Accelerating MATLAB Image Processing Toolbox Functions on GPUs

Jingfei Kong, Martin Dimitrov, Yi Yang#, Janaka Liyanage
Lin Cao, Jacob Staples, Mike Mantor*, Huiyang Zhou#

School of Computer Science, University of Central Florida, {jfkong,dimitrov,janaka,lcao,jstaples}@cs.ucf.edu
#Dept of Electrical and Computer Engineering, North Carolina State University, {yyang14,hzhou}@ncsu.edu
*Graphics Products Group, AMD, Michael.Mantor@amd.com

Abstract

In this paper, we present our effort in developing an open-source GPU (graphics processing units) code library for the MATLAB Image Processing Toolbox (IPT). We ported a dozen of representative functions from IPT and based on their inherent characteristics, we grouped these functions into four categories: data independent, data sharing, algorithm dependent and data dependent. For each category, we present a detailed case study, which reveals interesting insights on how to efficiently optimize the code for GPUs and highlight performance-critical hardware features, some of which have not been well explored in existing literature. Our results show drastic speedups for the functions in the data-independent or data-sharing category by leveraging hardware support judiciously; and moderate speedups for those in the algorithm-dependent category by careful algorithm selection and parallelization. For the functions in the last category, fine-grain synchronization and data-dependency requirements are the main obstacles to an efficient implementation on GPUs.

Categories and Subject Descriptors
D.1.3 [Software]: Programming Techniques—- *Concurrent Programming*

General Terms
Algorithms, Performance, Design

Keywords
GPGPU, Image Processing, CUDA, OpenCL, MATLAB

1. Introduction

With high volumes of graphics cards deployed in personal computers or workstations, graphics processor units (GPUs) become a parallel computing platform accessible to a wide range of users. Given their high memory bandwidth and teraflops computing capability, there is a strong incentive to use GPUs for general purpose computation (GPGPU) and there have been many successful reports in the literature on such efforts [9]. However, two main challenges remain for wide adoption of GPGPU. First, although GPU programming has been significantly simplified with the development of programming tools/environments such as AMD/ATI Brook+/CAL [16], NVIDIA CUDA [22], and a recently proposed open industry standard OpenCL [23], developing high-performance GPU programs requires deep understanding of both application algorithms and GPU hardware architectures, making it elusive for many end users. Second, existing work on GPGPU usually focuses on one or a few specific algorithms and lacks of a systematic way of dealing with more generic classes of applications.

In this work, we promote wide adoption of GPGPU by leveraging MATLAB™, one of the most commonly used platforms in scientific computing, and developing open-source GPU programs to replace the native implementation of MATLAB functions through the MEX interface [20]. This way, end users can easily experience the benefits of GPGPU with very little change in their MATLAB code. There have been some recent works on using GPUs to accelerate programs in MATLAB [21] [19]. Jacket [18] is a commercial product which is closely related to our work. It is a GPU engine for MATLAB and enables standard MATLAB code to run on NVIDIA CUDA-supported GPUs. In this work, we focus on the MATLAB Image Processing Toolbox (IPT), which in majority has not been covered by previous work. We selected a dozen of representative and performance-critical functions from IPT. Despite apparent resemblance between graphics and image processing in that they all work on a large number of pixels, many of the IPT functions exhibit significantly different characteristics from typical graphics processing. Based on the inherent parallelism and data dependency, we grouped these functions into four categories: data independent, data sharing, algorithm dependent, and data dependent. In each category, we present a detailed case study to reveal interesting insights in optimizing GPU code for algorithms in the category. In our work, we developed our GPU programs for both NVIDIA GTX 280 and AMD/ATI RV870 GPUs in order to exploit their unique hardware features for these IPT functions. Some of the performance-critical hardware features such as texture caches, to our knowledge, have not been well explored in the existing work.

Our experience with accelerating the *data-independent* IPT functions shows that they can easily achieve very good kernel speedups using GPUs. Some functions in this category, however, are limited by data bandwidth due to their low computational requirements per data element. Functions in the *data-sharing* category require the programmer to carefully balance thread-level parallelism and data/computation reuse, in order to overcome high latencies in accessing off-chip GPU memories. Due to their abundant parallelism these functions can also achieve drastic kernel speedups by effectively exploiting the GPU hardware features. Algorithm selection and the associated parallelization strategy are critical for *algorithm-dependent* function, in order to extract parallelism for GPUs. Nevertheless, we are able to achieve reasonably good kernel speedups from the functions in this category even with their inherent data dependencies and communication requirements. Finally, fine grain synchronization and data dependency requirements make the functions in the *data-dependent* category a relatively poor fit for GPU implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GPGPU'10, March 14, 2010, Pittsburg, PA, USA.
Copyright© 2010 ACM 978-1-60558-935-0/10/03 $10.00.

Table 1. Performance-critical GPU hardware features and their implications on GPGPU programs

Performance-Critical Hardware Features	Implication on GPGPU Programs	
	AMD/ATI HD5870 (RV870)	**NVIDIA GTX280**
Memory Access Bandwidth	Vector-type (float2 or float4) data access	Scalar-type (float) or vector-type (float2) data access
Register File	A high number of registers (1k float4 registers or 16kB) per core implies more computational work in each core	A relatively small number of registers (2k float registers or 8kB) per core implies less computational work in each core.
Shared Memory/Local Data Share	Higher amount of shared memory (32kB per SM) implies large tile sizes (one tile is the workload of one thread block or work group)	Smaller amount of shared memory (16kB per SM) implies small tile sizes (one tile is the workload of one thread block or work group)
Ratio of (Peak Computation Throughput / Peak Memory Bandwidth)	(2.72 TFLOPS)/(154 GB/s) means more computation needs to be performed for each loaded data item	(0.62 TFLOPS)/(141 GB/s) means a relatively small amount of computation needs to be performed for each loaded data item

The main contributions of this work include (1) Develop high-quality open-source library code to promote wide adoption of GPU as an effective accelerator for media processing; (2) Reveal interesting insights on how to efficiently parallelize a wide range of image processing algorithms; and (3) Identify performance-critical hardware features in different GPUs.

The remainder of the paper is organized as follows. In Section 2, we first highlight the impact of different GPU architectures on GPGPU programming and establish the experimental methodology to be used throughout the paper. In Section 3, we select representative and performance-critical functions from the MATLAB IPT and group them in four categories based on their characteristics. Then, we present a detailed case study and summarize the performance results as well as optimization strategy for each of the four categories in Sections 4-7, respectively. Finally, Section 8 concludes the paper.

