# EXPLORING GPU MEMORY PERFORMANCE USING DIGITAL IMAGE PROCESSING ALGORITHMS

Puya Memarzia

Department of Computer Science and Engineering
Shiraz University, Shiraz, Iran
pm@memarzia.com

Farshad Khunjush

Assistant Professor, Department of Computer Science and Engineering
Shiraz University, Shiraz, Iran
fkhunjush@shirazu.ac.ir

**Abstract**

Leveraging the incredible parallel computational power of graphics processing units (GPUs) is a proven method for accelerating general applications. Efficient utilization of the GPU remains one of the greatest challenges facing programmers. The performance of GPU applications is extremely reliant on memory performance, to the point that it can be considered a critical bottleneck. This is further amplified when working with large amounts of data, which is common. In this paper, we explore several well-known data transfer and memory access methods. Our aim is to find out how they affect the performance of different applications. To do so, we first examine and specify the different techniques; then, we apply these techniques to a variety of digital image processing applications, which serve as the case study. The NVIDIA CUDA parallel programming framework serves as the foundation for our research. Our experimental results highlight the merits of each optimization method. We then use these results to categorize the benchmarks according to their behavior. We demonstrate significantly superior performance including speedups of up to 24x compared to naïve implementations and up to 157x compared to serial implementations.

***Keywords*****:** digital image processing; parallel computing; GPU computing; memory; data transfer; GPGPU; optimization; CUDA.

## 1. Introduction

This Graphics processing units (GPUs) are being increasingly used to accelerate a wide variety of applications. Recent advances in GPU architectures have not only improved the baseline performance but have also provided programmers with new options to optimize their applications for better performance. Despite these advances, writing efficient GPU applications remains a challenging task. Chief among these challenges is the problem of memory performance.

GPGPU refers to the use of graphics processing units (GPUs) to perform general processing tasks, and it involves explicitly copying large amounts of data over the PCIe bus, between the CPU and GPU. This process can be relatively time consuming. Further complications and slowdowns can occur if the data does not fit on the GPU's memory, multiplying the problem by necessitating frequent CPU-GPU data transfers. Unlike a CPU, a GPU lacks the benefit of automatic memory paging. As a result, GPU memory management is explicitly handled by the programmer. This makes memory optimization a delicate process. The large number of threads running on a GPU will multiply any mistakes and inefficiencies in the code. This can seriously hamper an application's performance. Differences in algorithms and GPU architectures serve to complicate this matter even further. Data that arrives on a GPU's memory must then be accessed by a huge number of threads via the memory hierarchy. The threads need to be able to access this data efficiently, and through limited means. Inefficient GPU memory access is a common occurrence, and serves to hamper performance even further, particularly in workloads that are data intensive.

Optimizing GPU memory performance is a complex and challenging problem. There are a wide assortment of memory optimization techniques and configurations. The suitability and efficiency of these optimization techniques may vary between different applications and data sizes.

Our goal is to explore various techniques for improving memory performance and mitigating the latencies caused by data transfers and data accesses. Prior research studies have attempted to tackle this issue either by proposing automatic optimization solutions [1, 2, 3], manual techniques [4, 5], or by performing low-level analysis on simulated hardware [6, 7].

Our work revolves around manually implementing memory optimizations in CUDA, which is the framework used for most GPGPU applications [8], and running experiments on real hardware. It differs from a large portion of prior work by comparing many different memory optimization strategies, and being targeted at the Fermi GPU architecture.

In order to evaluate the effectiveness of our optimizations, we selected image processing as a case study. We chose image processing because it relies heavily on memory performance to fetch and process massive numbers of pixels simultaneously, and because it is a relatively diverse field with many parallels to other research fields.

First, we present an overview of the GPU memory performance problem, how it relates to image processing, and discuss techniques that have potentials to mitigate this problem. We then present a testing methodology to evaluate and analyze these techniques, which we apply to a wide variety of image processing algorithms. Finally, we attempt to categorize these algorithms based on their behavior.

The reminder of this paper is organized as follows. Section 2 explores related work. We provide some background information on GPGPU, GPU memory, and image processing in Section 3. We elaborate on our approach in Section 4, and describe our experimental methodology and metrics in Section 5. Our experimental results are presented discussed in Section 6. We conclude the paper in Section 7, and outline future work in Section 8.

## 2. Related Work

The variety of approaches that can be used to explore or improve GPU memory performance are very diverse. There has been a number of related work on this subject.

An analysis on the primary factors in implementing and evaluating image-processing algorithms was performed by Park et al. [5]. The authors proposed and evaluated a set of metrics aimed at helping programmers predict the characteristics of an algorithm's parallel implementation as well as its appropriateness. The metrics were tested on image processing algorithms from four distinct domains. The authors used NVIDIA G92 and G200 GPU hardware.

Qin et al. [4] presented an improved CUDA implementation of the differential evolution algorithm. They identified memory performance as the primary bottleneck and used a variety of optimization techniques such as streaming and kernel merging. The experimental results indicated that their implementation significantly outperformed prior serial and CUDA implementations, up until a certain problem size.

