CUDA Programming

Recap

- Write a CUDA code corresponding to the following sequential C code.

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

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

Note that there is no loop here.

Classwork

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>
#define N 100
int main() {
    int a[N], i;
    for (i = 0; i < N; ++i)
        a[i] = i * i;
    return 0;
}
```

Classwork

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>
#define N 100
int main() {
    int a[N], i;
    for (i = 0; i < N; ++i)
        a[i] = i * i;
    return 0;
}
```

```
#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;
}
```

Classwork

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>
#define N 100
int main() {
    int a[N], i;
    for (i = 0; i < N; ++i)
        a[i] = i * i;
    return 0;
}
```

Observation

No cudaDeviceSynchronize required.

```
#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;
}
```

Hello World with a Global.

```
#include <stdio.h>
#include <cuda.h>
const char *msg = "Hello World.\n";
__global__ void dkernel() {
    // no-op
}
int main() {
    printf(msg);
    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;
}
```

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

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).

GPU Hello World with a Global.

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

Takeaway

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

GPU Hello World with a Global.

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

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

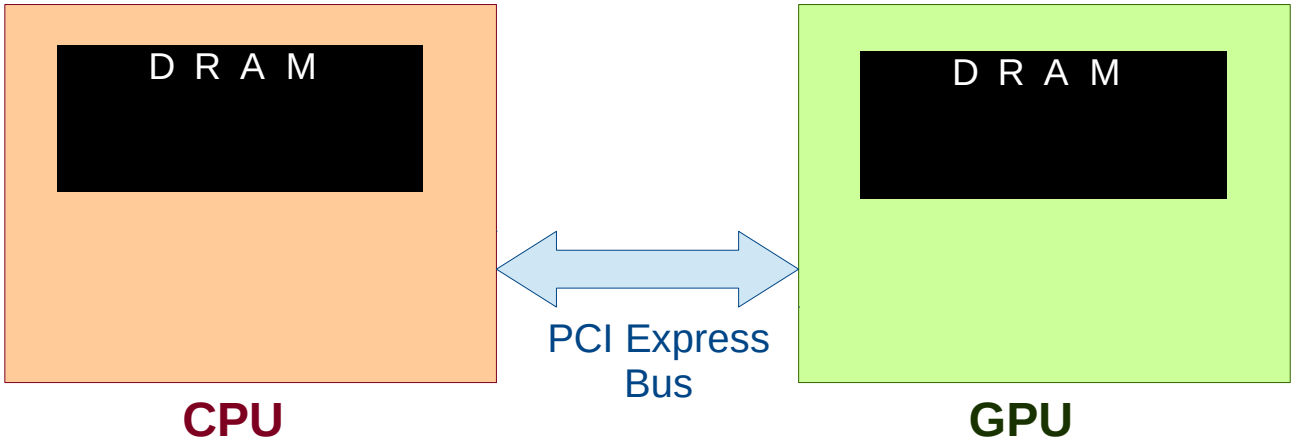
Takeaway

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

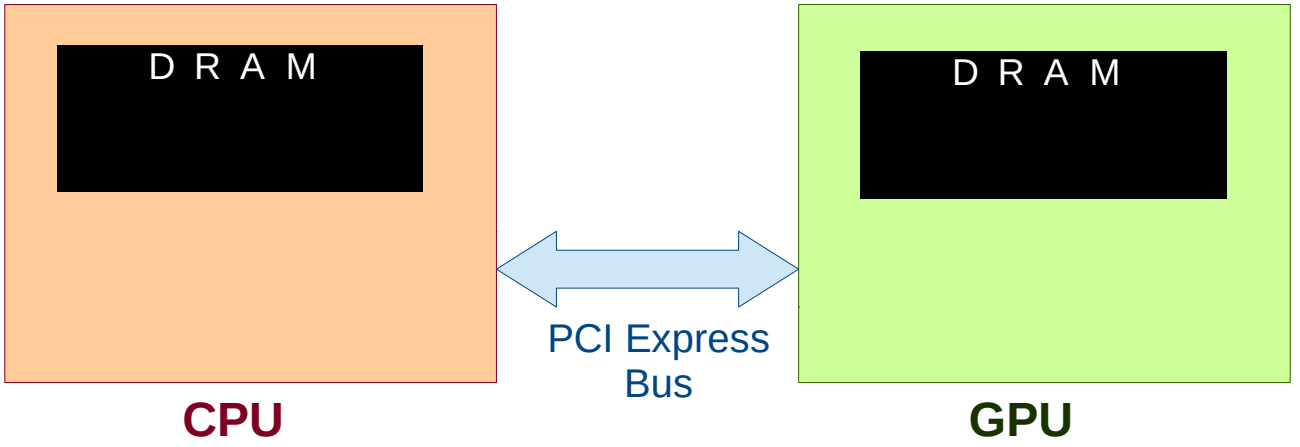
#define msg “Hello World.\n”

is okay.

Separate Memories

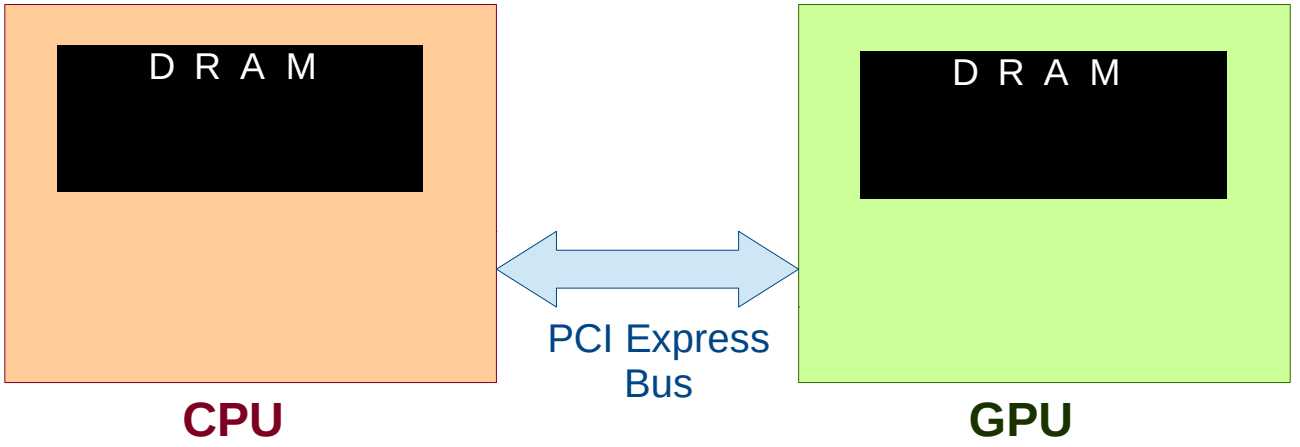


Separate Memories



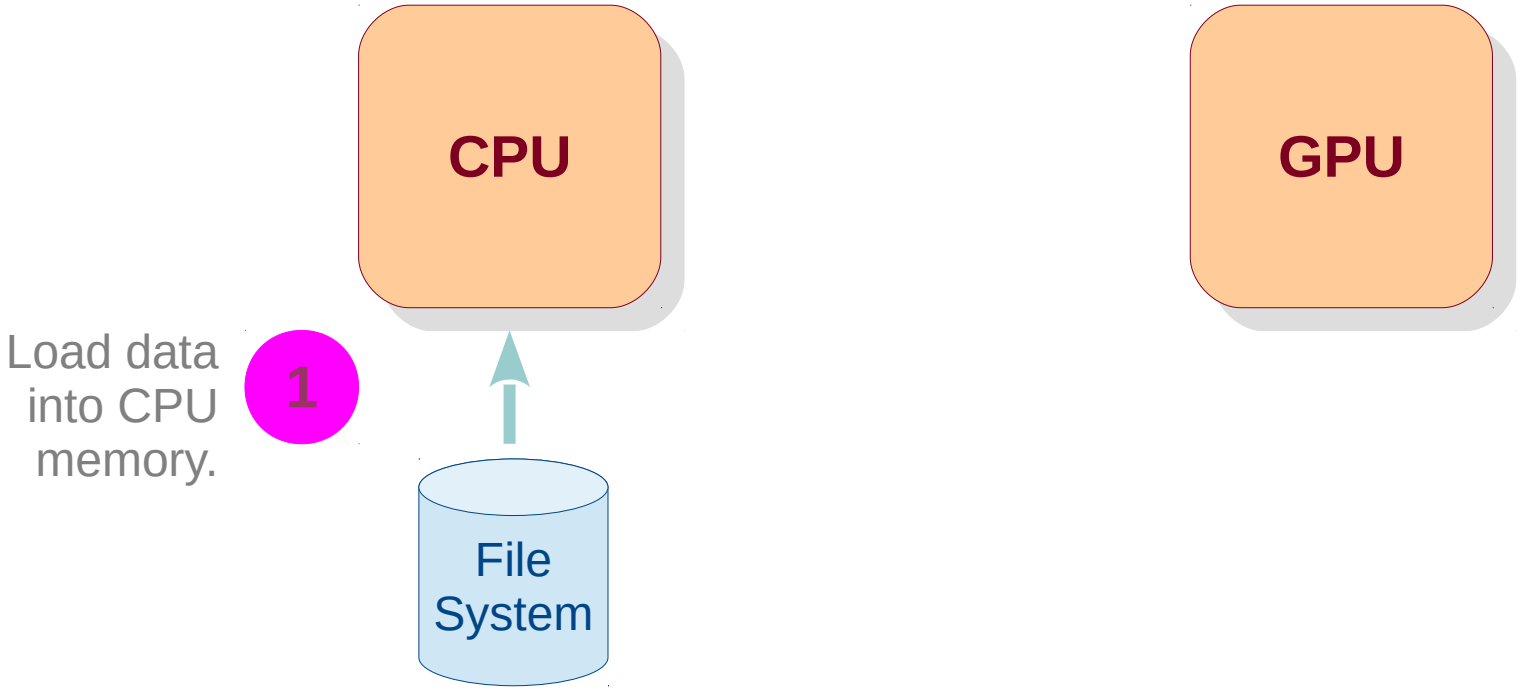
- CPU and its associated (discrete) GPUs have separate physical memory (RAM).