2. Experimental Methodology

2.1. The Impact of Different GPU Architectures on GPGPU Programming

The key performance-critical features of the latest AMD/ATI GPU and NVIDIA GPUs are summarized in Table 1. As shown in the table, different GPU architectures share some common requirements but also have their unique ones for GPGPU programs to achieve high performance. First, different GPUs favor different data types of memory accesses for high memory bandwidth. In an experiment of copying 128MB data within the GPU memory, when we used float, float2, float4 data types (all being coalesced accesses), the sustained bandwidth in HD5870 (RV870) was 71GB/s, 98GB/s, and 101GB/s, respectively (the kernel is written in OpenCL and compiled with ATI Stream SDK 2.0). For GTX 280, in comparison, the bandwidth achieves 57 GB/s, 120 GB/s, and 98 GB/s for float, float2, float4 data types, respectively (the same OpenCL kernel compiled with NVIDIA GPU Computing SDK 2.3b). Interestingly, when the same copy kernel is written in CUDA and compiled with CUDA, the GTX 280 achieves 96 GB/s, 98 GB/s, and 77GB/s for the three data types, which suggests that either the drivers need to be optimized or the programming models introduce different performance overheads. Nevertheless, float2/float4 is the desired data type for HD5870 while the float2 data type works well on GTX280 with both OpenCL and CUDA. The best scalar support is from GTX280 using CUDA.

Second, hardware features including the register file size, shared memory size and the ratio of the peak computation throughput over the peak memory bandwidth, suggest that more computational workloads need be allocated for AMD GPUs than NVIDIA GPUs, at both the thread and thread-block/work-group levels.

Another interesting difference, which has significant performance impact on some IPT functions, is the texture caches, whose sizes are not disclosed in either GPU. Our experiments indicate a possibility that RV870 has a larger level-one (L1) texture cache than GTX 280 (see Section 5). More detailed discussion on either GPU architecture can be found in [16] and [22].

2.2. Experimental Setup

We use both NVIDIA GTX 280 and AMD/ATI RV870 GPUs in our experiments to exploit their unique hardware features. In our experiments, we use a NVIDIA GTX 280 card with NVIDIA GPU Computing SDK. Our AMD/ATI machine uses a Radeon 5870 card (RV870) with ATI Stream Computing SDK 2.0. For CPU comparison, we use an AMD Dual-Core Opteron processor running at 2.2 GHz and MATLAB 2008a. All machines use Windows XP 32-bit Operating Systems.

2.3. Speedup Computation

In typical GPGPU setups, a GPU is used as an accelerator to offload computation from a CPU. As a result, in order to perform computation on a GPU, we must first initialize and copy the problem input data from the CPU to the GPU. The data is transferred over the PCI Express® (PCIe) bus. After the GPU computation, we also need to transfer the results back to the CPU. Even though the data transfer over the PCIe bus is normally fast (peak theoretical transfer rate for PCIe 1.1 X 16 in our experimental setup is 4GB/s in both directions), it may still be a bottleneck for applications with low computational requirements per data element. In addition, accelerating MATLAB functions involves an overhead of going through the MATLAB interpreter and the MEX interface [20] and possibly casting the input/output data to a different format before using them on the GPU. Therefore for functions in case studies, we report the GPU performance results in two formats: overall speedup and kernel speedup over the CPU. Overall speedup includes both the input/output data transfer overhead between CPU and GPU and the overhead of going through the MEX interface. The kernel speedup only includes the GPU kernel execution time. Note, that for many applications it is desirable to perform multiple transformations of the input data (i.e., to run multiple kernels) before outputting the results back to the CPU. In this case the cost of data transfer is amortized and the kernel execution time is a more relevant metric.

Table 2. Categories of selected IPT functions for GPU implementation - Kernel Speedups in X

Function Category	Function Name	Function Description	Kernel Speedup on GTX 280		Kernel Speedup on HD5870
			CUDA	OpenCL	OpenCL
(A) Data independent	intlut	Convert integer values using lookup table	17.7	17.5	12.7
	imadjust	Adjust image intensity values	21.4	15.7	11.9
	imlincomb	Linear combination of images	944.6	593.7	1385.4
(B) Data sharing	edge	Find edges in grayscale image	3385.9	1175.2	4955.1
	imregionalmax	Regional maxima of an image	2117.8	798.4	3694.0
	ordfilt2	2-D order-statistic filtering	1199.6	171.6	1727.1
	conv2	2D convolution of an image	345.5	156.9	649.8
	mean2	Average of matrix elements	50.5	25.2	34.7
	imdilate/imerode	Dilate/erode a grayscale image	951.5	523.3	1579.8
(C) Algorithm dependent	bwdist	Euclidean distance transform of a binary image	134.8	126.2	104.3
	radon	Radon transform	84.3	67.4	61.2
(D) Data dependent	dither	Represent grayscale images in binary format	10.2	6.5	7.6

2048 x 2048 images of random values are used by default. 1024 x 1024 is used for bwdist
512 x 512 is used for radon with 320 angles and a 7 x 7 filter is used in conv2.

In all our experiments, the CPU time is obtained using the corresponding functions from the MATLAB Image Processing Toolbox on CPU. The correctness of our GPU implementation is checked with the results using the MATLAB CPU code.

3. Accelerating MATLAB IPT Functions

The MATLAB functions that we selected (shown in Table 2) from the Image Processing Toolbox^{tm} (IPT) perform very frequently used media processing operations, from image enhancement, restoration to image transforms, filtering, arithmetic, morphology, etc. Many of these functions are performance-critical and have already been accelerated in MATLAB using C code through the MATLAB MEX interface. These functions also serve as building blocks to many other algorithms and thus have a strong impact in the scientific computing community.

We classified those functions into four categories based on their inherent data-parallelism and algorithm characteristics. We name the first category (Category A) of functions data independent. The functions in this category operate independently on each data element (or pixel) with no communication between elements. Such functions fit naturally on a GPU and require relatively little effort in order to optimize and achieve high kernel performance. Bandwidth, including CPU-GPU transmission and off-chip GPU memory access, is usually the limiting factor for those functions, due to small computational load per data element. We call the second category (Category B) of functions data sharing. The functions in this category also possess abundant parallelism. However, the computation of a single element requires neighborhood information. In order to achieve good speedups on such functions, we need to carefully balance thread-level parallelism and data reuse, which are used to either hide high off-chip GPU memory access latency or to reduce the number of memory accesses. We also need deep knowledge of the underlying GPU architecture, so that we can map GPU resources such as arithmetic logic units (ALUs), registers, shared memory, on-chip caches and threads efficiently in order to achieve high performance. The third category (Category C) of functions, algorithm dependent, have inherent parallelism, however in order to extract this parallelism we need to fundamentally re-think the algorithm with respect to the sequential MATLAB implementation on CPU. The speedups that we can expect from this category of functions may be limited by data dependencies, communication or sequential portions of the algorithm. We call the final category (Category D) of functions data dependent. These applications have very difficult to extract parallelism, due to fine grain communication, synchronization and data dependencies. In the following sections we elaborate on our experience through case studies in each of the categories.

In Table 2, we also report the kernel speedups of the functions using NVIDIA and AMD/ATI GPUs. Both GPUs exhibit good speedups against MATLAB CPU implementations. One thing we notice that for GTX 280, there is often a significant performance difference between CUDA and OpenCL implementation, even with the same kernel code and same configuration. The reason may be due to the compiler difference. For different GPUs, we develop different kernel functions. The reason is due to the hardware impact discussed in Section 2.1. For AMD/ATI GPUs, we use the vector type float4 instead of the scalar/float or float2 type and we assign more workload in each thread and thread group/block for ATI/AMD code. The further details can be referred in the code on the project website [15].

