Katholieke
Universiteit
Leuven

**Department of
Computer Science**

GEAVANCEERDE COMPUTERARCHITECTUUR

Lab 4

Thibaut Beck
Wannes Paesschesoone

Academic year 2025–2026

Contents

1 Part 1: Coalesced vs. Uncoalesced Memory Access

In the CUDA execution model, threads are organized into warps of 32. The GPU executes instructions for each warp in a SIMT (Single Instruction, Multiple Threads) fashion. When accessing global memory, the GPU attempts to coalesce the memory requests from all threads in a warp into the minimum number of 32-byte transactions.

This process is most efficient when consecutive threads access consecutive memory locations. For example, if every thread in a warp requests a 4-byte value, and these values are all next to each other in memory, the GPU can satisfy all 32 requests with just four 32-byte transactions (a total of 128 bytes). This is a coalesced memory access pattern and it makes optimal use of memory bandwidth.

Conversely, if threads access memory locations that are far apart (for example striding), the hardware may be forced to issue a separate 32-byte transaction for each thread's request, even if each thread only needs a small part of that 32-byte segment. This leads to a significant waste of bandwidth and slower performance. In this part of the lab, we will investigate the performance impact of these two access patterns.

1.1 Overview

We implemented two CUDA kernels that invert the red channel of an image, one using coalesced memory access and the other using uncoalesced access. The coalesced kernel operates on a planar image format where all red pixels are stored consecutively, while the uncoalesced kernel works on an interleaved format (RGBRGB...).

1.2 Results

To measure the performance difference, we benchmarked the two kernels across 100 runs for various input sizes and thread block configurations. For this benchmark we took the second and third execution as result. The input image is 960x1280 pixels, and we tested it at its original size (1x), as well as scaled up by 2x and 4x. The tables below show the average execution time in milliseconds. Figure 1 summarizes the results graphically.

From the results, we can observe a few key points:

- The coalesced memory access pattern is consistently faster than the uncoalesced one, with a speedup factor typically around 1.5x to 1.7x for threads per block between 128 and 512.
- For each input size, there appears to be an optimal number of threads per block. For our tests, using 128, 256 or 512 threads per block gives the best performance for the coalesced kernel.
- As the input size increases, the absolute time saved by using coalesced access becomes more significant.

Table 1: Benchmark Results for 1x Image Size (1,228,800 pixels)

Threads/Block	Uncoalesced (ms)	Coalesced (ms)	Speedup
64	0.0428	0.0356	1.20x
128	0.0353	0.0227	1.55x
256	0.0338	0.0199	1.70x
512	0.0371	0.0211	1.76x
1024	0.0417	0.0265	1.57x

Table 2: Benchmark Results for 2x Image Size (2,457,600 pixels)

Threads/Block	Uncoalesced (ms)	Coalesced (ms)	Speedup
64	0.0644	0.0564	1.14x
128	0.0590	0.0367	1.61x
256	0.0568	0.0372	1.53x
512	0.0554	0.0385	1.44x
1024	0.0670	0.0542	1.24x

Table 3: Benchmark Results for 4x Image Size (4,915,200 pixels)

Threads/Block	Uncoalesced (ms)	Coalesced (ms)	Speedup
64	0.1153	0.0965	1.19x
128	0.1062	0.0627	1.69x
256	0.1048	0.0639	1.64x
512	0.1072	0.0649	1.65x
1024	0.1169	0.0919	1.27x

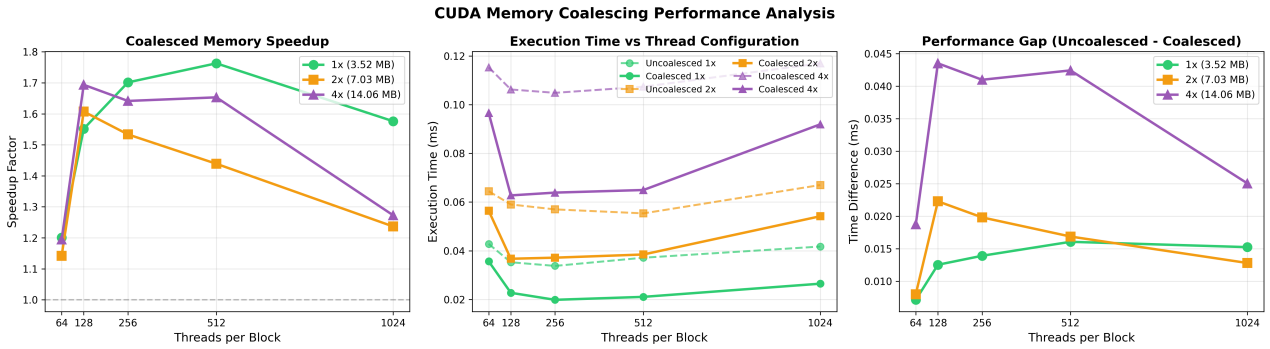

Figure 1: Results Summary

1.3 Effect of Input, Block, and Grid Size

The number of threads per block affects the GPU's ability to schedule and overlap warps to hide memory latency. The total number of blocks, known as the grid size, is determined by the input size and the number of threads we choose for each block.