Ryoo et al. [10] presented a thorough evaluation of the NVIDIA G80 architecture's performance in CUDA. While G80 may be considered obsolete by today's standards, some of the optimization principles and strategies outlined in this paper also can be applied to current GPU architectures.

Moazeni et al. [5] proposed a memory optimization scheme based on graph coloring. Their goal was to maximize data reuse and reduce the pressure on global memory bandwidth by automatically managing the use of shared memory on the NVIDIA G80 architecture. The authors evaluated their memory optimization technique on an image processing benchmarking suite from the medical imaging domain.

Jablin et al. [1] presented an automatic system for managing and optimizing CPU-GPU communication called CGCM. Their aim was to construct a significantly simpler programming model, enabling programmers to focus on other aspects of their code without having to worry about memory optimization. Their experimental results indicated a performance boost in most cases but also a drop in performance in a few instances. They later improved on CGCM by adding a runtime system for dynamic data management [2].

Yang et al. [3] presented a GPGPU compiler for memory and parallelism optimization. Their compiler works by receiving a naïve GPU kernel, identifying memory access patterns, and outputting an optimized version that takes advantage of memory hierarchy and increased parallelism. Their experiments, which were performed on NVIDIA G80 and G200 GPUs, showed performance that was better or close to a highly tuned library.

A compiler framework to automatically translate OpenMP shared memory programs to CUDA and automatically optimize them was presented by Lee et al. [10]. Part of their optimization phase involves

exploiting the GPU's cache and memory hierarchy based on a predefined caching strategy table, however, the compiler still requires a programmer or automatic tuning system to guide the application of these optimizations.

Gupta et al. [6] presented an analysis of memory locality, using matrix multiplication as a case study. Their work explores memory access patterns and the way they interact with the GPU's memory hierarchy, for this particular application. The authors used GPGPU-Sim to simulate an NVIDIA GTX 480.

All of the work mentioned in this section is related to GPU memory performance, optimization, or digital image processing on the GPU. Our work examines GPU memory performance by manually implementing a series of well-known techniques on a series of image processing kernels. It is dissimilar to prior work in regards of the types of techniques and benchmarks that are examined. Some of these studies are centered on NVIDIA GPU architectures that predate the Fermi architecture, mainly because Fermi GPUs were not available at the time. This is relatively significant because the changes made in the Fermi generation included a relatively major overhaul of the memory system (see section 4.2).

## 3. Background

In this section, we provide some background information on the core topics that are relevant to our work.

### 3.1. *General-purpose computation on graphics processing units (GPGPU)*

GPUs can significantly accelerate many tasks, particularly tasks with a high degree of parallelizability. The superior computational power of a GPU allows performance that is beyond anything possible with current CPU technology. A typical GPU workload involves data being copied from the CPU memory to the GPU memory, the GPU performing computation on data using a large number of threads, and the result being copied back into the CPU memory.

In terms of hardware, a GPU contains a number of stream multiprocessors. Each multiprocessor is capable of executing a large number of threads, in parallel. Threads on a GPU are organized into blocks called thread blocks. Each thread block can be independently executed by a multiprocessor, and threads within a block can use shared memory to communicate and share data. A collection of thread blocks are organized into a grid, depending on the size of the data. This grid forms the basis for the GPU program's execution [11].

### 3.2. *GPU memory*

The GPU's memory is where most of the action in a GPGPU application takes place. GPUs rely heavily on their own memory systems because access to the CPU's memory is relatively time consuming. Recent advances in GPU architectures have resulted in significant increases to the number of GPU cores, memory size, and bandwidth. The high computational power and memory bandwidth of a GPU is offset by its limited memory capacity and relatively long memory access times. As a result, many GPU programs are limited by memory performance rather than computational power. Efficient utilization of a GPU's resources relies on the alleviation of these weaknesses. This necessitates the use of suitable data transfer and memory access methods. Fig. 1, provides an overview of the relationship between the CPU and GPU memory systems.

### 3.3. *Digital image processing*

Digital image processing refers to the use of computer algorithms to perform image processing on digital images. A digital image can be regarded as a matrix of colored dots called pixels. A typical serial image-processing algorithm operates by looping through these pixels and performing computation, accordingly. More often than not, each output pixel will rely on calculations performed on multiple input pixels. This has the potential to increase execution time, significantly.

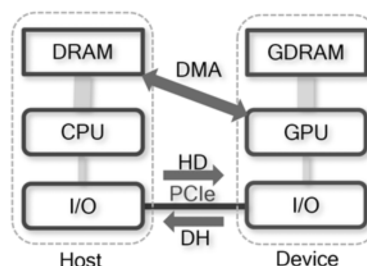


Fig. 1. CPU and GPU Memory Overview

Digital image processing is ideal for parallel programming because the workloads typically involve massive grids of pixels that can easily map to the large number of processing cores present on a GPU. The challenge lies in writing efficient GPU code that can handle transferring large digital images and accessing them from a massive number of threads.

Examples of image processing include edge detection, rotation, noise removal, and sharpening. Image processing has a wide variety of applications including television, medical imaging, tomography, robotics, and photography.