4. Category A: Data Independent

The IPT functions in this category feature abundant parallelism and independent computation upon each data element. Function *intlut* transforms an image into another image using a lookup table. The input pixel value is used as an index to the lookup table and the result is the output pixel value. *Imlincomb* combines several images using a linear combination of the input pixel values. Function *imadjust* transforms each pixel value using a given value mapping formula.

Our optimizations of the IPT functions in this category focus on the effective utilization of bandwidth. For example, pixels in input images for *imadjust* are represented as type ‘unsigned char’. We pack 4 input pixels into one scalar integer or 16 input pixels into one vector type of int4(128-bit or 4 32-bit integers). The vector type of int4 fully utilizes the register width and transmission bandwidth of RV870, while the scalar integer type may be sufficient for NVIDIA GTX 280 (if CUDA is used). Such packing requires bitwise operations to extract the field of interest from a 32-bit number, thereby adding some computational overheads. However, the savings in data transmission (GPU off-

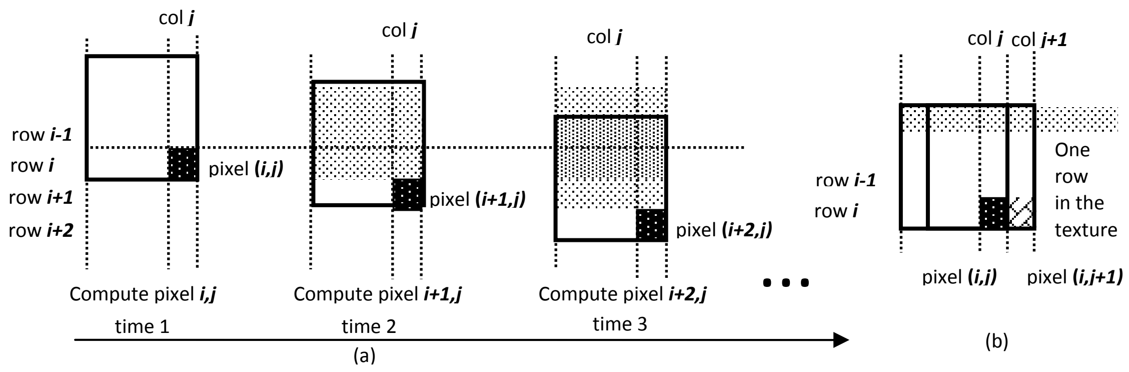


Figure 1. Data sharing patterns exploited in *conv2*. (a) Intra-thread sharing in the register file (b) Inter-thread sharing using texture cache. The black rectangle indicates the input pixels required to compute a given output pixel. The shaded area within the rectangle indicates which pixels are shared/reused.

chip memory accesses) due to packing overweigh such computational overhead.

We also applied the same packing scheme to two other functions in this category, *intlut* and *imlincomb*, as they have a low computation computational requirement per data element and thus are bandwidth bound. Overall, the functions in this category readily achieve dramatic kernel speedups (up to 1385X as shown in Table 2). On the other hand, the relatively low computational requirements of these functions may expose the CPU-GPU transmission as a bottleneck, especially when the image sizes are small (e.g., our experiments show 0.3X or 70% slowdown for *imadjust* on images with 256x256 pixels in terms of overall speedup). Therefore, such functions need to be combined with additional GPU processing to amortize the CPU-GPU transmission cost.

5. Category B: Data Sharing

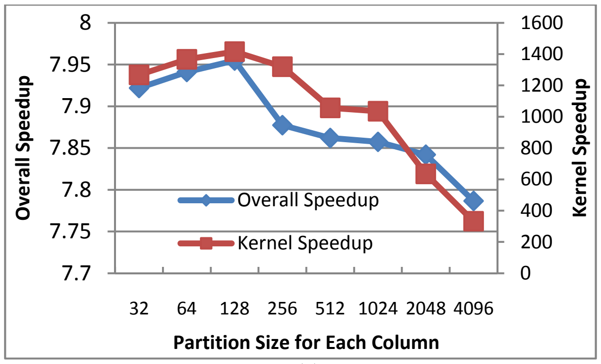
The functions in this category also contain abundant parallelism. However the processing of elements/pixels requires information of neighboring pixels. The selected MATLAB functions in this category include: *edge*, *imregionalmax*, *ordfilt2*, *conv2*, *mean2* and *imdilate/imerode*. Function *edge* (the Prewitt edge detector) uses a maximal of 8 neighborhood pixels to approximate the gradient or the derivative at the center pixel point and therefore is used to recognize the areas where intensity changes sharply. Function *imregionalmax* compares each pixel in a 2D image with their 8 neighborhood pixels and outputs 1 if all neighborhood pixels are smaller, otherwise it outputs 0. Function *ordfilt2* is similar to *imregionalmax*, however the output pixel is equal to the n^{th} greatest element from the set of neighbors. Function *mean2* is a classical reduction operation. Our implementations of 2D convolution (*conv2)* on ATI RV870 and *imdilate/imerode* on NVIDIA GTX280 are discussed in detail in the following case studies.

5.1. Case Study: *conv2*

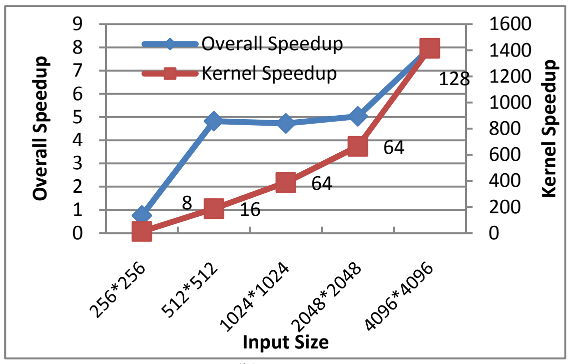
Convolution is widely used in image processing to sharpen the edges or blur and remove noise from an image. The *conv2* function takes as input a source image A, a filter B and outputs an image C. Intuitively, in convolution we drag the filter B over each pixel of the input image A and multiply and accumulate the overlapped input elements to the filter elements to generate an output pixel. More formally, the formula to compute one output pixel $C_{m,n}$ is as follows: $C[m,n] = \sum_{j=0}^{J-1}\sum_{k=0}^{K-1} B[j,k]A[m-j,n-k]$, where J and K are the width and height of the filter respectively. If the filter size is comparable to the source matrix, we can also use Fast Furrier Transform (FFT) to compute convolution [14]. However, to use FFT, the filter matrix has to be padded so that it becomes the same size as the input image. CUDA SDK provides sample convolution code using such an implementation. However, for many applications, which use small filters, such as 5x5 or 7x7, convolution using FFT is not an efficient approach. In this work, we focus on the case when the filter matrix is small, and we assume a 7x7 filter.

