

CPU GPU Data Transfer Bandwidth using CUDA

Ali K. Rahimian

November 30, 2025

1 Objective

The goal of this experiment is to measure the effective bandwidth between host (CPU) memory and device (GPU) memory using CUDA. In particular, we measure:

- Host to Device (H2D) transfer bandwidth.
- Device to Host (D2H) transfer bandwidth.
- The effect of host memory type: pageable vs pinned memory.

The experiment helps characterize the cost of data movement in GPU accelerated applications and motivates the use of pinned memory and large batched transfers.

2 Background

2.1 Host and device memory

In CUDA, the CPU is referred to as the *host* and the GPU as the *device*. Each has its own physically separate memory:

- Host memory is allocated with standard mechanisms such as `malloc` or `new`.
- Device memory is allocated with `cudaMalloc` and resides in the GPU's global memory.

Data must be explicitly transferred between host and device using `cudaMemcpy` in order for GPU kernels to operate on host initialized data or for the host to read back results.

2.2 H2D vs D2H

We distinguish two transfer directions:

- H2D (Host to Device): data moves from host RAM to GPU memory.
This uses `cudaMemcpyHostToDevice`.
- D2H (Device to Host): data moves from GPU memory back to host RAM.
This uses `cudaMemcpyDeviceToHost`.

The effective bandwidth in each direction can differ due to hardware, driver, and caching behaviors.

2.3 Pageable vs pinned host memory

Host memory used in transfers can be either:

- **Pageable memory:** standard host memory allocated with `malloc` or `new`. The operating system is allowed to move this memory in physical RAM or swap it out.
- **Pinned (page locked) memory:** host memory allocated with `cudaHostAlloc` or `cudaMallocHost`. The memory is locked in physical RAM and cannot be swapped out or relocated.

For pageable memory, the CUDA driver typically performs an extra copy from the user buffer into an internal pinned buffer before using DMA to transfer data to or from the GPU. Pinned memory can be used directly by the DMA engine, so transfers from pinned memory usually have higher and more stable bandwidth, especially for large buffers.

3 Methodology

3.1 Bandwidth measurement

We allocate large contiguous arrays on the host and on the device and use `cudaMemcpy` to transfer data between them. Timing is done using CUDA events:

1. Create start and stop events with `cudaEventCreate`.
2. Record a warm up transfer to reduce first use overheads.
3. Record the start event.
4. Perform a given number of repeated `cudaMemcpy` calls.
5. Record the stop event and synchronize.
6. Use `cudaEventElapsedTime` to obtain the elapsed time in milliseconds.

The effective bandwidth is then computed as

$$\text{Bandwidth (GB/s)} = \frac{\text{Total bytes transferred}}{\text{Elapsed time in seconds} \times 10^9}.$$

For each array size, we repeat the transfer `num_iters` times and compute bandwidth using the total bytes over all iterations.

3.2 Array sizes and configurations

We test array sizes from 1 MB up to 1 GB in powers of two. Specifically, the sizes are

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 MB.

For each size, we measure four configurations:

- H2D using pageable host memory.
- D2H using pageable host memory.
- H2D using pinned host memory.
- D2H using pinned host memory.

Host arrays are initialized with simple dummy data (sequential floating point values) and device arrays are allocated with `cudaMalloc`. Pageable host memory is allocated with `malloc`, while pinned memory is allocated with `cudaHostAlloc`.

Size (MB)	Pageable		Pinned	
	H2D	D2H	H2D	D2H
1	3.846	2.118	2.506	4.864
2	11.652	6.564	9.892	8.520
4	13.352	7.090	10.056	8.928
8	19.567	12.200	22.272	15.912
16	20.608	12.727	23.510	15.352
32	19.644	13.076	12.188	15.112
64	8.120	9.064	15.961	15.407
128	5.152	13.316	16.913	15.363
256	4.525	13.428	17.027	15.364
512	4.494	13.397	17.120	15.197
1024	13.679	13.396	17.392	15.021

Table 1: Measured H2D and D2H bandwidth (GB/s) for pageable and pinned host memory across different transfer sizes.

4 Results

4.1 Tabulated measurements

The raw CSV output from the program is summarized in Table 1. For each transfer size, the table reports the effective bandwidth in GB/s for both directions (H2D and D2H) and both host memory types (pageable and pinned).

4.2 Bandwidth vs array size plot

Using a Python script with `matplotlib`, the CSV file is converted into a plot of bandwidth versus array size. The plot contains four curves:

- H2D pageable.
- D2H pageable.
- H2D pinned.
- D2H pinned.

