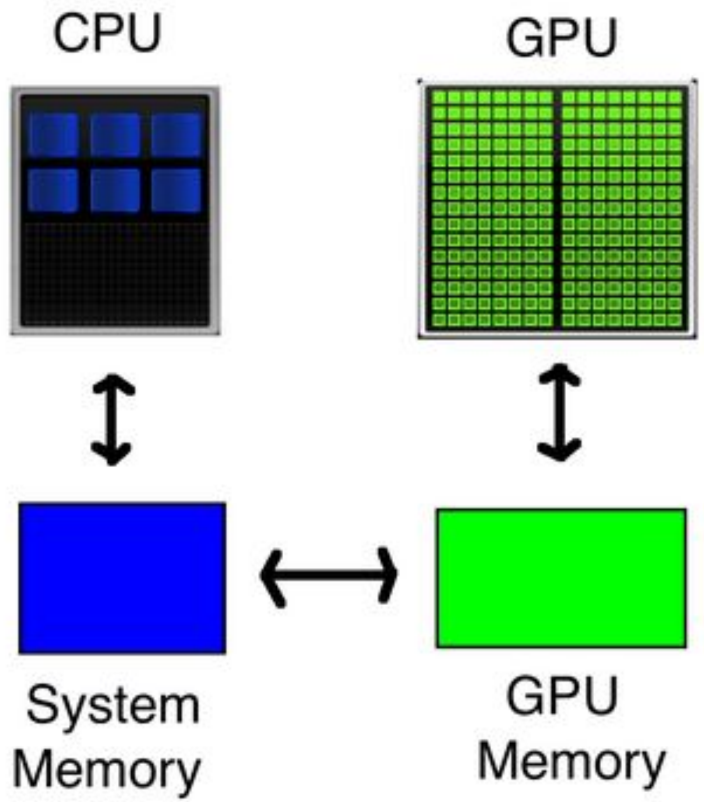
- In 2007, fully programmable GPUs
  - CUDA released