In our implementation we use ATI/AMD CAL for maximum optimization and texture cache support (the current OpenCL drivers do not support texture memory). In our first implementation we use one thread to compute one output pixel. That means that each thread reads 7x7 filter pixels, 7x7 source pixels and performs 49 fused multiply-add (FMA) operations. Thus the ratio of texture operations (or off-chip memory accesses) to ALU MAD (multiply-add) instructions (TEX:ALU) is 2:1. Thus the performance of this approach is texture bound. One way to improve the bandwidth efficiency in CAL is to pack data into vectors to enable data reuse. We improve our first implementation, by using vectors of float4 for output and having a single thread compute 4 output pixels in the same row. In this case, we need 7x7 filter pixels, 7x10 source pixels and we perform 4x(49) FMA operations. Thus we reduce the TEX:ALU ratio to 1:1.6. This approach is, however, still texture bound, since we need to have a TEX:ALU ratio of at least 1:5 in RV870 to be able to keep all the computational resources busy. The reason is that each SIMD engine in RV870 contains 80 streaming processors, and can support only 16 texture fetches per cycle [17], for a TEX:ALU ratio of 16:80 or 1:5. We propose a novel approach to solve this problem, by enabling two types of data reuse, *intra-thread* reuse and *inter-thread* reuse, and leveraging texture caches to further reduce memory accesses.

In our approach, we let each thread compute multiple output pixels from the same four columns. Note that computing multiple pixels from the same four columns also satisfies the memory coalescing request of RV870: memory accesses from different threads in the same wavefront (the same as a warp in CUDA except that a wavefront has 64 threads) using consecutive memory addresses. We load the 7x7 filter into 14 float4 registers and use it for the lifetime of the thread. To illustrate how this approach achieves intra-thread reuse, consider the simplified case when one thread is responsible for computing one column of pixels (in practice we use float4 so each thread works on 4 columns). As shown in Figure 1(a) (at time 1) a thread loads 7x7 input pixels



(a)



(b)

Figure 2. Speedups of GPU (ATI RV870) over CPU for *conv2*. (a) The impact of column-partition size (number of column elements computed by one thread). (b) Speedups for different input sizes. Partition sizes for different size input images are shown along the line.

(the dark rectangular region) and multiply-adds them with the filter to obtain the output pixel (i,j). At time 2, when the same thread computes pixel (i+1,j), it can reuse the majority (6x7) of those input pixels, which are shaded in Figure 1(a). In general, for a 7x7 size filter, we may reuse each input row of 7 pixels up to 7 times. Therefore, in our approach when a row of input pixels is loaded from memory into registers, it is used to contribute to up to 7 output pixels in the same column and then it is discarded. In the next time step another input row is loaded into those registers and used to contribute again to 7 output pixels. When an output pixel has accumulated the results of all 7 rows, it is written to memory. This way, we maximize data reuse, while minimizing the number of registers which hold the temporary data.

Figure 1(b) shows how our scheme achieves inter-thread data reuse. As shown in the figure, thread j and thread (j+1) are in the same wavefront and they compute columns j and (j+1) respectively in the output matrix. Given the locked-step execution in a wavefront, many threads will access the same row of the source image (shaded in the figure) at the same time. With the texture cache, this row will be stored there after the first access. Subsequent accesses to the same row from different threads will all hit the cache and experience very short latency. The degree of such inter-thread reuse is determined by the overall cache size and the block size: the larger the texture cache size and block size, the more the inter-thread data reuse.

Optimizing the TEX:ALU ratio through data reuse is only one dimension of optimizing GPU performance. We also need to carefully balance the amount of thread-level parallelism available (number of warps in CUDA or wavefronts in CAL) to effectively hide memory latency. Thus, we experiment with partitioning each column, and letting each thread work on only part of the column in order to increase thread-level parallelism. The trade-offs in using a different size partitions per column are evaluated in Figure 2(a). For a 4096x4096 image, a partition of 128 elements is optimal. Increasing the size of the partition beyond 128 impacts performance due to reduced thread-level parallelism. Reducing the size of the partition, on the other hand, reduces the amount of intra-thread data reuse. Furthermore, with the partition size of 128, the TEX:ALU ratio is about 1 : 6.4 for image size of 4096x4096, which is close to the resource ratio 1:5. This implementation achieves 733 GFLOPS and kernel speedup of 1415x compared to CPU. When the CPU to GPU data transfer time is included, we obtain up to around 8x speedup, as shown in Figure 2(b). For images with different sizes, we select partition sizes such that we generate enough wavefronts (at least 128) to hide memory access latencies.

To better understand the impact of hardware support from different GPUs, we also applied the abovementioned optimization on GTX 280 using CUDA. Given the smaller register size of GTX 280 compared to RV870, we store the 7x7 convolution filter in the shared memory rather than allocating 49 registers in each thread. We use texture memory for the source image and let each thread processes a partition of a column as in our RV870 experiments. Surprisingly, the performance of GTX 280 using our approach is a little less effective: a kernel-only throughput of 352 GFLOPS is achieved for an image of size 4096x4096 compared to 733 GFLOPS using RV870. Even for HD4870 (RV770) a higher performance is achieved (515 GFLOPS). One likely reason is that the L1 texture cache size for each TPC of GTX 280 (exact size undisclosed) is smaller than the texture cache per SIMD in RV870/RV770 (exact size undisclosed). The overall speedup over CPU is 16x, higher than the ATI results due to more efficient CPU-GPU transmission with CUDA. Our CUDA implementation is also faster than 2D convolution (conv2) provided in Jacket [18], a proprietary commercial GPU engine for MATLAB. Even excluding the CPU-GPU transfer time (using gforce in Jacket), 2D convolution of one 7x7 kernel on a 4096x4096 image takes 0.258s using Jacket while our CUDA implementation with MEX interface takes 0.127s with the CPU-GPU transfer overhead included.

Another optimization strategy for convolution on GTX 280 is to use shared memory for data reuse. Our optimized shared memory version reports a throughput of 330 GFLOPS for a 4096x4096 image, a little less effective compared to our proposed approach. As discussed earlier, the CUDA SDK sample code uses FFT to perform convolution, which is not suitable for small filters. With a filter size of 7x7 on a 4096x4096 image, the achieved kernel throughput of the CUDA SDK sample code is 12 GFLOPS.