Figure 1 shows a representative result.

5 Discussion

For very small array sizes (for example 1 to 4 MB), the measured bandwidth is relatively low and exhibits more variability. In this regime, fixed overheads such as driver calls, event handling, and setup costs dominate the total transfer time, so the effective bandwidth is limited by latency rather than by the hardware’s peak throughput.

As the array size increases into the medium range (around 8 to 32 MB), the bandwidth increases and approaches a plateau. For these sizes the measurements begin to reflect the true throughput of the CPU GPU interconnect and the memory subsystem. Pinned memory already shows an advantage over pageable memory, especially for H2D transfers.

For large array sizes (for example 64 MB and above), pinned memory provides higher and more stable bandwidth values. In the example results:

- H2D pinned transfers for large sizes reach roughly 16 to 17 GB/s.

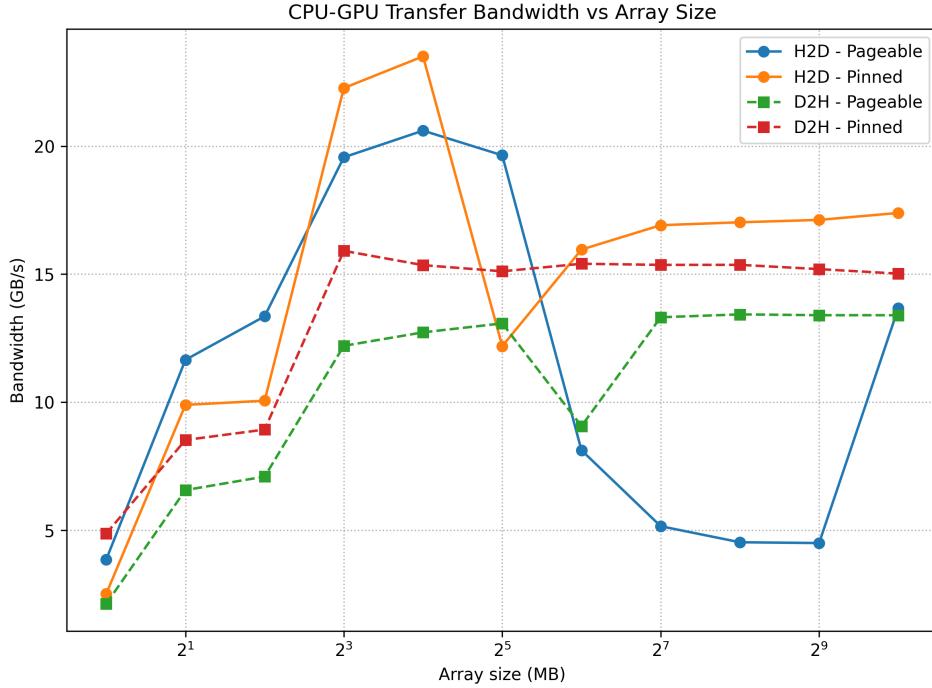


Figure 1: CPU GPU transfer bandwidth vs array size for H2D and D2H, using pageable and pinned host memory.

- D2H pinned transfers are slightly lower, around 15 to 16 GB/s.
- Pageable D2H transfers stabilize around approximately 13 GB/s.

Pageable H2D bandwidth is noticeably lower and more variable, which is consistent with the extra staging copy required by the driver when using pageable host memory. Pinned memory avoids this internal copy and allows the GPU to perform DMA directly from or to the locked pages, which explains the higher sustained bandwidth.

The small asymmetry between H2D and D2H bandwidths is expected and can arise from hardware details, caching, and driver implementation. Overall, the results support the typical recommendation to use pinned memory for performance critical large transfers and to batch small transfers into larger ones when possible.

6 Source Code

This section lists the full source code used to perform the measurements and generate the plot.