- A variable in CPU memory cannot be accessed directly in a GPU kernel.

Separate Memories

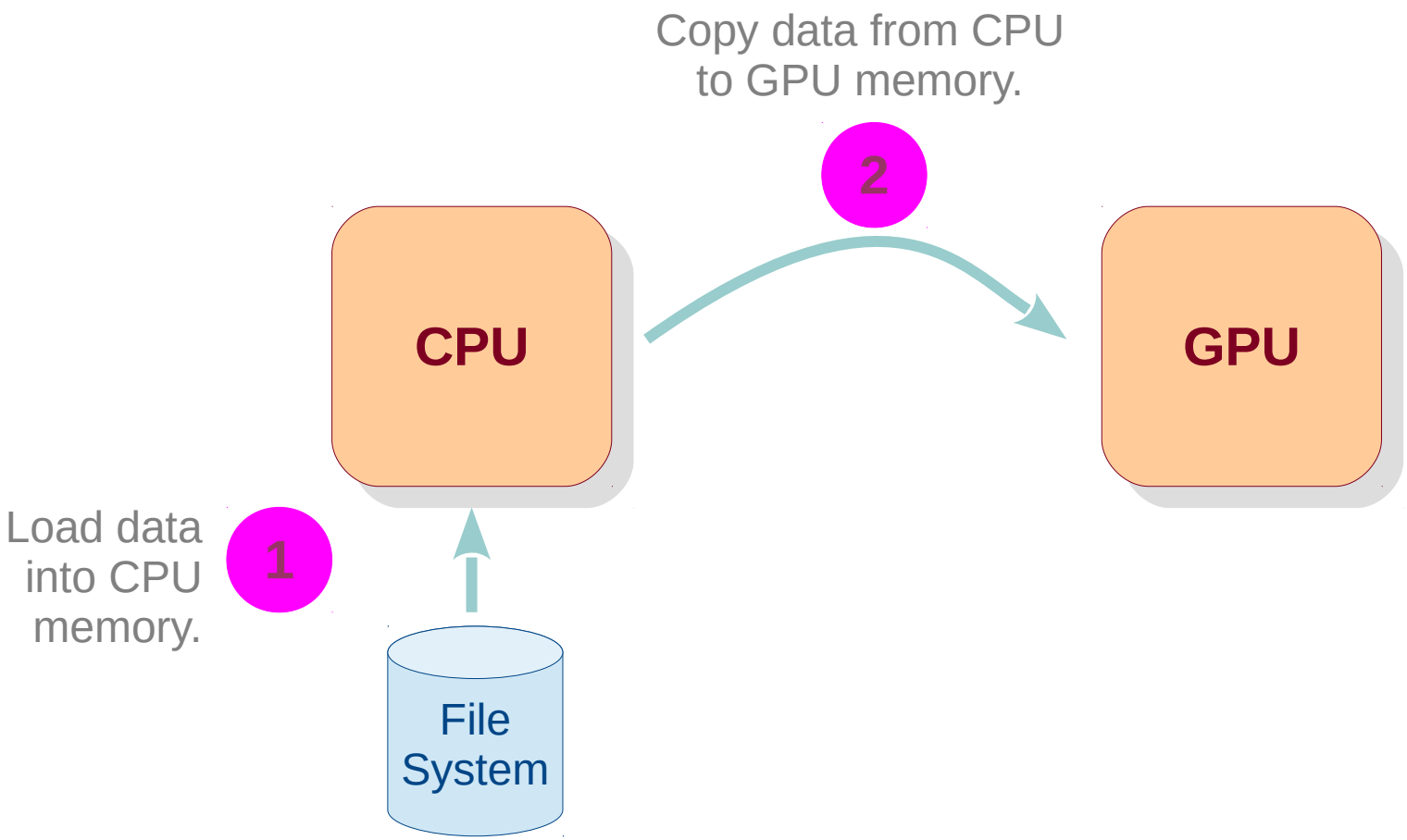


- CPU and its associated (discrete) GPUs have separate physical memory (RAM).
- A variable in CPU memory cannot be accessed directly in a GPU kernel.
- A programmer needs to maintain copies of variables.
- It is programmer's responsibility to keep them in sync.

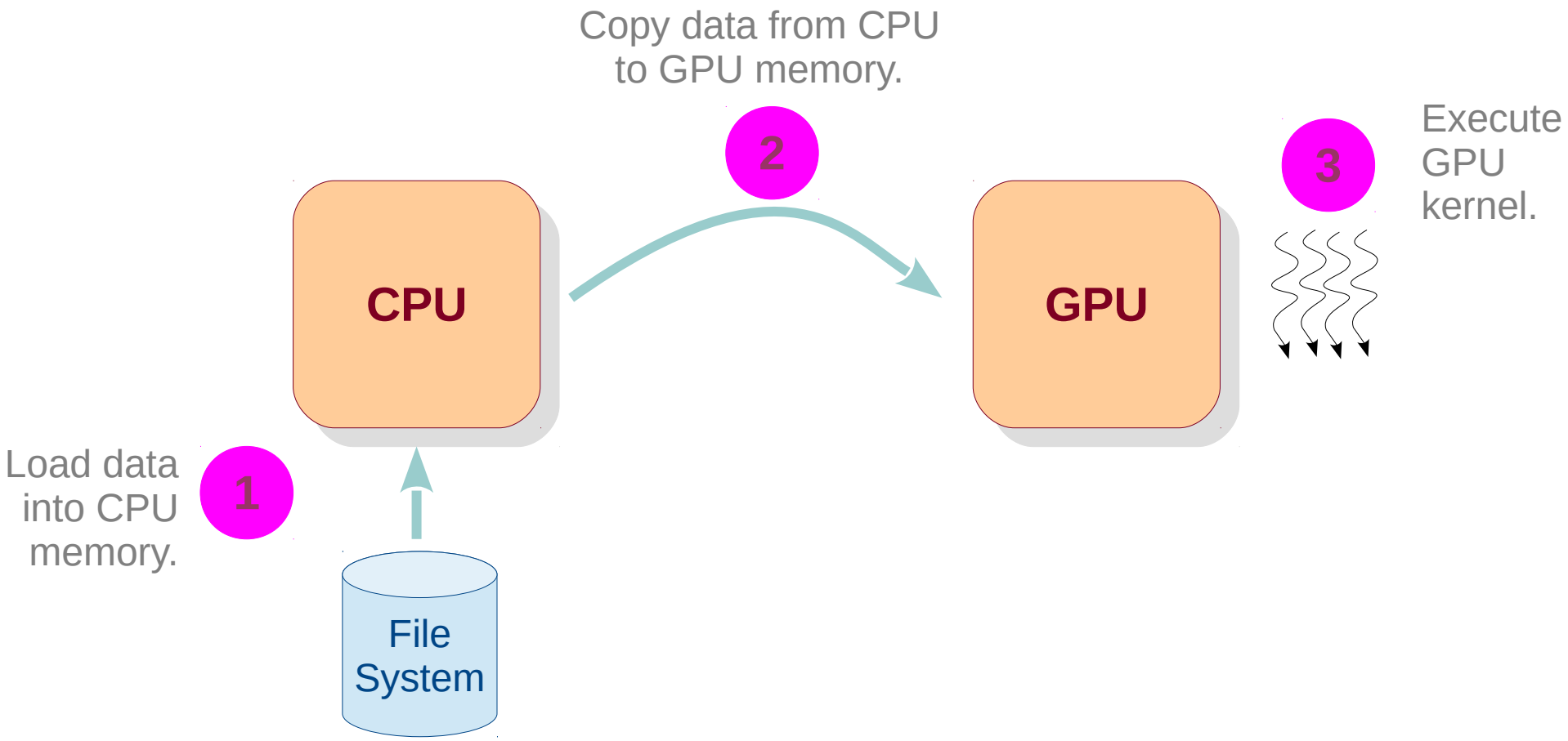
Typical CUDA Program Flow



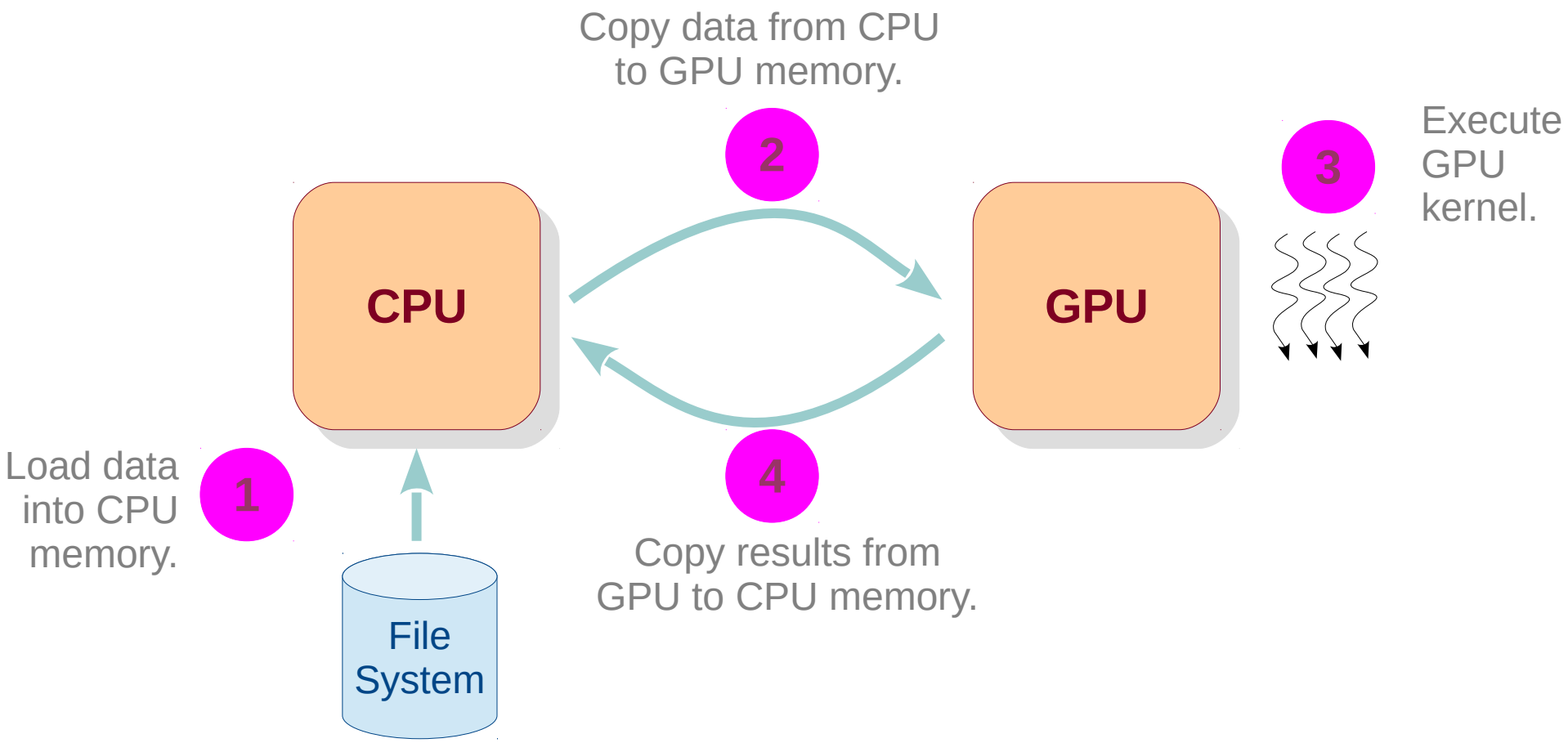
Typical CUDA Program Flow



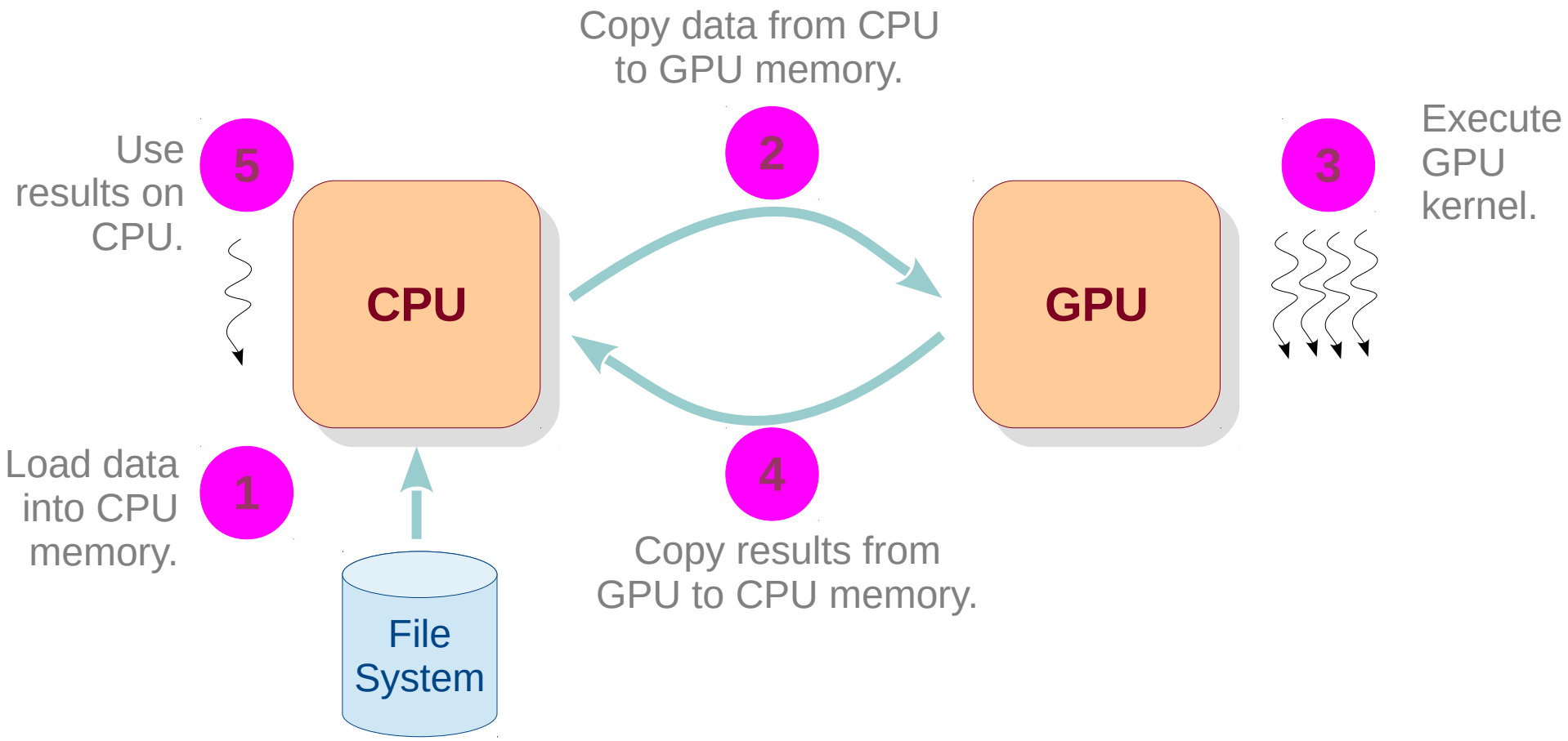
Typical CUDA Program Flow



Typical CUDA Program Flow