5.2. Case Study: *imdilate* and *imerode*

Dilation and erosion are two common morphological operations used in various computer vision and image processing tasks such as blob analysis and text detection. Both dilation and erosion are binary operations. They take an image and a binary kernel matrix as inputs and produce an image as the output. Dilation can be defined as follows: pixel (*x*,*y*) of the output image is the maximum pixel value of the neighborhood defined by the

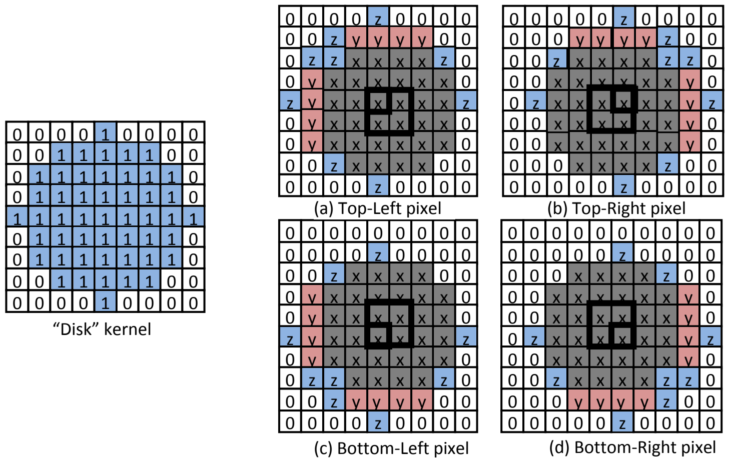


Figure 3. Reusable components identified when processing 4 pixels at a time, using a "disk" kernel. The non-zero cells in the kernel define the neighborhood for computing the minimum maximum. Gray cells marked with 'x' are common to all four pixels. Red areas marked with 'y' are common to two pixels. Blue areas marked with 'z' are unique to each pixel and therefore not reusable.

kernel centered at (*x*,*y*) of the input image. Similarly erosion takes the minimum of the neighborhood.

In our GPU implementation, besides data reuse optimization, we also propose computation reuse. We compute a square of four output pixels in one thread so as to reuse computations that are common to some of the four pixels. We explain our method using the example kernel shown in Figure 3. When the kernel is applied to an image and we compute four output pixels at a time, various common computations that can be reused can be identified. These common segments are illustrated in Figure 3. The input pixels in the gray area (marked with 'x') in (a) are the same as those in the gray areas in (b), (c) and (d). Therefore, the maximum of the input pixel values in the gray area in (a) should be the same for gray areas elsewhere. This makes it possible to reuse the computation of maximum four times, once for each of the four output pixels. Similarly the computations of red areas (marked with 'y') are reused twice. However blue areas (marked with 'z') are unique to each output pixel and therefore computed for each pixel separately. It is fairly easy to identify the reusable segments from a given kernel. For example the computation of the non-zero cell (*i*,*j*) of the kernel is reusable in all four pixels if cells (*i*-1,*j*), (*i*,*j*-1) and (*i*-1,*j*-1) all are non-zero. It should be noted that one might decide to compute even larger number of output pixels at a time (e.g., 8 or 16). For small sized binary kernels, this does not result in additional performance gains because the amount of reusability is low for smaller kernels. For larger kernels, this is beneficial, however at the cost of increasing complexity in identifying the reusable segments.

Our GPU implementations on NVIDIA GTX 280 achieves up to 952x (949x) kernel speedup for *imdilate*(*imerode*) against the CPU version in MATLAB.

5.3. Summary

In summary, to optimize the functions in this category, we mainly focus on two directions. First, we aim to optimize the TEX:ALU ratio by leveraging various types of memory/ computation reuse, which may require utilization of different hardware structures in different GPUs. Second, we need to carefully balance data/computation reuse with thread-level parallelism so as to ensure there are enough threads to hide memory access latencies.

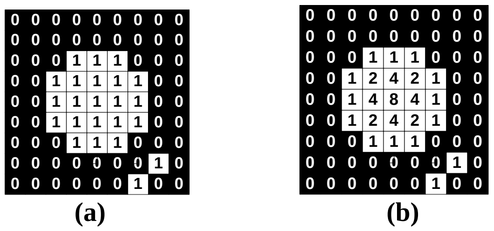


Figure 4. Example of EDT [5]. (a) Original image (b) Distance map from white to black pixels. The distances are squared.

6. Category C: Algorithm-Dependent

In order to parallelize and optimize the *algorithm dependent* functions on GPUs, in-depth algorithm analysis is required. In the following, we present case studies on the 2D Euclidean distance transform (EDT) in the function *bwdist*.

6.1. Case Study: 2D Euclidean Distance Transform

To illustrate the EDT algorithm, consider a 2D binary image *I*, where each pixel *p* is either black (0) or white (1). The output of the algorithm is a map *D*, whose value in each pixel *p* is the smallest distance from this pixel to a black pixel in the image *I*, and the distance is computed as the Euclidean distance:

$$\boldsymbol{D(p) := min\{distance(p,q) \mid I(q) = 0\}}, \quad \boldsymbol{distance(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}}.$$

A numerical example of the EDT is given in Figure 4. For a detailed discussion on EDT, including details on different algorithms and interesting applications, we refer the reader to [5].

6.1.1 Efficient EDT Algorithms

Due to the ubiquitous nature of this algorithm and its O(n^4) worst case and Θ(n^3) average case complexity [5] for the brute force algorithm (where *n* is the width/height of the image), a lot of research effort has been invested in developing efficient EDT algorithms. In general, efficient EDT algorithms can be classified into three categories depending on the way pixels in the image are processed: *ordered propagation*, *raster scanning* and *independent scanning* [5]. Order propagation algorithms start from seeds and progressively transmit information to neighboring pixels. We decide against propagation approaches, due to the number of global propagation steps and data dependencies between each propagation step. Raster scanning algorithms (such as Danielson's algorithm [3]) use a 2D mask and scan the pixels of each row top to bottom, and then bottom to top. Unfortunately, raster scanning algorithms do not produce exact results unless a corrective step is applied. Approximate algorithms have already been implemented on GPUs using ordered propagation [2][10] or raster scanning algorithms[12].

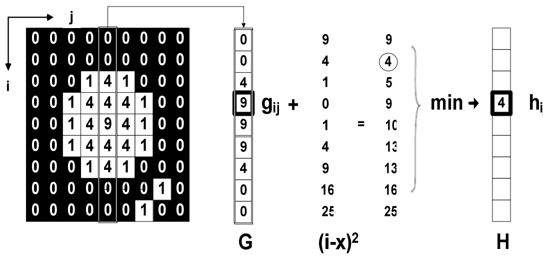


Figure 5. Second stage of Saito's EDT algorithm [11].

However, in this work we are interested in an exact EDT solution to match MATLAB results. Independent scanning algorithms [7][11], compute independently on the rows of the image, followed by independent computations on the columns of the image. In this work, we use the CPU-based algorithm from Saito et al. [11] as a reference for our GPU implementation.