## 4. Our Approach

In this section, we explain our approach and elaborate on the various techniques that we use for our experiments. GPGPU memory performance can be divided into two distinct categories: CPU-GPU data transfers, and device memory access. The following subsections lay out our approach to exploring each of these memory performance categories.

### 4.1. *Exploring CPU-GPU data transfers*

In this section, we explain and explore different methods that can be used to speed up CPU-GPU data transfers, on the CUDA platform. CPU-GPU data transfers are used to feed the GPU cores with raw input data, and copy back the results.

*Pinned memory*

One of the data transfer optimization techniques is the use of pinned memory. Pinned memory is host that has been locked to a physical address, preventing it from being paged out. Pinned memory transfers are accelerated because the host memory can be directly accessed by the device (DMA). Using pinned memory yields a significant boost to performance but its applicability greatly depends on the target machine's available RAM. On the other hand, excessive use of pinned memory may lead to reducing the overall system's performance because it prevents part of the host's memory from being page out or used for anything else. Therefore, programmers must take great care when allocating pinned memory to ensure that the host system can handle the loss of RAM without adversely affecting system stability, or application performance. Pinned memory is allocated using the *cudaHostAlloc* function.

*Pinned memory microbenchmarks*

In order to get a better understanding of the performance benefits of pinned memory, we measure its performance impact on a single, isolated data transfer.

Table 1 depicts the average execution times for data transfers, using pinned and non-pinned memory.

Table 1. Pinned memory microbenchmarks

| **CPU-GPU Data Transfer** | **Average Execution Time (milliseconds)** | | | | | |
|---|---|---|---|---|---|---|
| | **640x480** | **1024x768** | **1920x1080** | **2560x1440** | **3735x3648** | **4896x4188** |
| **Non-Pinned Host Memory** | 0.36 | 0.80 | 2.09 | 3.60 | 12.88 | 19.40 |
| **Pinned Host Memory** | 0.23 | 0.52 | 1.31 | 2.29 | 8.49 | 12.72 |

The numbers 640x480, 1024x768, etc. indicate results obtained using different image resolutions. Both host to device and device to host data transfers were measured and found to be identical, so the results apply to data transfers in both directions. According to the results, pinning host memory conveys an average performance speedup of 55% for data transfers. This indicates a potential for significant performance gains for data-intensive GPU applications.

*Asynchronous data transfers*

Asynchronous data transfers can also be used to mitigate the data transfer latencies. In this technique, control is immediately returned to the CPU after being initiated. In CUDA, an asynchronous data transfer is performed using the *cudamemcpyasync* command. Performing data transfers asynchronously allows data transfers, host (CPU) code, and kernel execution to be overlapped. While overlapping host code and data transfers has always

been possible, overlapping data transfers with kernel execution requires at the very least a Fermi generation GPU. Asynchronous data transfers require that the host memory be pinned in order to function.

When used in conjunction with CUDA streams, this allows the programmer to perform concurrent data transfers by executing an additional *cudamemcpyasync* command on a different stream. It is important to remember that asynchronous data transfers on the same stream are executed in serial order.

*CUDA streams*

A stream is a sequence of operations that execute on GPUs, based on the order in which they are issued [11]. Using CUDA streams allows the programmer to express work dependencies by putting dependent operations in the same stream. Operations in different streams are independent and can be executed concurrently. By default, all CUDA operations are performed in the default stream (also known as Stream 0). Using of multiple streams can effectively reduce the communication overhead associated with transferring data between the CPU and GPU.

Fermi GPUs can simultaneously execute asynchronous data transfers while executing kernels. This depends on the number of copy engines that the GPU has. Some GPUs have two copy engines and can simultaneously perform one asynchronous data transfer from the host to the device, an additional asynchronous data transfer from the device to the host, and execute kernels [12]. Additional data transfers in the same direction are serialized. Fig. 2, shows how an example of CUDA streams can be used to improve performance.

*CUDA stream microbenchmarks*

We perform a set of microbenchmarks in order to discover how CUDA streams affect an application's performance. To do so, we measure a data transfer and kernel execution separately, then both (on the same stream), and finally, both on separate streams. We use the Sharpen Kernel for this microbenchmark. As with the pinned memory microbenchmarks, we measure the execution time on multiple image resolutions. Table 2 depicts the results these microbenchmarks.

Table 2. CUDA stream microbenchmarking results

| **Operation** | **Average Execution Time (milliseconds)** | | | | | |
|---|---|---|---|---|---|---|
| | **640 x480** | **1024 x768** | **1920 x1080** | **2560 x1440** | **3735 x3648** | **4896 x4188** |
| **Data Transfer** | 0.2 | 0.5 | 1.3 | 2.3 | 8.6 | 12.8 |
| **Kernel Execution** | 0.1 | 0.3 | 0.8 | 1.3 | 5.0 | 6.9 |
| **Both (Non-Concurrent)** | 0.3 | 0.8 | 2.1 | 3.6 | 13.7 | 19.8 |
| **Both Using CUDA Streams (Concurrent)** | 0.2 | 0.5 | 1.3 | 2.3 | 8.6 | 12.8 |

The results indicate that the execution time of two concurrent streams is approximately identical to the stream with the longest execution time. Therefore, it can be concluded that the more balanced the streams are, the greater the potential speedup.