Typical CUDA Program Flow



Typical CUDA Program Flow

1. Load data into CPU memory.
   - fread / rand
2. Copy data from CPU to GPU memory.
   - cudaMemcpy(..., cudaMemcpyHostToDevice)
3. Call GPU kernel.
   - mykernel<<<x, y>>>(...)
4. Copy results from GPU to CPU memory.
   - cudaMemcpy(..., cudaMemcpyDeviceToHost)
5. Use results on CPU.

Typical CUDA Program Flow

2. Copy data from CPU to GPU memory.

   - cudaMemcpy(..., cudaMemcpyHostToDevice)

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

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

Matrix cpumat, gpumat;

Graph cpug, gpug;

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[] = "Gdkkn\x1fVnqkc-",
         *gpuarr;

    cudaMalloc(&gpuarr, sizeof(char) * (1 + strlen(cpuarr)));
    cudaMemcpy(gpuarr, cpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyHostToDevice);
    dkernel<<<1, 32>>>(gpuarr, strlen(cpuarr) + 1 );
    cudaDeviceSynchronize();        // unnecessary, but okay.
    cudaMemcpy(cpuarr, gpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyDeviceToHost);
    printf(cpuarr);

    return 0;
}
```

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[] = "Gdkkn\x1fVnqkc-",
         *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, but okay.
    cudaMemcpy(cpuarr, gpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyDeviceToHost);
    printf(cpuarr);

    return 0;
}
```

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.
3. Create another kernel that adds *i* to *array[i]*.

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.
3. Create another kernel that adds *i* to *array[i]*.
4. Change the array size to 8000.
5. Check if answer to problem 3 still works.

Homework ($z = x^2 + y^3$)

- Read a sequence of integers from a file.
- Square each number.
- Read another sequence of integers from another file.
- Cube each number.
- Sum the two sequences element-wise, store in the third sequence.
- Print the computed sequence.