6.1.2 Saito's EDT Algorithm

Saito's EDT algorithm consists of two stages. In the first stage, we scan each row of the image. For each pixel *p* on a row, we compute the squared distance to the closest black pixel on the same row. This is easily implemented by a forward scan of the row, followed by a backward scan of the row. The first stage of the algorithm is more formally given by:

Transformation 1. Given an input image *I*, perform independent scans on the rows of *I* and generate an image *G* defined by:

$$G(i,j) = min_y\{(j-y)^2 | I(i,y) = 0\} \qquad (1)$$

The second stage of the algorithm is more involved and is pictorially represented in Figure 5. In this stage we work on the columns of image *G* as produced by the previous transformation. Consider a pixel g_{ij} in column *j*. To compute the final value of g_{ij}, we have to add each element of column *j* to the vertical distance between this element and g_{ij} and find the minimum of those sums. Formally this transformation is given by:

Transformation 2. Given an input image *G*, perform independent scans on the columns of *G* and generate an image *H* defined by:

$$H(i,j) = min_x\{G(x,j) + (i-x)^2\} \qquad (2)$$

6.1.3 Experimental Methodology

The performance of the EDT algorithm is dependent on the input images as black and white pixels are processes differently. In order to evaluate the performance, we used the test images suggested by Fabbri et al. [5]. Some of the images are shown in Figure 6. These images test different pathological cases for the EDT. For instance the corner-pixel image is the worst case for a sequential brute-force algorithm and the image containing circles exposes the highest amount of errors in an inexact implementation of EDT. We also evaluated the performance on the inverse of those images, and we append an "_i" to the image name to indicate the inverse. The size of each input image used is 1024x1024 pixels. Since we wanted to use scalar (instead of vector) shared memory operations we chose to use CUDA and the GTX 280 card in the following optimization analysis. Scalar shared memory operations in ATI cards can be also exploited in the future once related hardware details are out. On the CPU, we compared to the optimized MATLAB version based on Breu et al. [1].

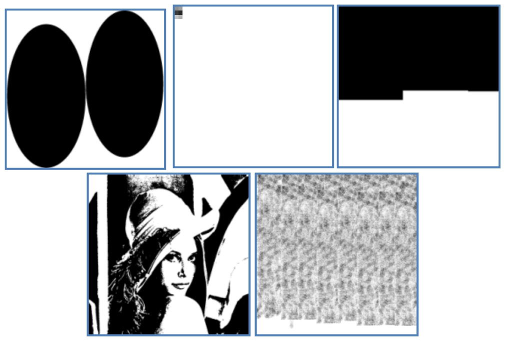


Figure 6. Images used to evaluate EDT. The inverse of those images are also used. (a) black/white circles (b) black/white pixel in the corner (c) half-filled (d) image of Lena (e) random filling.

6.1.4 GPU Implementation and Experimental Results

A summary of our experimental results is presented in Table 3. We performed several iterations of optimizing EDT, which are given in the table as (a), (b) or (c). Since Saito's algorithm consists of two independent transformations of the image, we first present our GPU optimization process of the first transformation, followed by the second.

First transformation: The first stage of the EDT algorithm scans each row of the image independently. In order to fully utilize the off-chip memory bandwidth of the GPU we allocate a large block of threads (512), and each thread loads 2 pixels from global memory into the on-chip shared memory. After the row has been read into shared memory in parallel, we use a single thread to perform the scan[1] and compute the distances from the white to

[1] The scan operation essentially searches for black pixels on the row. Once a black pixel is found, the distances from the black pixel to the previous white pixels can be computed. We believe that it is difficult/inefficient to adapt scan primitives proposed in previous work [13] [4]for our purpose. The reason is that, the scan primitives proposed in [13] [4]are most efficient when the same binary operation (such as addition) is to be performed across all elements of the array. However, in our scan function, black and white pixels are treated differently and are randomly distributed across the row. We have optimized our implementation of scan, so that we read each pixel from shared memory only once instead of two times, which is the case when performing a

Table 3. Execution time of EDT on NVIDIA GTX 280 vs. MATLAB implementation. The GPU time is divided into the time to execute the first and second transformation. For each transformation, we implemented several optimization strategies, which are given as (a), (b), or (c). Column "Total GPU execution time" gives the total execution time including the time to transfer the input/output data between CPU and GPU, and using our best optimization strategy.

Test Image	Transformation 1. Kernel execution time (ms)			Transformation 2. Kernel execution time (ms)			Total GPU execution time (ms)	CPU execution time (ms)	Overall Speedup	Kernel Speedup
	(a)	(b)	(c)	(a)	(b)	(c)				
half_filled_i	4	26	1	53	16	13	45	230	5x	15x
half_filled	4	19	1	53	14	11	44	214	5x	19x
corner_i	3	0.6	1	54	5	0.6	32	291	5x	147x
corner	3	64	1	53	25	24	56	168	5x	12x
circles_i	5	43	1	53	10	9	42	343	7x	28x
circles	3	13	1	53	7	4	36	286	10x	68x
random_i	4	1	1	53	6	1	33	318	11x	215x
random	5	4	2	53	8	2	36	360	9x	69x
lena	4	16	1	53	6	3	36	326	9x	81x

the black pixels on the row. When the computation is done, we again use all 512 threads to write the data back to global memory. The advantage of using a single thread to perform the scan is that it results in only *n* reads from shared memory where *n* is the number of pixels on a row and only *m* distance computations, where *m* is the number of white pixels on a row. The downside to this approach is that many threads remain idle and perform no work during the scan. The execution time of this kernel is given in Table 3: Transformation 1 (a). Next, we measured that the time to load the image into shared memory and write it back out to global memory is only .66 ms, compared to 3-5 ms of kernel execution time. Therefore, the majority of the time is spent in the sequential scan of the row. Our next optimization strategy attempts to remove this bottleneck.

In our second approach, we tried using all 512 threads to compute on a row at the same time. Each one of the 512 threads works on two pixels of the row. For each of the two pixels, if the pixel is white, then the thread scans the row to the left and to the right to find the closest black pixel and compute the Euclidean distance. This strategy results in many redundant reads from shared memory. In the worst case, each pixel scans the whole row, which results in n^2 reads instead of n. This strategy creates divergent code as well as bank conflicts in shared memory. Interestingly, our results from Table 3: Transformation 1 (b) show that this brute-force approach actually outperforms the single-thread approach on some of the input images that we studied. The most prominent speedup comes from 'corner_i', where almost all of the image pixels are black. In this case, the kernel execution time 0.68ms close to 0.66 ms, just the time needed to load the data in shared memory. This strategy is also very fast for the random images, where only a couple of shared memory reads need to be performed in order to find a neighboring black pixel.

In our third strategy we combine the benefits of both previous approaches. We wanted to increase the amount of parallelism, while at the same time limit the amount of redundant accesses to shared memory. In this approach, after the initial load of a row to shared memory, we used 16 threads (from the first warp in a thread block) to compute on a row. Each thread scans a region of 64 pixels and computes the Euclidean distances for that region. Since some pixels cannot be computed without visiting pixels outside of the region, threads communicate using shared memory. In particular, after the initial scan of 64 pixels, each thread writes to shared memory the locations of its leftmost and rightmost black pixels. Subsequently each thread reads the information from its neighbors and completes the computation. We also reduced the total number of threads, which initially load the row into shared memory to 128 from 512. This allows us to map three thread blocks to a GPU computational unit vs. only one thread block. Overall, this approach results in the best performance as visible from Table 3: Transformation 1 (c).