6.1 cuda_utils.cuh

```
#ifndef CUDA_UTILS_CUH
#define CUDA_UTILS_CUH

#include <cstdio>
#include <cstdlib>
#include <cuda_runtime.h>

#define CHECK_CUDA(call) \
    do { \
        cudaError_t err = (call); \
        if (err != cudaSuccess) { \
            fprintf(stderr, "CUDA error %s\n", \
                cudaGetErrorString(err)); \
            exit(1); \
        } \
    } while (0)
```

```

    if (err != cudaSuccess) { \
        fprintf(stderr, "CUDA error at %s:%d: %s\n", \
            __FILE__, __LINE__, cudaGetErrorString(err)); \
        std::exit(EXIT_FAILURE); \
    } \
} while (0)

#endif // CUDA_UTILS_CUH

```

6.2 bandwidth.h

```

#ifndef BANDWIDTH_H
#define BANDWIDTH_H

#include <cstddef>
#include <vector>

struct BandwidthResult {
    double sizeMB; // Array size in MB
    double h2d_pageable; // GB/s
    double d2h_pageable; // GB/s
    double h2d_pinned; // GB/s
    double d2h_pinned; // GB/s
};

/**
 * Run bandwidth tests for a list of sizes.
 *
 * @param sizes Array sizes in bytes.
 * @param num_iters Number of repetitions per measurement.
 * @param results Output vector of BandwidthResult, one per size.
 */
void run_bandwidth_tests(const std::vector<size_t>& sizes,
                        int num_iters,
                        std::vector<BandwidthResult>& results);

#endif // BANDWIDTH_H

```

6.3 bandwidth.cu

```

#include "bandwidth.h"
#include "cuda_utils.cuh"

#include <cstdlib>

// Measure bandwidth for a single direction and memory pairing
static double measure_bandwidth_single_direction(
    void* dst,
    const void* src,
    size_t bytes,
    cudaMemcpyKind kind,
    int num_iters)
{
    cudaEvent_t start, stop;
    CHECK_CUDA(cudaEventCreate(&start));

```

```

CHECK_CUDA(cudaEventCreate(&stop));

// Warm up to avoid first use overhead
CHECK_CUDA(cudaMemcpy(dst, src, bytes, kind));

CHECK_CUDA(cudaEventRecord(start, 0));
for (int i = 0; i < num_iters; ++i) {
    CHECK_CUDA(cudaMemcpy(dst, src, bytes, kind));
}
CHECK_CUDA(cudaEventRecord(stop, 0));
CHECK_CUDA(cudaEventSynchronize(stop));

float ms = 0.0f;
CHECK_CUDA(cudaEventElapsedTime(&ms, start, stop));

CHECK_CUDA(cudaEventDestroy(start));
CHECK_CUDA(cudaEventDestroy(stop));

double seconds = ms / 1000.0;
double total_bytes = static_cast<double>(bytes) * num_iters;

// Bandwidth in GB/s
double gb_per_s = total_bytes / (seconds * 1.0e9);
return gb_per_s;
}

// Run test for one size, filling in the 4 bandwidth numbers
static void run_test_for_size(size_t bytes,
                               int num_iters,
                               double& h2d_pageable,
                               double& d2h_pageable,
                               double& h2d_pinned,
                               double& d2h_pinned)
{
    // -----
    // Pageable host memory
    // -----
    {
        // Host allocation (pageable)
        float* h_buf = static_cast<float*>(std::malloc(bytes));
        if (!h_buf) {
            std::fprintf(stderr,
                         "Failed to allocate pageable host memory (%zu bytes)\n",
                         bytes);
            std::exit(EXIT_FAILURE);
        }

        // Initialize host data
        size_t n = bytes / sizeof(float);
        for (size_t i = 0; i < n; ++i) {
            h_buf[i] = static_cast<float>(i);
        }

        // Device allocation
        float* d_buf = nullptr;
        CHECK_CUDA(cudaMalloc(&d_buf, bytes));

        // Measure H2D pageable

```

```

h2d_pageable = measure_bandwidth_single_direction(
    d_buf, h_buf, bytes, cudaMemcpyHostToDevice, num_iters);

// Measure D2H pageable
d2h_pageable = measure_bandwidth_single_direction(
    h_buf, d_buf, bytes, cudaMemcpyDeviceToHost, num_iters);

CHECK_CUDA(cudaFree(d_buf));
std::free(h_buf);
}

// -----
// Pinned host memory
// -----
{
    // Host allocation (pinned)
    float* h_buf = nullptr;
    CHECK_CUDA(cudaHostAlloc(reinterpret_cast<void**>(&h_buf),
                           bytes,
                           cudaHostAllocDefault));

    // Initialize host data
    size_t n = bytes / sizeof(float);
    for (size_t i = 0; i < n; ++i) {
        h_buf[i] = static_cast<float>(i);
    }

    // Device allocation
    float* d_buf = nullptr;
    CHECK_CUDA(cudaMalloc(&d_buf, bytes));

    // Measure H2D pinned
    h2d_pinned = measure_bandwidth_single_direction(
        d_buf, h_buf, bytes, cudaMemcpyHostToDevice, num_iters);

    // Measure D2H pinned
    d2h_pinned = measure_bandwidth_single_direction(
        h_buf, d_buf, bytes, cudaMemcpyDeviceToHost, num_iters);

    CHECK_CUDA(cudaFree(d_buf));
    CHECK_CUDA(cudaFreeHost(h_buf));
}
}

void run_bandwidth_tests(const std::vector<size_t>& sizes,
                        int num_iters,
                        std::vector<BandwidthResult>& results)
{
    results.clear();
    results.reserve(sizes.size());

    for (size_t bytes : sizes) {
        double h2d_pageable = 0.0;
        double d2h_pageable = 0.0;
        double h2d_pinned = 0.0;
        double d2h_pinned = 0.0;

        run_test_for_size(bytes, num_iters,