### 4.2. *Exploring device memory hierarchy*

Optimizations that target device memory aim to take advantage of the GPU's memory hierarchy in order to improve memory accesses performed by threads. The results of these optimizations are highly dependent on the GPU kernel's memory access pattern and access frequency.

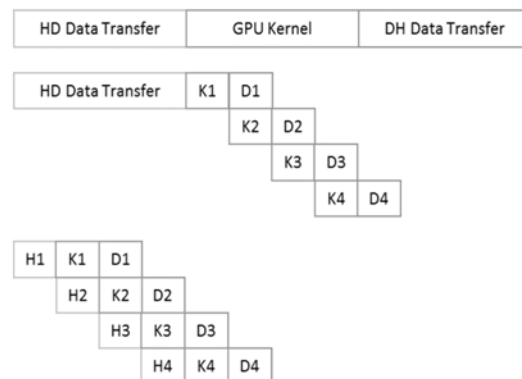


Fig. 2. Concurrent kernel and data transfers using CUDA streams

The Fermi GPU architecture provides multiple memory spaces that can be accessed by CUDA threads [11 and 13]. Each of these memory spaces are optimized for different memory usages and have varying sizes, caching schemes, access latencies, read/write characteristics and performance implications. Table 3 provides an overview of these memory spaces.

*Global memory*

Global memory is device memory that is accessible by all threads. In contrast with previous GPU architectures, global memory transactions are cached on the Fermi architecture. As a result, Fermi's global memory performance is significantly faster compared to previous generations. However, its performance still depends on the program's memory access pattern. Uniform access patterns yield the best performance but may be challenging to achieve. Local memory is private global memory that has been assigned to a particular thread. Global and local memory serves as the baseline for our device memory experiments because they are the simplest and most commonly used memory spaces.

*Shared memory*

Shared memory is a dedicated, high performance on-chip memory that is shared across a thread block. Shared memory is explicitly declared and used inside a GPU kernel. We apply shared memory to the image processing kernels by modifying them to copy the input image to shared memory and access it from there. The shared memory array is padded to avoid bank conflicts.

*Constant memory*

Constant memory is a read-only memory spaces that resides on the device and is accessible by all threads. Compared to the other memory spaces mentioned here, constant memory's capacity is extremely limited (64KB in our case). As a result, we cannot utilize constant memory as a means to access the input images in our experiments. However, constant memory may still be useful for kernels that contain require a small amount of data to be read by all threads. This is the case in one of our experiments, which uses a small read-only array in part of its calculations.

Table 3. GPU memory hierarchy overview [11]

| Memory | Location on/off chip | Cached | Access | Scope | Lifetime |
|---|---|---|---|---|---|
| **Global** | Off | Yes | R/W | All threads + host | Host Allocation |
| **Local** | Off | Yes | R/W | 1 thread | Thread |
| **Shared** | On | n/a | R/W | All threads in block | Block |
| **Texture** | Off | Yes | R | All threads + host | Host Allocation |
| **Constant** | Off | Yes | R | All threads + host | Host Allocation |

*Texture memory*

Texture memory is similar to global memory in that it is accessible by all threads. Its main difference is that it is read-only, and is cached in the texture cache. Texture cache is optimized for 2D spatial locality, which should in theory help with accessing 2D images. We test texture memory by replacing all reads from global memory to texture memory reads.

## 5. Experimental Methodology and Metrics

In this section, we discuss our setup, selected benchmarks, testing methodology, and measurement metrics.

### 5.1. *Experimental setup*

All experiments are performed on a system equipped with an Intel 2500k processor running at its default frequency of 3.2GHz, 4GBs of 1333MHz DDR3 RAM, and an NVIDIA GTX 570 video card with 480 CUDA cores running at 1.5GHz, and 1280MBs of GDDR5 RAM. Our setup uses version 5.0 of the NVIDIA CUDA Parallel Computing Platform to compile and run the experiments.

### 5.2. *Selected benchmarks*

The benchmarks selected for our experiments, are a series of well-known image processing kernels that we have implemented in CUDA and C++. The CPU and GPU implementations are completely identical in function, and constitute what we consider a fair comparison. The device memory experiments use separate versions of each kernel to take advantage of the GPU memory spaces. Excluding the parts of the code that are related to memory access, the remainder of the kernel remains identical between those versions. The benchmarks differ in terms of memory usage, memory access patterns, computational intensity, and data reuse. We provide the parameters used for each benchmark in Table 4. The benchmarks and their descriptions are as follows