Our benchmark results show a clear pattern related to the block size:

- **Small Blocks (64 threads):** Using only 64 threads per block gives the GPU too little work to do at once. If these threads have to wait for data, the GPU doesn't have enough other tasks to switch to, leading to wasted time and lower performance.
- **Optimal Blocks (128-512 threads):** Block sizes in this range provide a good balance. There are enough threads for the GPU to effectively switch between tasks, hiding the time spent waiting for memory. This keeps the hardware busy and leads to the best performance, as seen in our results.
- **Large Blocks (1024 threads):** While it might seem like more threads are always better, performance drops with 1024 threads. This is because a very large block uses up too many of the GPU's limited resources (like registers). This can prevent the GPU from running as many blocks in parallel, which hurts overall performance.

1.4 Code

The following kernels invert the red channel of an image.

1.4.1 Coalesced Kernel

```
// Operates on a PLANAR array (RRR...GGG...BBB...)
__global__ void invert_red_coalesced(uint8_t* planar_input, int num_pixels) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_pixels) {
        planar_input[idx] = 255 - planar_input[idx];
    }
}
```

Listing 1: Coalesced Memory Access

1.4.2 Uncoalesced Kernel

```
// Operates on an INTERLEAVED array (RGBRGBRGB...)
__global__ void invert_red_uncoalesced(uint8_t* interleaved_input, int num_pixels) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_pixels) {
        int pixel_index = idx * 3;
        interleaved_input[pixel_index] = 255 - interleaved_input[pixel_index];
    }
}
```

Listing 2: UnCoalesced Memory Access

1.5 Conclusion

This analysis shows that coalesced memory access patterns significantly improve performance in CUDA applications. By ensuring that threads within a warp access consecutive memory locations, we can reduce the number of memory transactions and make better use of the available bandwidth. Additionally, choosing an appropriate block size is crucial for maximizing GPU utilization. Blocks that are too small fail to hide memory latency effectively, while excessively large blocks can exhaust hardware resources and limit parallelism. For our tests, a block size between 128 and 512 threads was the sweet spot.

2 Part 2: Global, Shared, and Constant Memory for Matrix Multiplication

2.1 Overview

In this part of the lab, we explore different memory optimization techniques for matrix multiplication ($C = AB$) on the GPU. We implemented and benchmarked three distinct CUDA kernels:

- A baseline kernel that relies exclusively on **global memory** for all matrix data (A, B, and C).
- An optimized kernel that uses **shared memory** to implement a tiled matrix multiplication algorithm. This approach aims to reduce the high latency of global memory access by caching sub-matrices (tiles) in the fast, on-chip shared memory, thereby improving data reuse.
- A third kernel that leverages **constant memory** to store one of the input matrices (B). Constant memory is cached and provides efficient broadcast capabilities, which can be beneficial when all threads in a warp access the same memory location.

To evaluate their performance, we timed each kernel's execution, as well as the required memory transfers between the host (CPU) and device (GPU), for a range of square matrix sizes. The results were verified for correctness against a standard CPU-based calculation.

2.2 Results

The tables below summarize the average execution times (in milliseconds) for memory transfers and kernel executions across different matrix sizes (N x N) and block sizes (BLOCK_SIZE = 4 and 16). The grid dimensions used for each configuration are also provided. Next to the tables, several figures visualize the performance differences from the results in the table.