Second Transformation: In this stage Saito et al. [11] and Meijster et al. [7] propose different implementations based on column scans in order to speed up the direct implementation of Transformation 2. The basic goal in both algorithms is to restrict the number of pixels that are scanned in order to perform distance minimization. Unfortunately, previous algorithms assume that column scans are performed by a single CPU thread. In our work we want to avoid using a single thread to perform a column scan and we find the fast CPU algorithms proposed by previous work are difficult to parallelize due to data dependencies. Instead, in our work we use a more direct implementation of Transformation 2. First, we use 256 threads to load a column into shared memory. Then, each thread computes four elements of the column by adding the vertical distance to the column and finding the minimum as shown in Figure 5. The performance of this kernel is shown in: Table 3, Transformation 2 (a). We optimize the performance of this operation in two major ways. First, we modify the algorithm slightly. We observed that we can stop the scan up or scan down the column whenever we encounter a 0 or a black pixel. We also use a single column scan to compute four output values by reusing the value read from shared memory. Second, we pre-computed the vertical distance values (i-x)2 and loaded those in constant memory, so that they can be re-used by all thread blocks, to reduce the amount of computation. Constant memory results in very fast access times, as long as all the threads in a warp access the same element. Since each thread scans up and down starting from its pixel, the vertical distances for different threads are initially the same (e.g., 0, 1, 2, 3, etc.). Therefore, threads will access the same element in constant memory, until they diverge. Threads diverge when they find a black pixel during the scan or when they reach the end of the column. These optimizations let to significant improvement in the execution time of the second transformation as shown in: Table 3, Transformation 2 (b). Finally, we remove the bank conflicts to global memory due to the column-wise access by transposing the image before and after the second stage of the algorithm. We

forward and a backward scan. To do that, we remember the previously detected black pixel on a row, so that when we encounter the next black pixel on the row, we have enough information to compute the minimum distance for all the white pixels in between.

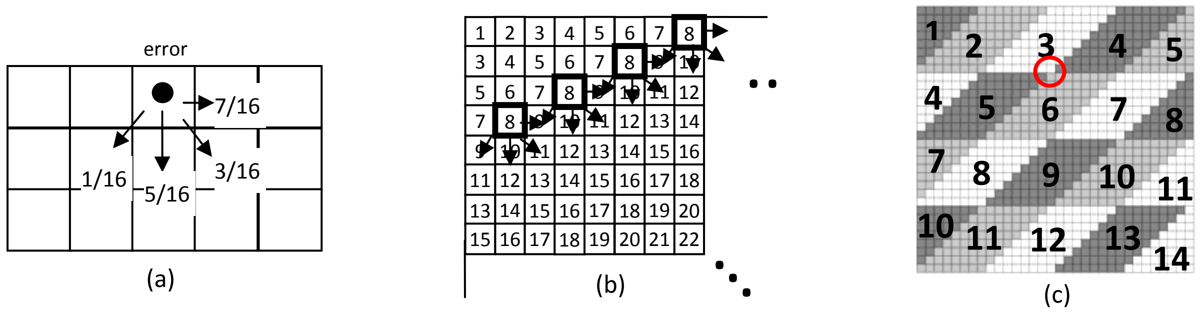


Figure 7. Dithering (a) error propagation pattern; (b) optimal parallel execution schedule proposed by Metaxas [8]; (c) proposed parallel execution schedule of thread blocks, assuming a 32x32 image and 8x8 thread-blocks.

reuse the highly optimized transpose provided with the CUDA SDK. The results after this optimization are given in Table 3, Transformation 2 (c). With all these optimizations, the overall speedups that we achieve on the GPU range between 5x and 11x (including the CPU-GPU transfer time and MEX overheads) compared to the highly optimized CPU code in MATLAB. In comparison, the kernel speedups reach up to 215x.

6.2 Summary

Another function we studied in this category is radon transform. One key optimization is the input/output block-tiling using shared memory, which helps to generate coalesced global memory writes. Another one is to use atomic operations to generate extra thread-level parallelism. Our performance results show 80x(84x) for overall speedup (kernel speedup) using CUDA on GTX280. Detailed information can be referred to the project website [15].

From the two functions that we studied in this category, we can see that we need to carefully re-think the algorithm and quite often the resulting GPU algorithm is fundamentally different from the sequential CPU implementation. Furthermore, the optimization strategies are also algorithm dependent in terms of how to exploit thread-level parallelism.

7. Category D: Data Dependent

The IPT functions in this category feature fine-grain data dependency and communication, making the functions difficult to port to GPUs or requiring substantial programming effort to obtain modest speedups. We illustrate our findings using function *dither*.

Case study: *Dither*

Dithering, also called digital half-toning, is a technique used to render a color/gray-scale image on a device with bi-level displays. In order to generate the output image, the algorithm scans the input image from left to right and from top to bottom in a strict order. In each step, the input pixel is compared against a threshold value to decide whether the corresponding output pixel should be set to 1 or 0. The difference between the original pixel and the threshold is called error. In order to smooth out the output image, the error is propagated over to the neighboring pixels as shown in Figure 7(a). This error diffusion algorithm was first proposed by Floyd et al. [6] and is the one used in MATLAB. The diffusion algorithm has been considered inherently serial for a number of years [8], since the pixel in the lower right corner of the image depends on all the other pixels in the image. However, Metaxas [8] analyzed the data dependencies and proposed an optimal parallel processing schedule, shown in Figure 7(b). This schedule enforces the invariant that a pixel p(i,j), can only be computed after all the pixels that it depends on {p(i-1,j), p(i-1,j-1), p(i,j-1), p(i+1,j-1)} have already finished computing. Such schedule reveals multiple elements that can be processed concurrently. For example, in Figure 7(b), we can see that at time 8 we can compute 4 pixels in parallel as all of their required pixels have been computed during time 7. Using this algorithm, we can complete the diffusion process in 2*n+n steps instead of n*n steps required by a sequential algorithm. For a 2kx2k size image, this translates into a theoretical speedup of: 2048*2048/(2*2048+2048) = 683x, which motivates an attempt to parallelize this algorithm on a parallel processor.

In order to implement this algorithm on a GPU, we may use one thread to work on one pixel. However, we need a mechanism to enforce fine grain synchronization of threads and thread blocks to follow the processing schedule. Current GPUs provide mechanism for synchronization of threads within a thread block. But there are no explicit provisions for global synchronization (across thread blocks). Thus we divide our discussion into two parts. First we present an approach for implementing error diffusion within a thread block. Then we present our solution for providing synchronization across thread blocks.