```

```

        h2d_pageable, d2h_pageable,
        h2d_pinned, d2h_pinned);

    BandwidthResult r;
    r.sizeMB = static_cast<double>(bytes) / (1024.0 * 1024.0);
    r.h2d_pageable = h2d_pageable;
    r.d2h_pageable = d2h_pageable;
    r.h2d_pinned = h2d_pinned;
    r.d2h_pinned = d2h_pinned;

    results.push_back(r);
}
}

```

6.4 main.cu

```

#include <cstdio>
#include <vector>

#include "cuda_utils.cuh"
#include "bandwidth.h"

int main(int argc, char** argv)
{
    // Use device 0 by default
    CHECK_CUDA(cudaSetDevice(0));

    // Sizes from 1 MB to 1 GB in powers of 2:
    // 1, 2, 4, ..., 1024 MB
    const int num_sizes = 11;
    std::vector<size_t> sizes;
    sizes.reserve(num_sizes);

    for (int i = 0; i < num_sizes; ++i) {
        size_t bytes = static_cast<size_t>(1) << (20 + i); // 2^(20+i)
        sizes.push_back(bytes);
    }

    // Number of repetitions per measurement
    int num_iters = 10;

    std::vector<BandwidthResult> results;
    run_bandwidth_tests(sizes, num_iters, results);

    // CSV header
    std::printf("size_MB,h2d_pageable_GBs,d2h_pageable_GBs,h2d_pinned_GBs,
               d2h_pinned_GBs\n");

    for (const auto& r : results) {
        std::printf("%.0f,%.3f,%.3f,%.3f,%.3f\n",
                   r.sizeMB,
                   r.h2d_pageable,
                   r.d2h_pageable,
                   r.h2d_pinned,
                   r.d2h_pinned);
    }
}

```

```

    CHECK_CUDA(cudaDeviceReset());
    return 0;
}

```

6.5 Makefile

```

NVCC := nvcc
NVCC_FLAGS := -O2

SRCS := main.cu bandwidth.cu
HDRS := bandwidth.h cuda_utils.cuh

TARGET := bandwidth

$(TARGET): $(SRCS) $(HDRS)
    $(NVCC) $(NVCC_FLAGS) $(SRCS) -o $(TARGET)

clean:
    rm -f $(TARGET) bandwidth.csv bandwidth_plot.png

```

6.6 plot_bandwidth.py

```

import numpy as np
import matplotlib.pyplot as plt

# Load CSV data produced by ./bandwidth > bandwidth.csv
data = np.loadtxt('bandwidth.csv', delimiter=',', skiprows=1)

size_mb = data[:, 0]
h2d_pageable = data[:, 1]
d2h_pageable = data[:, 2]
h2d_pinned = data[:, 3]
d2h_pinned = data[:, 4]

plt.figure(figsize=(8, 6))

# H2D curves
plt.plot(size_mb, h2d_pageable, 'o--', label='H2D - Pageable')
plt.plot(size_mb, h2d_pinned, 'o--', label='H2D - Pinned')

# D2H curves
plt.plot(size_mb, d2h_pageable, 's--', label='D2H - Pageable')
plt.plot(size_mb, d2h_pinned, 's--', label='D2H - Pinned')

plt.xscale('log', base=2)
plt.xlabel('Array size (MB)')
plt.ylabel('Bandwidth (GB/s)')
plt.title('CPU-GPU Transfer Bandwidth vs Array Size')
plt.grid(True, which='both', linestyle=':')
plt.legend()
plt.tight_layout()
plt.savefig('bandwidth_plot.png', dpi=300)
plt.show()

```