Table 4: Average Times for BLOCK_SIZE = 16 (in ms)

N	8	16	32	64	96	192	384	768
Memory Transfers/Copies								
Host to Device (A)	0.0066	0.0057	0.0069	0.0107	0.0173	0.0930	0.2370	0.6899
Host to Device (B)	0.0053	0.0057	0.0063	0.0090	0.0147	0.0284	0.1359	0.6680
Host to Constant (B)	0.0029	0.0019	0.0017	0.0016	0.0021	(too large)	(too large)	(too large)
Load to Shared (Global to Shared)	-	0.0060	0.0037	0.0040	0.0042	0.0060	0.0105	0.0461
Kernel Execution Times								
Grid	1x1	1x1	2x2	4x4	6x6	12x12	24x24	48x48
Global Kernel	0.0044	0.0046	0.0052	0.0065	0.0094	0.0296	0.1904	1.3779
Shared Kernel	-	0.0046	0.0049	0.0060	0.0086	0.0279	0.1439	0.8480
Constant Kernel	0.0046	0.0074	0.0120	0.0243	0.0401	-	-	-

Table 5: Average Times for BLOCK_SIZE = 4 (in ms)

N	8	16	32	64	96	192	384	768
Memory Transfers/Copies								
Host to Device (A)	0.0055	0.0055	0.0060	0.0095	0.0134	0.0261	0.1449	0.4733
Host to Device (B)	0.0054	0.0054	0.0060	0.0095	0.0142	0.0234	0.1302	0.6503
Host to Constant (B)	0.0026	0.0017	0.0017	0.0017	0.0020	(too large)	(too large)	(too large)
Load to Shared (Global to Shared)	0.0034	0.0037	0.0037	0.0039	0.0054	0.0220	0.1263	0.8405
Kernel Execution Times								
Grid	2x2	4x4	8x8	16x16	24x24	48x48	96x96	192x192
Global Kernel	0.0039	0.0038	0.0046	0.0064	0.0122	0.0534	0.3762	2.8381
Shared Kernel	0.0040	0.0043	0.0051	0.0075	0.0152	0.0694	0.4715	3.4653
Constant Kernel	0.0040	0.0043	0.0053	0.0092	0.0244	-	-	-