- Sharpening Filter: sharpening filter enhances an image by making details stand out. One commonly used implementation method sharpens an image by first applying a blur filter to each pixel, and then subtracting the resulting values from the original image. A strength factor controls the intensity of the effect.
- Bloom Effect: The Bloom effect creates the illusion of light sources glowing and bleeding light into the surrounding area. This effect frequently occurs when using real-world cameras. We reproduce Bloom by applying multiple convolution filters.
- Median Filter: Median Filters are often used to eliminate noise from an image, or are sometimes a building block in other, larger image processing algorithms. Each pixel is replaced by the median of its neighboring pixels.
- Ordered Dithering: Dithering is commonly used to eliminate color banding and for color reduction. Ordered dithering uses a threshold matrix of an arbitrary size called a Bayer matrix to determine the pixel's final color. We use an 8x8 Bayer matrix for our benchmarks.
- Morphological Gradient: Morphological gradient is typically used for edge detection and image segmentation. It performs morphological erosion and dilation for each pixel and then calculates the difference.
- Oil Painting: The oil painting algorithm receives an image and outputs a rendition that looks like an oil painting. A common implementation of this algorithm counts the colors in a radius surrounding each pixel. It then finds the maximum repeated color and uses writes that to the output image.

Table 4. GPU kernel parameters

| Kernel Name | Parameters |
|---|---|
| Sharpen Filter | Radius = 1, Strength = 2.3 |
| Bloom Effect | Radius1 = 1, Radius2 = 3 |
| Median Filter | Radius = 1 |
| Ordered Dithering | BayerSize = 8 |
| Morphological Gradient | Radius = 2 |
| Oil Painting | Nbins = 20, Radius = 2 |

### 5.3. *Input data*

We use a series of bitmap images with a diverse array of resolutions, for our experiments. Table 5 presents the specifications of these images.

Table 5. Input data specifications

| Row | Image Resolution | Pixels | Size (KB) |
|---|---|---|---|
| 1 | 640 x 480 | 307200 | 901 |
| 2 | 1024 x 768 | 786432 | 2305 |
| 3 | 1920 x 1080 | 2073600 | 6076 |
| 4 | 2560 x 1440 | 3686400 | 10801 |
| 5 | 3735 x 3648 | 13625280 | 39926 |
| 6 | 4896 x 4188 | 20504448 | 60072 |
| 7 | 6587 x 8336 | 54909232 | 160892 |

### 5.4. *CPU-GPU data transfer experiments*

We explore the effects of CPU-GPU data transfer optimizations in order to identify the best data transfer strategy. These optimizations include asynchronous data transfers, different streaming configurations, and memory pinning. The benchmarks are performed on a series of images of varying sizes. Several metrics exist

that can be used to evaluate data transfers (such as memory bandwidth); however, we focus on the effects that these data transfers have on the application's execution time. As such, we measure the total execution time of the parallel GPU program, which includes host to device and device to host data transfers, and the GPU kernel execution times. We then calculate the relative speedup over naïve ($Speedup_n$) as the ratio of naïve parallel execution time ($T_n$) to optimized parallel execution time ($T_p$). The speedup is defined in Eq (1).

$$Speedup_n = \frac{T_n}{T_p} \quad (1)$$

We start with the slowest configuration (naïve) which uses page-able (non-pinned) host memory and blocking, synchronous data transfers. We then accelerate the synchronous data transfers by pinning the host memory. Next, all data transfers become asynchronous transfers and we experiment with different streaming configurations. We split the work between two streams in a variety of different configurations for Host-to-Device (HD) and Device-to-Host (DH) data transfers. Finally, we assign a separate stream for each data transfer and kernel launch. The final speedups are calculated using the average from seven different test images specified in table 5. Each configuration is tested 100 times and the results are averaged. Performance gains are limited by the ratio of GPU computation to data transfers. The higher the ratio, the less we can expect to improve performance by optimizing CPU-GPU data transfers.

### 5.5. *Device memory experiments*

The general idea behind these experiments is to evaluate the performance benefits of utilizing the GPU's memory hierarchy. To do so, we compare the performance of each benchmark, modified to take advantage of either global memory, shared memory, constant memory, texture memory, or a combination of shared and texture memory, to access the device memory. Each configuration is then measured relative to its naïve implementation, which uses global memory, and its serial, CPU equivalent. In order to obtain a clear perspective on the fundamental differences between these memory spaces, we avoid performing miscellaneous optimizations, such as altering the overall access pattern, or optimizing the computational sections of the code. We use constant memory in only one of our explored algorithms, due to its limited applicability.

Device memory optimizations only affect kernel execution time; therefore, we exclusively measure kernel execution time, excluding any overheads caused by CPU-GPU data transfers. This allows us to focus our attention on the device memory performance.

To get a clearer picture of the program's performance, we compute the relative speedups over two baselines: Naïve and Serial. Speedup over naïve is calculated as defined in (1). Speedup over serial (Speedups) is defined based on the ratio of serial execution time ($T_s$) to parallel execution time ($T_p$), as shown in Eq. (2).

$$Speedup_s = \frac{T_s}{T_p} \quad (2)$$

As with the CPU-GPU data transfer experiments, each configuration is tested 100 times on seven input image resolutions, and the results are averaged.

## 6. Experimental Results and Discussion

In this section, we present the results of our experiments, discuss their significance, and speculate on the factors that may have had an impact on the results.