Consider the pseudo code implementation in Figure 8, which presents our diffusion approach for a single thread block. The kernel in Figure 8 takes a source image and a processing schedule as inputs, and generates an output image. Each thread in a thread block is responsible for computing 1 output pixel. First, each thread in the kernel obtains its processing step my_step from the pre-computed schedule. The threads will also collectively load a block of the input image into shared memory to improve memory performance. Next, each thread in a thread block will spin inside the for-loop (line 9) until the current time step becomes equal to my_step (line 10), so that it can begin computation. Note that concurrently executing threads will diffuse to overlapping pixels, as can be seen from Figure 7 (b). To prevent data races we may use atomic operations, such as atomicAdd() during diffusion step. Alternatively, we can split the error diffusion into two parts and use _syncthreads() to prevent the data race (not shown in the figure). We choose the _syncthreads() approach, because it performed faster than atomicAdd in our experiments. Notice that this approach computes only very small size images, since it is limited by the number of threads per block and the amount of shared memory. In our implementation, we can only support an image size up to 32x16 because of the maximum number of threads in a thread block is 512 in our NVIDIA GPU. Due to the sequential processing order in the schedule, only a fraction of threads are active at any given time, resulting in low resource utilization.

```
dither_block_gpu( input_image,  schedule, output_image)

tx = threadIdx.x;   ty = threadIdx.y;    // Obtain thread index
my_step = schedule[ty, tx];              // Read my time step
// load an input block into shared memory
SM[ty, tx] = input_image[ty, tx]);
__syncthreads();                         // barrier

for( i=1; i<= num_steps; i++){
   if(my_step == i){     // wait for my time step
     compute_pixel;  // compute pixel in SM and propagate error
   }
   __syncthreads();       // barrier
}
```

Figure 8. Pseudo-code implementation of dithering on GPU using a single thread block.

In order to compute larger images, we need a hierarchical solution, where a processing schedule is also computed for thread blocks and thread blocks are scheduled for execution based on that schedule. We envisioned two different mechanisms to synchronize thread-blocks globally. In the first mechanism, we can construct a global-barrier using atomic operations in global memory. The second mechanism is to use the CPU to dispatch thread blocks at the appropriate times and synchronize the completion of each thread-block-step by using cudaThreadSynchronize() (otherwise the kernels may be executed asynchronously). We adopted the second approach and used CPU to dynamically create a number of thread blocks for launch at each time step. One issue with dividing the image into sub-blocks is that we need to minimize the data dependencies between thread blocks. The naïve partitioning of the image into square sub-blocks does not work, due to pixel dependencies along the anti-diagonal. After analyzing the dependencies, we designed a novel approach which divides the image into rhombus-shape blocks (and triangles at the corners), which minimizes data dependencies between blocks to only the borders of adjacent blocks. Based on this layout, we also computed a parallel schedule for computing the thread blocks, given in Figure 7(c). Note, that this schedule is more conservative (requiring more steps) than the intra-block schedule in (b). The reason is that the corners of two rhombus blocks meet at one pixel (as circled in Figure 7(c) for blocks labeled 4 and 5), which causes a data race on that pixel. To avoid this race, we forced the blocks executing in parallel to be further apart. We also modified the intra-block schedule from Figure 7(b) to fit the new shape of thread blocks.

Notice that sub-dividing the image into blocks reduces parallelism, since the independent elements that we can compute are restricted to the diagonal of one thread block. The performance that we obtained for our implementation is given in Figure 9. For large images, we obtain an overall speedup of about up to 3.5x, and up to 10.3x kernel speedup, while we incur an overall slowdown for images smaller than 512x512. In addition to the low resource utilization and synchronization overhead discussed above, this kernel is also memory bound, since very little computation per pixel is required for error diffusion. Since there is no data reuse, it is difficult to mitigate this memory bottleneck. Overall, we conclude that inherent limited parallelism and the requirement for fine-grain communication synchronization make programming very hard for such applications.

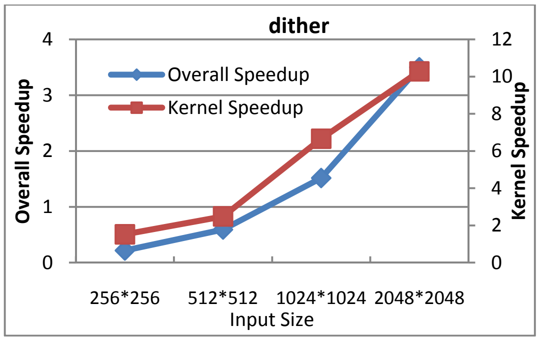


Figure 9. Speedups of dither on GPU (GTX 280) over CPU.

8. Conclusions

In this paper, we present our experience in developing high performance GPU code for a dozen functions from the MATLAB Image Processing Toolbox. Based on the algorithm characteristics, we group the functions into four categories: data independent, data sharing, algorithm dependent and data dependent. Then, we present detailed studies to establish optimization strategy for each category of functions. Our results show drastic speedups for the functions in the data-independent or data-sharing category; and moderate speedups for those in the algorithm-dependent category. For the functions in the last category, fine-grain synchronization/data-dependency requirements make them a poor fit for GPU implementation.

9. Acknowledgement

We thank the anonymous reviewers for their valuable comments. This work is supported by an NSF CAREER award CCF-0968667 and an AMD grant and equipment donations.

10. References

[1] H. Breu et.al. , Linear Time Euclidean Distance Transform Algorithms, The Insight Journal, 2006.

[2] N. Cuntz and Andreas Kolb , Fast Hierarchical 3D Distance Transforms on the GPU, EUROGRAPHICS 2007.

[3] P. Danielson, Euclidean distance mapping", Comput. Graph. Image Proc. 14, 227–248., 1980.

[4] Y. Dotsenko et. al, Fast Scan Algorithms on Graphics Processors, ACM ICS 2008.

[5] R. Fabbri, et al. , 2D Euclidean Distance Transform Algorithms: A Comparative Survey, ACM Computing Surveys, 2008.

[6] R. Floyd, L. Steinberg, An adaptive algorithm for spatial grayscale, Proceeding of the Society for Information Display,1976.

[7] A. Meijster, et al, A general algorithm for computing distance transforms in linear time, ISMM, 2000.

[8] P. Metaxas, Parallel digital halftoning by error-diffusion, ACM International Conference Proceeding Series, 1995.

[9] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 2008.

[10] G. Rong, T. Tan, Jump flooding in gpu with applications to Vornoi diagram and distance transform, ACM I3D, 2006.

[11] T. Saito and J. Toriwaki, New algorithms for Euclidean distance transformations of an n-dimensional digitised picture with applications, Patt. Recog. 27, 11, 1551–1565., 1994.

[12] J. Schneider et. al, GPU-Based Real-Time Discrete Euclidean Distance Transforms With Precise Error Bounds,VISAPP 2009.

[13] S. Sengupta, M. Harris, Y. Zhang, J. Owens, Scan Primitives for GPU Computing, Graphics Hardware, 2007.

[14] T. Stockham, High speed convolution and correlation with application to digital filtering, Digital Processing of Signals, McGraw-Hill, 1969.

[15] https://sites.google.com/site/iptatiproject/

[16] ATI Stream Computing User Guide, 2009.

[17] ATI Radeon™ HD 5870 GPU Feature Summary, http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx.
[18] Jacket: a GPU engine for MATLAB, http://www.accelereyes.com, 2009.
[19] MATLAB Acceleration.
http://www.nvidia.com/object/matlab_acceleration.html
[20] MATLAB MEX files guide, http://www.mathworks.com/support/tech-notes/1600/1605.html, 2009.
[21] MATLAB plug-in for CUDA, http://developer.NVIDIA.com/object/MATLAB_cuda.html, 2009.
[22] NVIDIA CUDA Programming Guide, Version 2.2, 2009
[23] OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/, 2009.