Figure 2: Kernel Execution Times block size 4

Figure 3: Kernel Execution Times block size 16

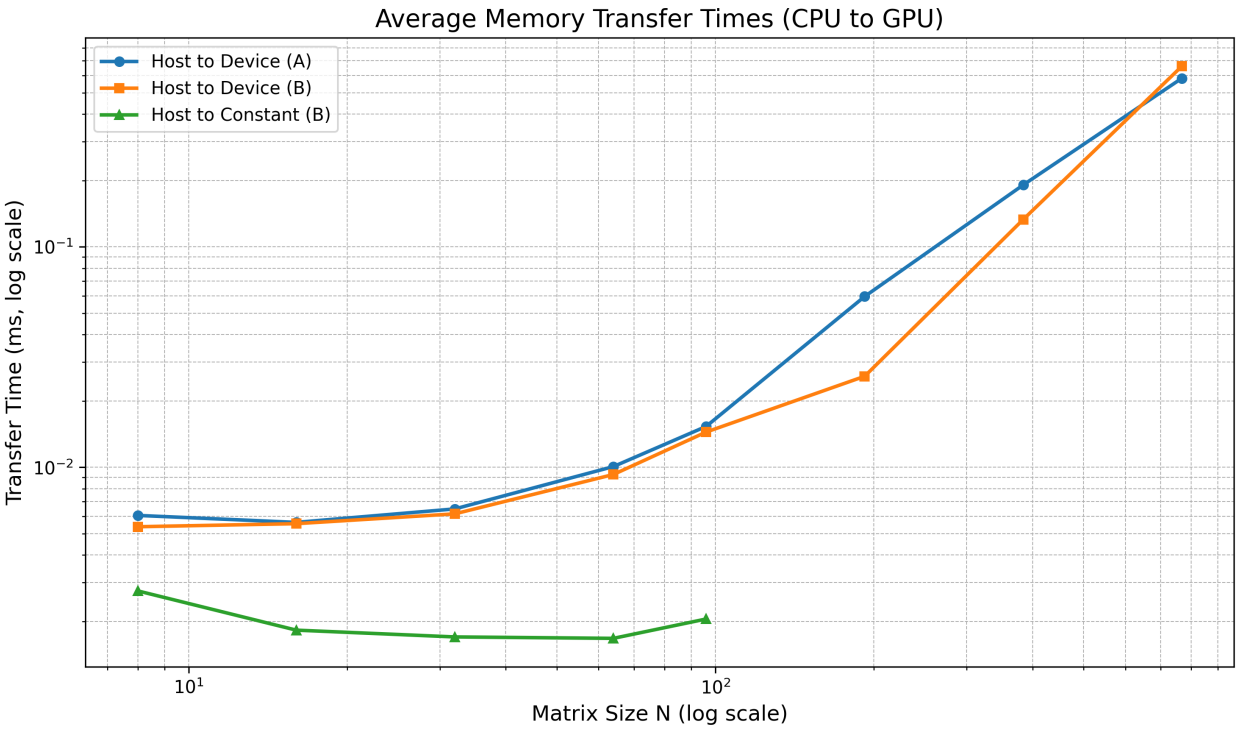

Figure 4: Memory Transfer Times

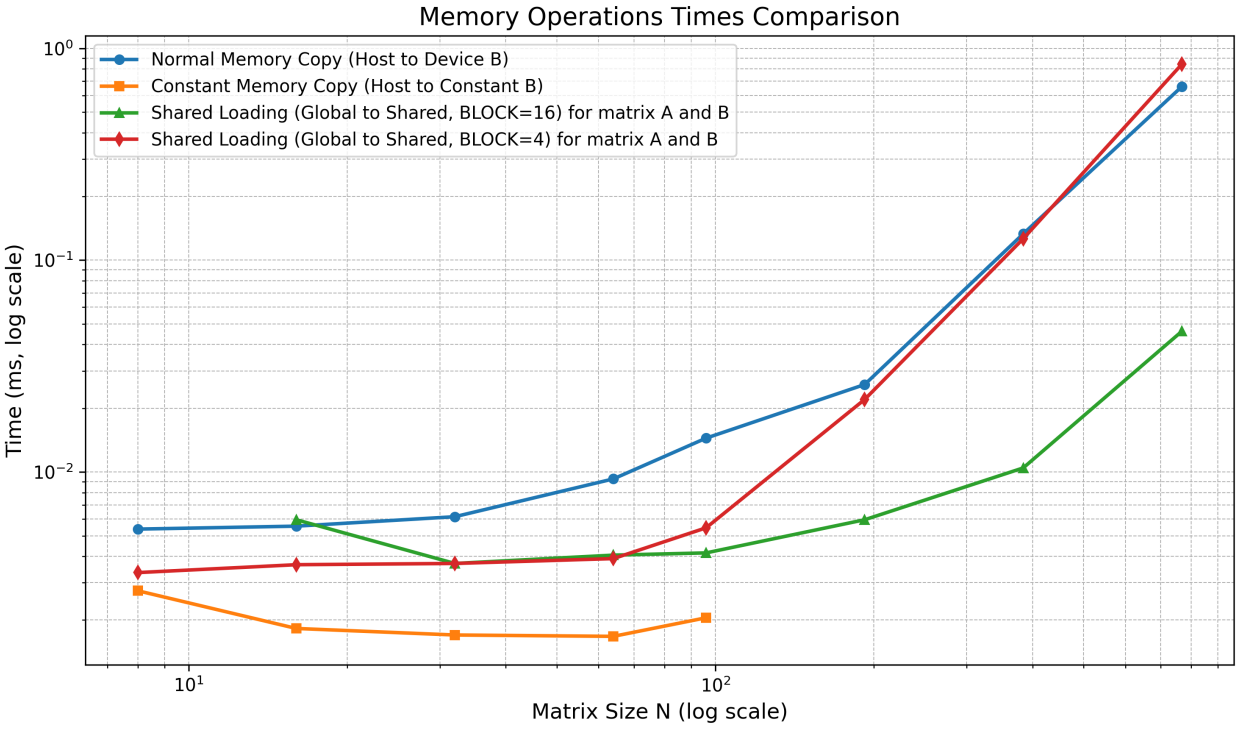

Figure 5: Memory Operations Comparison

2.3 Analysis

The performance of the three different matrix multiplication kernels (Global, Shared, and Constant) was evaluated for block sizes of 4 and 16. The analysis focuses on memory transfer times, kernel execution speeds, and the impact of block and matrix size on overall performance.

1. Memory Transfer Times As shown in the tables and visualized in Figure 4, the Host to Constant (B) transfer is notably faster for small matrices but is constrained by the 64 KB limit of constant memory, becoming unavailable for N=192 and larger. The Load to Shared time, which measures the cost of moving data from global to shared memory, also scales with matrix size. This is an overhead inherent to the tiled algorithm but is compensated by the faster access to shared memory during computation.

2. Kernel Execution Time Comparison The kernel execution times, plotted in Figures 2 and 3, reveal the strengths and weaknesses of each memory type.