### 6.1. *CPU-GPU data transfer results*

In this subsection, we evaluate the data transfer techniques discussed in section 4.2. The experiments revolve around the use of pinned memory and CUDA streams in a wide variety of configurations. The CPU-GPU data transfer results revolve around three primary operations: The GPU kernel, the host to device data transfer, and the device to host data transfer. The sum of these operations constitutes the total execution time, which is used to calculate the relative speedup compared to the baseline, naïve configuration. All operations in a stream use pinned memory and asynchronous data transfers. The Naïve configuration refers to the use of synchronous data transfers on non-pinned host memory (the most basic configuration possible). Table 6 displays the results of our data transfer experiments.

Table 6. CPU-GPU data transfer results

| Configuration | | | | Average Speedup$_n$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ***Row*** | ***Host to Device*** | ***GPU Kernel*** | ***Device to Host*** | ***Oil Painting*** | ***Sharpen Filter*** | ***Morph. Gradient*** | ***Ordered Dithering*** | ***Bloom Effect*** | ***Median Filter*** |
| 1 | Non-Pinned | Stream0 | Non-Pinned | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 2 | Pinned | Stream0 | Pinned | 1.02 | 1.41 | 1.13 | 1.17 | 1.23 | 1.22 |
| 3 | Stream1 | Stream1 | Stream1 | 1.02 | 1.42 | 1.14 | 1.17 | 1.24 | 1.22 |
| 4 | Stream1 | Stream2 | Stream2 | 1.04 | 1.62 | 1.29 | 1.38 | 1.55 | 1.52 |
| 5 | Stream1 | Stream2 | Non-Pinned | 1.03 | 1.42 | 1.20 | 1.24 | 1.35 | 1.33 |
| 6 | Stream1 | Stream1 | Stream2 | 1.04 | 1.63 | 1.28 | 1.36 | 1.55 | 1.50 |
| 7 | Non-Pinned | Stream1 | Stream2 | 1.03 | 1.43 | 1.20 | 1.25 | 1.37 | 1.35 |
| 8 | Stream1 | Stream2 | Stream1 | 1.06 | 1.82 | 1.50 | 1.66 | 2.11 | 2.00 |
| 9 | Stream1 | Stream2 | Stream3 | 1.06 | 1.82 | 1.50 | 1.66 | 2.11 | 1.99 |

The first column, labeled "Configuration", shows the state of the Host to Device data transfer, GPU Kernel, and Device to Host data transfer. The data transfers can have three different states: 1) Non-pinned memory 2) Pinned memory 3) CUDA Stream. In the case of CUDA Streams, the number denotes the stream number to which the data transfer has been assigned. GPU Kernels can be either explicitly assigned to a CUDA stream, or left to execute on the default stream (Stream0 is always synchronous). The second column shows the average speedup over the naïve configuration (shown in the first row) for each benchmark, across all of the tested configurations.

As shown in Table 6, the Median Filter results indicate that a 20% performance boost can be achieved by switching to pinned host memory. Beyond that, using two CUDA streams to perform kernel execution and only one of the data transfers nets another 30% performance boost. The difference between optimizing the host to device data transfer or the device to host data transfer appears to be negligible. The next configuration (row 8), optimizes both host to device and device to host data transfers, resulting in a total speed up of approximately 2x compared to the naïve configuration. The final configuration attempts to utilize three discrete CUDA Streams, which due to the limitations of our hardware (one copy engine), does not provide any additional performance.

The results from our Bloom benchmarks are extremely similar to our Median Filter results, including the same max speedup of x2. This indicates that they share a similar kernel execution to data transfer balance ratio.

The Morphological Gradient results follow a similar trend, but indicate slightly lower performance gains. This can be attributed to a higher amount of work being done by the GPU kernel, such as fetching 49 pixels for each processed pixel, compared to nine pixels for the Median Filter. As a result, our final speedup is limited to 50%.

In a similar fashion, Ordered Dithering gains an average of 17% from using pinned memory, 36% from placing the kernel and one of the data transfers in CUDA Streams, and a total average of 66% from optimizing both data transfers.

The Oil Painting benchmark displayed the lowest speedup out of all of our benchmarks. This can be attributed to the fact that the Oil Painting algorithm is the most computationally intensive algorithm tested here. In this case, the amount of time spent executing the kernel far outweighs the time spent transferring the input/output images. As a result, we gain very little from optimizing the data transfers.

The Sharpening Filter results show a maximum speedup of 82%, which indicates a nice balance between the kernel and the data transfers.

One of our goals is to categorize these benchmarks according to the results. This categorization can then act as a guide for optimizing similar applications. Fig. 3, depicts the maximum speedups obtained for each benchmark. We use this to sort the benchmarks into three categories: poorly balanced, moderately balanced, and well balanced. Oil painting's computation heavy kernel makes it a poorly balanced kernel, because data transfers account for only a small fraction of its workload. Morphological gradient and Ordered Dithering are moderately balanced kernels, because they exhibit medium performance gains, but less than what we expected. Median filter, sharpen filter, and bloom effect displayed the highest performance gains, and can be considered well-balanced kernels.