Images Source: Internet

GPU Architecture



GPU Architecture



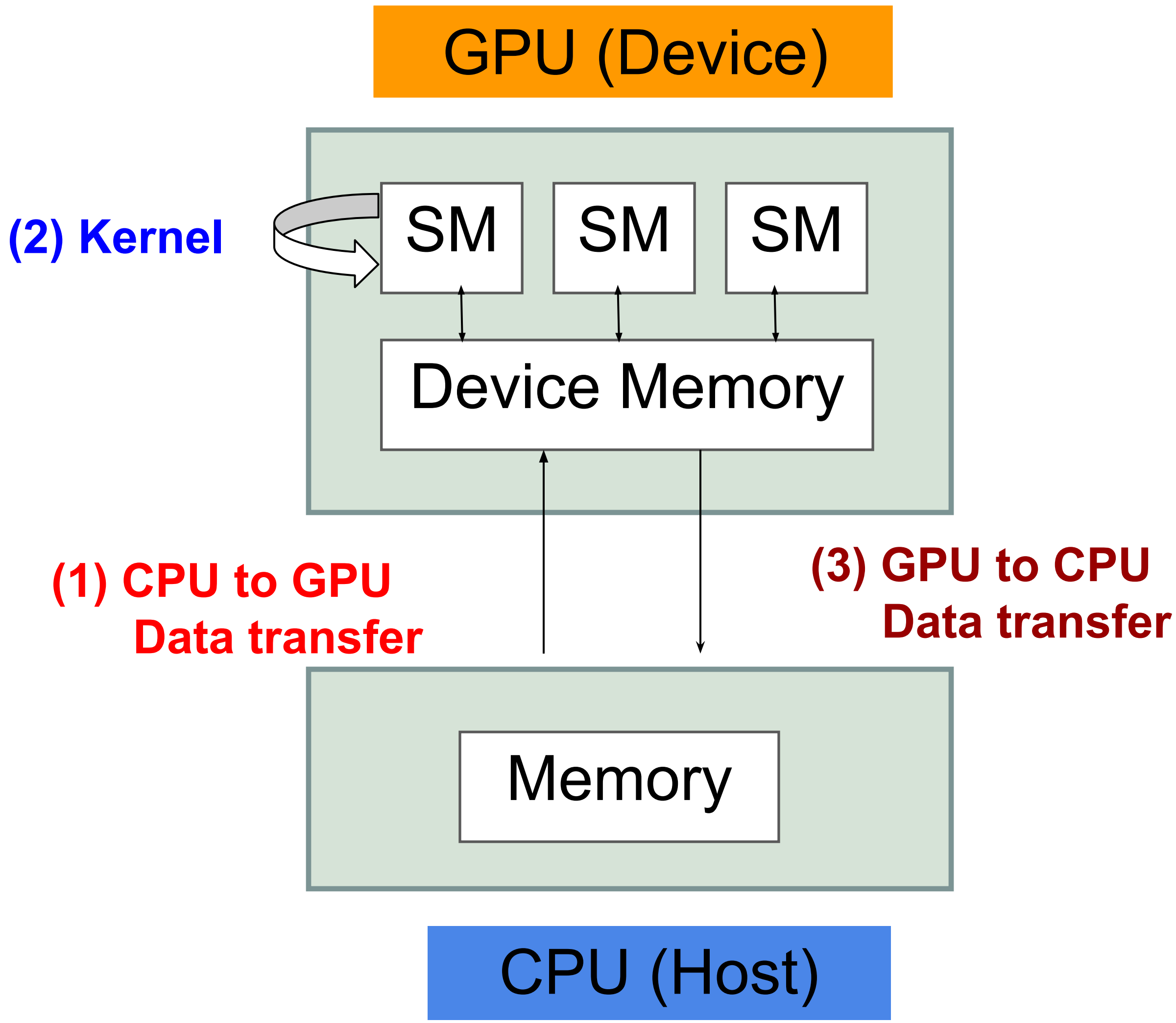
Parallelizing Programs on GPUs

Programming Models

- CUDA (Compute Unified Device Architecture)
  - Supports NVIDIA GPUs
  - Extension of C programming language
  - Popular in academia

- OpenCL (Open Computing Language)
  - Open source
  - Supports various GPU devices

Introduction to CUDA Programming



Hello World

```c
#include <stdio.h>

int main() {

   printf("Hello World.\n");

   return 0;

}
```

Compile: **gcc hello.c**
Run: **./a.out**
Hello World.

Hello World in GPU

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf("Hello World.\n");
}
int main() {
    dkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile: **nvcc hello.cu**

Run: **./a.out**

Hello World.

Hello World in GPU

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf("Hello World.\n");
}
int main() {
    dkernel<<<1, 1>>>();
    return 0;
}
```

Compile: **nvcc hello.cu**
Run: **./a.out**
No output

GPU Kernel launch is asynchronous!

Hello World in GPU

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
   printf("Hello World.\n");
}
int main() {
   dkernel<<<1, 1>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Compile: **nvcc hello.cu**
Run: **./a.out**
Hello World.

Hello World in Parallel in GPU

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf("Hello World.\n");
}
int main() {
    dkernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile: **nvcc hello.cu**

Run: **./a.out**

$\left.\begin{array}{l}\textbf{Hello World.}\\ \textbf{Hello World.}\\ \dots\dots\dots\dots\\ \textbf{Hello World.}\end{array}\right\}$ 32 times

Example-1

```
#include <stdio.h>
#define N 100
int main() {
int i;
   for (i = 0; i < N; ++i)
      printf("%d\n", i * i);
   return 0;
}
```

Example-1

```c
#include <stdio.h>
#define N 100
int main() {
int i;
    for (i = 0; i < N; ++i)
        printf("%d\n", i * i);
    return 0;
}
```

```c
#include <stdio.h>
#include <cuda.h>
#define N 100
 __global__    void fun() {
    printf("%d\n", threadIdx.x*threadIdx.x);
}
int main() {
    fun<<<1, N>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