- For `BLOCK_SIZE=16` the **Shared Kernel** consistently outperforms the **Global Kernel**, especially as the matrix size N increases. For `BLOCK_SIZE=16` and $N = 768$, the shared kernel is approximately $1.6\times$ faster than the global kernel (0.8480 ms vs. 1.3779 ms). This speedup is due to the tiled approach, which significantly reduces redundant global memory accesses by reusing data from the much faster shared memory.
- The **Constant Kernel** is effective for small matrices where the data fits into the constant memory cache. However, its performance degrades relative to the other kernels as N increases, and it becomes unusable for $N \geq 192$. The broadcast mechanism of constant memory is beneficial, but its small size is a major limitation.
- With `BLOCK_SIZE=4`, the shared memory kernel is surprisingly slower than the global kernel for larger matrices. This suggests that the overhead of tiling and the smaller tile size do not provide enough data reuse to overcome the cost of loading data into shared memory, making the simpler global memory approach more efficient in this specific configuration.

3. Impact of Block Size Comparing the results for `BLOCK_SIZE=16` and `BLOCK_SIZE=4` highlights the importance of block size in performance tuning.

- For the shared memory kernel, a larger block size (`BLOCK_SIZE=16`) allows for more data reuse within each tile, which better handles the cost of loading data from global memory. The smaller tiles of `BLOCK_SIZE=4` do not seem to hold enough data to be effective.
- For the global memory kernel, a smaller block size (`BLOCK_SIZE=4`) appears to yield better performance for smaller matrices. This might be related to how the GPU schedules the smaller blocks.

4. Scaling with Matrix and Grid Size All kernel execution times increase with matrix size, which is consistent with the $\mathcal{O}(N^3)$ complexity of matrix multiplication. However, the rate of increase differs. The shared memory kernel (with `BLOCK_SIZE=16`) scales better than the global kernel, as its effective use of shared memory mitigates the memory bandwidth bottleneck that becomes more pronounced at larger scales. The grid size, which is determined by N and `BLOCK_SIZE`, grows accordingly, providing more parallelism for the GPU to exploit.

2.4 Code

The code snippets below illustrate the key components of each kernel implementation.

2.4.1 Helper Functions

```
struct Matrix {
    int width, height, stride;
    int* elements;
};

__device__ int  GetElement(const Matrix A, int row, int col) {
    return A.elements[row * A.stride + col];
}
__device__ void SetElement(Matrix A, int row, int col, int value) {
    A.elements[row * A.stride + col] = value;
}
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {
    Matrix sub;
    sub.width = sub.height = BLOCK_SIZE;
    sub.stride = A.stride;
    sub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
    return sub;
}
```

Listing 3: Helper Functions for Matrix Access

2.4.2 Global Memory Kernel

```
__global__ void MatMulGlobal(Matrix A, Matrix B, Matrix C) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= C.height || col >= C.width) return;

    int sum = 0;
    for (int k = 0; k < A.width; ++k)
        sum += GetElement(A, row, k) * GetElement(B, k, col);
    SetElement(C, row, col, sum);
}
```

Listing 4: CUDA Kernel: Matrix multiplication using only Global Memory

2.4.3 Shared Memory Kernel

```
__global__ void MatMulShared(Matrix A, Matrix B, Matrix C) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    Matrix Csub = GetSubMatrix(C, by, bx);

    int sum = 0;
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        Matrix Asub = GetSubMatrix(A, by, m);
        Matrix Bsub = GetSubMatrix(B, m, bx);

        __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx] = GetElement(Asub, ty, tx);
        Bs[ty][tx] = GetElement(Bsub, ty, tx);
        __syncthreads();

        for (int e = 0; e < BLOCK_SIZE; ++e)
            sum += As[ty][e] * Bs[e][tx];
        __syncthreads();
    }

    int globalRow = by * BLOCK_SIZE + ty;
    int globalCol = bx * BLOCK_SIZE + tx;
    if (globalRow < C.height && globalCol < C.width)
        SetElement(Csub, ty, tx, sum);
```

```
}
```

Listing 5: Matrix multiplication using Shared Memory (tiling)

2.4.4 Constant Memory Kernel

```
__global__ void MatMulConstant(Matrix A, Matrix B, Matrix C) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= C.height || col >= C.width) return;

    int sum = 0;
    for (int k = 0; k < A.width; ++k)
        sum += GetElement(A, row, k) * constB[k * B.width + col];
    SetElement(C, row, col, sum);
}
```

Listing 6: Matrix multiplication using Constant Memory

2.5 Conclusion

Using shared memory improves performance for large matrices if the tile size is chosen appropriately. Constant memory can be used for small matrices due to its size limitations. Global-only approaches, while simpler, are significantly less efficient at large scales due to limited memory bandwidth reuse.

References

[1] NVIDIA Developer Blog, "Unlock GPU Performance: Global Memory Access in CUDA," https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/. Accessed 9 November 2025.

[2] NVIDIA Corporation, "CUDA C Programming Guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=matrix%20multiply. Accessed 10 November 2025.