In conclusion, we can say that, when optimizing CPU-GPU data transfers, it is important to balance out data transfers with kernel execution, as neither operation should be waiting for the other to finish.

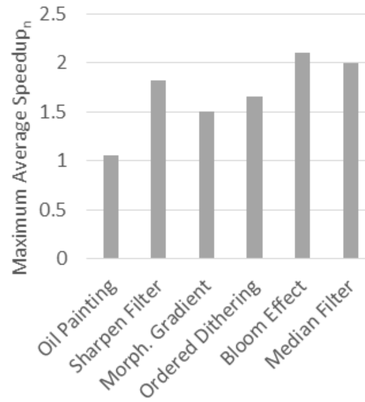


Fig. 3. Maximum average speedups for CPU-GPU data transfer benchmarks

### 6.2. *Device memory experimental results*

In this section, we present and discuss the results of our device memory experiments. Our device memory experiments deal with data that has already been copied to the GPU and needs to be accessed by all the threads running a GPU kernel.

We use the following abridged row labels for our result tables

- Naïve: Refers to a basic implementation using global memory, and no other optimizations
- Texture: Texture Memory
- Shared: Shared Memory
- Texture + Shared: Refers to the use of texture memory to fill shared memory, which is then used as the primary means of accessing the data.
- Constant: Constant Memory

Table 7 depicts the results of our experiments relative to the naïve configuration, and Table 8 shows the results of the same experiments relative to the serial implementation.

Table 7. Device memory benchmarks – speedup over naïve

| | Average Speedup$_n$ | | | | |
|---|---|---|---|---|---|
| **Memory Configuration** | ***Oil Painting*** | ***Sharpen Filter*** | ***Morph. Gradient*** | ***Ordered Dithering*** | ***Bloom Effect*** |
| **Naïve** | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| **Shared** | 1.10 | 1.11 | 0.91 | 13.35 | 0.52 |
| **Texture** | 1.08 | 1.46 | 1.00 | 21.21 | 0.90 |
| **Texture + Shared** | 1.15 | 1.04 | 0.94 | 14.11 | 0.51 |
| **Constant** | N/A | N/A | N/A | 18.37 | N/A |

Table 8. Device memory benchmarks – speedup over serial

| | Average Speedup$_s$ | | | | |
|---|---|---|---|---|---|
| **Memory Configuration** | ***Oil Painting*** | ***Sharpen Filter*** | ***Morph. Gradient*** | ***Ordered Dithering*** | ***Bloom Effect*** |
| **Naïve** | 14.34 | 100.07 | 107.21 | 2.30 | 146.40 |
| **Shared** | 15.98 | 111.60 | 97.23 | 30.31 | 76.59 |
| **Texture** | 15.77 | 146.08 | 107.68 | 47.74 | 131.58 |
| **Texture + Shared** | 17.04 | 103.61 | 100.65 | 32.03 | 74.15 |
| **Constant** | N/A | N/A | N/A | 41.43 | N/A |

The device memory experiments display widely varying reactions to our optimizations. This is a result of the different patterns of memory access, and data re-use, in our benchmarks. A discussion of these results follows.

Ordered Dithering receives a relatively massive performance boost from our optimizations. The reason behind this is due to the Bayer matrix used for the calculations. In the naïve configuration, the Bayer matrix is defined inside the kernel. As a result, each thread will have a separate copy of it. This places undue pressure on the GPU's memory and cache hierarchy, because the Bayer matrix is read by all threads and never modified.

The highest speedup occurs when utilizing texture memory, followed closely by global and constant memory. This can be attributed to the fact that texture memory is optimized for 2D locality. Constant memory, despite appearing to be a perfect fit for this kind of algorithm, does not manage to dethrone texture memory.

The Sharpening filter benchmarks receive a minor boost when using shared memory, and a much larger speedup when utilizing texture memory to access their required input data. This is due to texture memory's innate advantage when working with 2D arrays (such as images), coupled with its relatively low degree of data reuse and computational intensity (compared to the other algorithms we examined).

The Bloom benchmarks performed best in their naïve configurations. This is something that we would almost never see with pre-Fermi GPU architectures. As mentioned earlier, kernels that re-use data in a uniform manner can benefit the most from the global memory cache. Given the use of subsequent convolutions in the bloom implementation, there is a considerable amount of data overlap. It is worth noting here that texture memory is not too far behind and is only 10% slower. Shared memory does not cope well with the huge amount of data accesses.

For the Morphological Gradient benchmarks, performance is very similar between the various memory configurations. This indicates that we are compute limited rather than memory limited, and thus should focus on optimizing the computational portion of the kernel.

The Oil Painting benchmarks display a preference for a combination of shared and texture memory (albeit a relatively miniscule one). Being our most memory intensive benchmark, combining different memory spaces allows the algorithm to cope with the large amount of data (relative to the other algorithms) that it needs to access. Speedup over serial (Table 8) is significantly lower than the other algorithms, even with the applied optimizations, and uncharacteristically slow for a GPU implementation. This indicates that the kernel has other unresolved inefficiencies, which cannot be solved by merely modifying the device memory accesses.