GPU Hello World with a Global

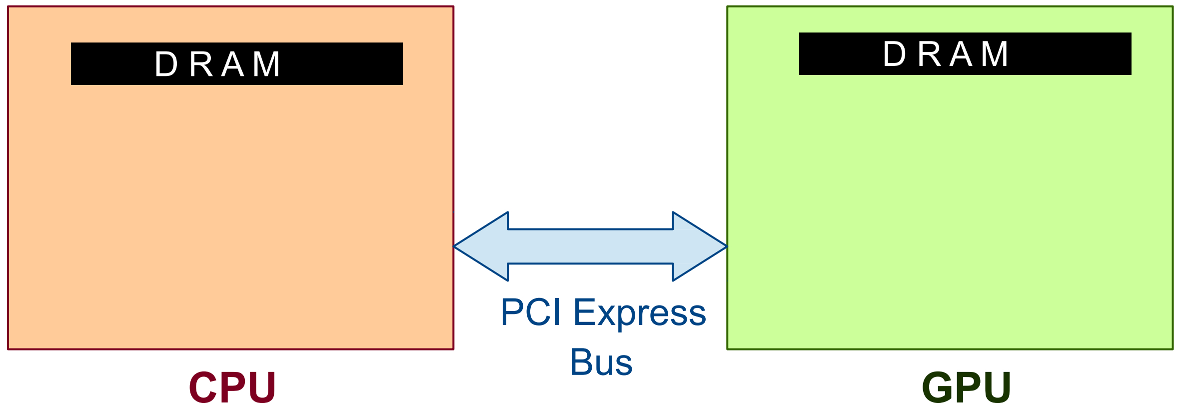
```
#include <stdio.h>
#include <cuda.h>
const char *msg = "Hello World.\n";
__global__ void dkernel() {
    printf(msg);
}
int main() {
    dkernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile: nvcc hello.cu
error: identifier "msg" is undefined in device code

Takeaway

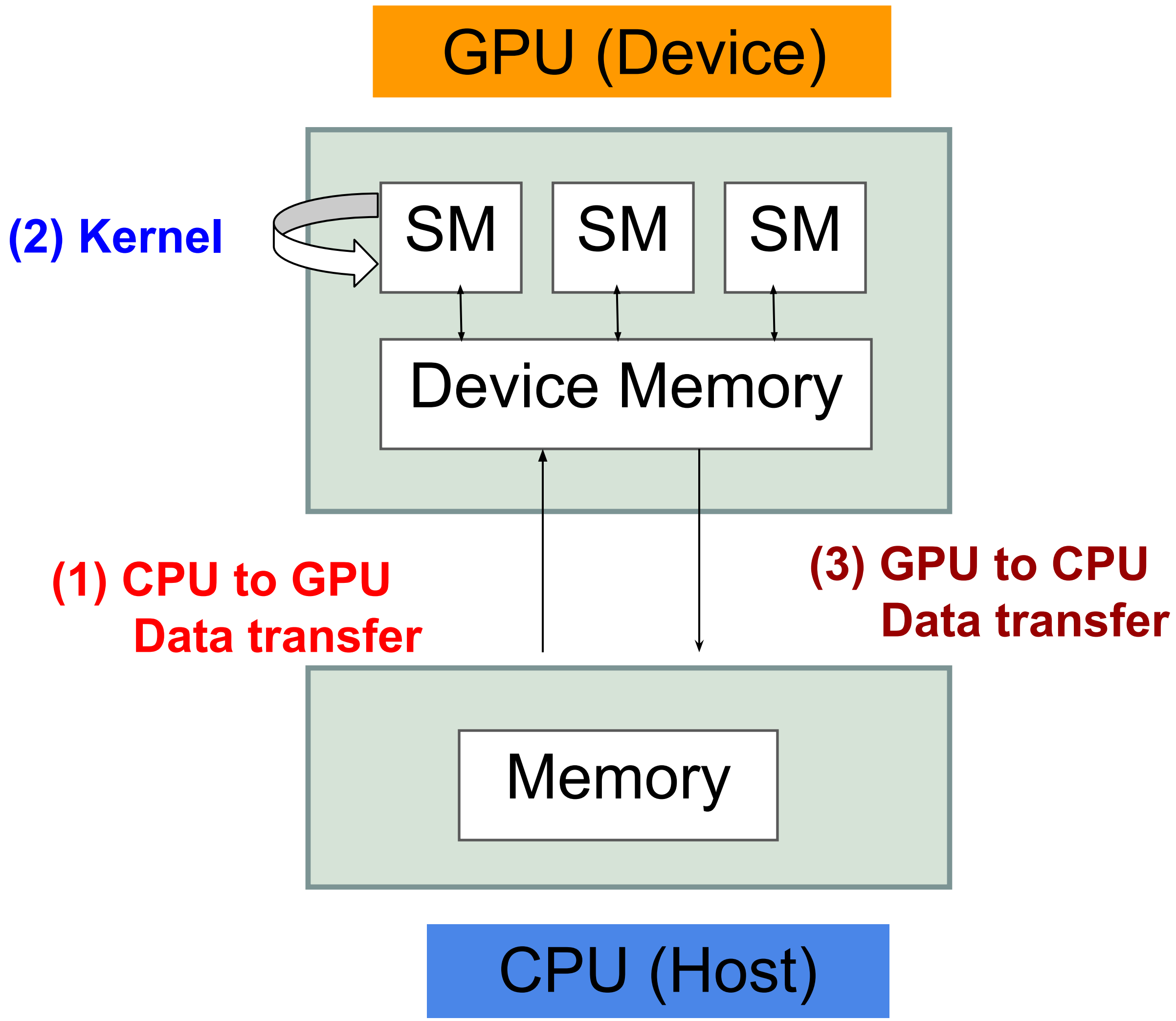
CPU and GPU memories are separate (for discrete GPUs).

Separate Memories



- CPU and its associated (discrete) GPUs have separate physical memory (RAM).
- A variable in CPU memory cannot be accessed directly in a GPU kernel.
- A programmer needs to maintain copies of variables.
- It is programmer's responsibility to keep them in sync.

CUDA Programs with Data Transfers



Data Transfer

- Copy data from CPU to GPU

  cudaMemcpy(gpulocation, cpulocation, size, cudaMemcpyHostToDevice);

- Copy data from CPU to GPU

  cudaMemcpy(cpulocation, ppulocation, size, cudaMemcpyDeviceToHost);

This means we need two copies of the same variable – one on CPU another on GPU.

e.g., int *cpuarr, *gpuarr;

CPU-GPU Communication

```
#include <stdio.h>
#include <cuda.h>
__global__void dkernel(char *arr, int arrlen) {
        unsigned id = threadIdx.x;
        if (id < arrlen) {
                ++arr[id];
        }
}
```

```
int main() {
      char cpuarr[] = "CS516",  *gpuarr;
      cudaMalloc(&gpuarr, sizeof(char) * (1 + strlen(cpuarr)));
      cudaMemcpy(gpuarr, cpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyHostToDevice);
      dkernel<<<1, 32>>>(gpuarr, strlen(cpuarr));
      cudaDeviceSynchronize();    // unnecessary.
      cudaMemcpy(cpuarr, gpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyDeviceToHost);
      printf(cpuarr);
      return 0;
}
```

Example

```c
#include <stdio.h>

#define N 100

int main() {

    int a[N], i;

    for (i = 0; i < N; ++i)
        a[i] = i * i;

    return 0;

}
```

```c
#include        <stdio.h>
#include        <cuda.h>
#define N 100
__global__ void fun(int *a) {
        a[threadIdx.x] = threadIdx.x * threadIdx.x;
}
int main() {
        int a[N], *da;
        int i;
        cudaMalloc(&da, N * sizeof(int));
        fun<<<1, N>>>(da);
        cudaMemcpy(a, da, N * sizeof(int),
                        cudaMemcpyDeviceToHost);
        for (i = 0; i < N; ++i)
                printf("%d\n", a[i]);
        return 0;
}
```

References

- CS6023 GPU Programming
  - https://www.cse.iitm.ac.in/~rupesh/teaching/gpu/jan20/
- Miscellaneous resources from internet
- https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/