Table 9 recommends the best memory space for each benchmark, based on our observations, and states their defining characteristic. This can guide programmers towards choosing the best memory space when implementing these algorithms, or other algorithms with similar characteristics.

Table 9. Categorization of the device memory benchmarks

| Benchmark | Defining Characteristic | Recommended Memory Space |
|---|---|---|
| Oil Painting | Computationally intense | Texture + Shared |
| Sharpen Filter | 2D Spatial locality | Texture |
| Morph. Gradient | 2D Spatial locality | Global or Texture |
| Ordered Dithering | Non-uniform data access | Texture |
| Bloom Effect | Data reuse | Global |

## 7. Conclusion

In this paper, we have described the challenges of implementing efficient GPGPU applications. We have discussed memory performance and the ways in which it can become a major bottleneck. We have outlined and explained various memory optimization techniques that could be used to solve this problem. The optimizations have been evaluated on a series of image processing applications. Finally, we have categorized each set of benchmarks according to their behavior.

Our results have been divided into two groups: CPU-GPU data transfer optimizations, and device memory optimizations.

For the CPU-GPU data transfer results, we can conclude that the use of pinned memory and CUDA streams will generally net significant performance gains, as long as the relative proportion of the GPU kernel and the data transfers is not skewed in the favor of either one of them. We have sorted the benchmarks into three categories, based on their observed behavior: poorly balanced, moderately balanced, and well balanced, based on their maximum speedup. We observed a speedup of up to x2.1 compared to the naïve implementation.

For the device memory results, we can conclude that texture memory provides the overall best performance. It is the fastest configuration in two of the image processing experiments, ties for best performance in one of them, and trails in the remaining two experiments by a narrow margin.

We have categorized the device memory benchmarks based on the memory space that obtained the best performance, and their defining characteristic. We observed a speedup of up to x21 compared to the naïve GPU implementation and up to x146 compared to the serial implementation.

Our benchmark results and categorizations could be used to assist developers in choosing better configurations for their GPU applications.

## 8. Future Work

Our future work will consist of exploring GPU memory optimizations using other programming platforms, such as OpenCL. We plan to expand our work to include other applications from the image-processing domain, and possibly other domains, in order to obtain a more complete categorization. The experiments could also be performed again, on other GPU architectures. Work towards an improved compiler or automatic optimizer that can intelligently detect and apply the most suitable data transfer and memory access techniques is another potential avenue for research.

## References

[1] T. B. Jablin, P. Prabhu, J. A. Jablin, N. Johnson, S. Beard and D. I. August, “Automatic CPU-GPU Communication Management and Optimization”, Programming Language Design and Implementation (PLDI), June 2011.
[2] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu and D. I. August, “Dynamically Managed Data for CPU-GPU Architectures”, ACM Code Generation and Optimization (CGO), 2012.
[3] Y. Yang, P. Xiang, J. Kong, and H. Zhou. "A GPGPU Compiler For Mmemory Optimization and Parallelism Management." In ACM Sigplan Notices, vol. 45, no. 6, pp. 86-97. ACM, 2010.
[4] A. K. Qin, F. Raimondo, F. Forbes and Y. S. Ong, “An Improved CUDA-Based Implementation of Differential Evolution on GPU”, Proceeding of the fourteenth international conference on Genetic and evolutionary computation conference (GECCO), pp. 991-998. ACM, 2012.
[5] M. Moazeni, A. Bui, and M. Sarrafzadeh, “A Memory Optimization Technique for Software-managed Scratchpad Memory in GPUs”, IEEE Symposium on Application Specific Processors (SASP), July 2009.
[6] S. Gupta, P. Xiang, and H. Zhou. "Analyzing locality of memory references in GPU architectures." (2013).
[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. “Analyzing CUDA workloads using a detailed GPU simulator”. In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pages 163 –174, 2009.
[8] http://hgpu.org/?page_id=3529
[9] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim, "Design and Performance Evaluation of Image Processing Algorithms on GPUs", IEEE Transactions on Parallel and Distributed Systems, pp. 91-104, 2011.
[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA”. ACM SIG PLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 73–82, 2008.
[11] NVIDIA CUDA C Best Practices Guide from CUDA Toolkit 5.0.
[12] NVIDA. Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Fermi, 2009.
[13] NVIDIA CUDA Programming Guide, Version 5, NVIDIA Corporation, Santa Clara, CA, 2012
[14] NVIDIA Corporation. Tuning CUDA Applications for Fermi, Version 1.5, 2011.
[15] CUDA C/C++ Streams and Concurrency Webinar, NVIDIA Corporation. 2012.
[16] J. Stam, "Maximizing GPU Efficiency in Extreme Throughput Applications." Proceedings of the GPU Technology Conference 2009. 2009.
[17] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W. W. Hwu. "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors." In ACM SIGPLAN Notices, vol. 47, no. 8, pp. 23-34. ACM, 2012.
[18] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, “A Memory Model for Scientific Algorithms on Graphics Processors,” in SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM, 2006, p. 89.