Hybrid Implementation of Error Diffusion Dithering

Aditya Deshpande, Ishan Misra, and P J Narayanan
Centre for Visual Information Technology
International Institute of Information Technology, Hyderabad
India

Abstract

Many image filtering operations provide ample parallelism, but progressive non-linear processing of images is among the hardest to parallelize due to long, sequential, and non-linear data dependency. A typical example of such an operation is error diffusion dithering, exemplified by the Floyd-Steinberg algorithm. In this paper, we present its parallelization on multicore CPUs using a block-based approach and on the GPU using a pixel based approach. We also present a hybrid approach in which the CPU and the GPU operate in parallel during the computation. High Performance Computing has traditionally been associated with high end CPUs and GPUs. Our focus is on everyday computers such as laptops and desktops, where significant compute power is available on the GPU as on the CPU. Our implementation can dither an $8K \times 8K$ image on an off-the-shelf laptop with an Nvidia 8600M GPU in about 400 milliseconds when the sequential implementation on its CPU took about 4 seconds.

1. Introduction

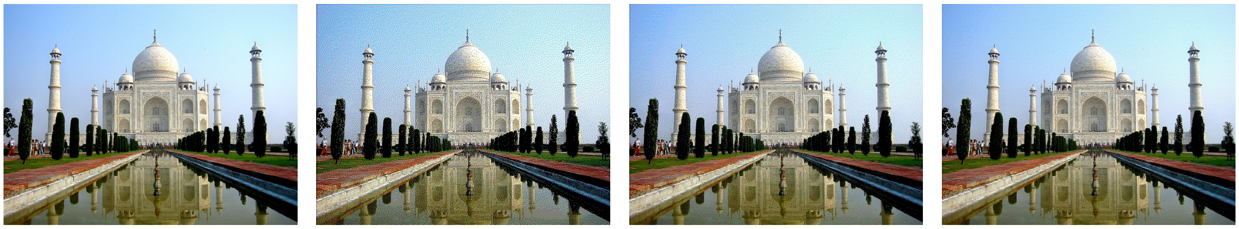
Image processing is seen as a source of rich data parallelism, suitable for processing by a device like the GPU. Common image filtering operations provide high parallelism as each input pixel creates one output pixel value, either independently or by processing a small neighborhood around itself in the input image. The fact that output pixels depend only on input pixels, and that the latter do not change as a result of processing, allows all pixels to be processed in parallel. Such embarrassing parallelism is not available if the operation has a causal dependence on the output of previous pixels. Thus, the output value of "later" output pixels depend on a mix of input pixel values and values of output pixels computed "earlier". A causal processing order for 2D images can be defined by choosing one corner as the first pixel and its opposite corner as the last in processing order. The processing can then proceed along rows or along columns.

The processing step on each pixel may be a linear operation or can involve non-linear computations. Linear operations are easy to parallelize, as the output at each pixel can be written as a linear combination of the input values of some or all "past" pixels. The pixel with the longest dependency decides the running time. This may not be possible if the operation is non-linear. The potential dependency of the first pixel on the last makes such operations inherently sequential and difficult to parallelize.

Several image processing operations involve non-linear, progressive processing of pixels. We use error diffusion dithering as the sample application, because the optimal error diffusion algorithm poses the maximum challenge for a parallel implementation. Dithering is a technique used to create an illusion of a higher color depth in images using a limited color palette. It approximates other colors not available in the color palette using a spatial distribution of available colors, as the human visual system averages the colors in a neighborhood. Applications of dithering vary from printing, display on small devices like cellphones, computer graphics [1], visual cryptography [2], image compression [7], etc. Among the class of dithering algorithms, error diffusion dithering algorithms are popular due to their high quality output.

The Floyd-Steinberg Dithering (FSD) [8] is an optimal error diffusion algorithm [9]. It generates better results than other dithering methods. Its traditional sequential implementation has $O(mn)$ time complexity (m, n being the height and width of the image in pixels). The error distribution scheme processes pixels from left to right and top to bottom. This introduces a sequential dependency of the last pixel on the first. Consequently, FSD problem is difficult to parallelize [4].

In this paper, we present strategies to perform FSD on multicore CPUs, the GPU, as well as a combina-



(a) Original Image (b) Image dithered to two levels (c) Image dithered to four levels (d) Image dithered to eight levels

Figure 1. An example of image dithering. The original and dithered images are of comparable quality when dithered at 8 levels per color channel. The images are best seen in electronic form than in print.

tion of the CPU and the GPU. We show that high performance can be obtained by using appropriate, problem-dependent data layout and by minimizing communications, especially in a hybrid setting. The hybrid approach is especially promising on low-end computers used by millions of people, such as a laptop or a desktop with a decent CPU and a mid or low-end GPU. Practical parallel applications have to focus on such low end platforms to bring the benefits of parallelism to a huge number of users and not merely the users of expensive HPC systems.

2. Previous Work

Metaxas [4] was one of the first to parallelize an error diffusion dithering algorithm. They parallelized the FSD algorithm with $2m{+}n$ parallel steps compared to mn for the sequential version, for an $m \times n$ image. An optimal scheduling order ensured that a pixel was processed as soon as all its data dependencies were met. This scheduling order is same as the knight's chessboard move pattern that we discuss later. The use of a linear array of processors was proposed, with each processor processing three pixels in a row, followed by the pixels in the next row. After a processor processed three pixels, it transmitted errors to the neighboring processor and activated them for processing. This effort was mainly directed at the PRAM processing model.

Zhang et al. take an altogether different approach to parallelizing error diffusion [11]. They dismissed the FSD algorithm for dithering due to the low possible parallelism. They used a pinwheel dithering algorithm [6] instead, which has much more inherent parallelism. The image is divided into blocks of two types in a checkerboard fashion. All blocks of the same type can be processed in parallel and independently of each other. First, blocks of one type were processed followed by processing the other type of blocks. Metrics that take into account the human visual system show that the raster-order FSD approach provides better results compared to the serpentine order error diffusion used in the pin-wheel algorithm [9].

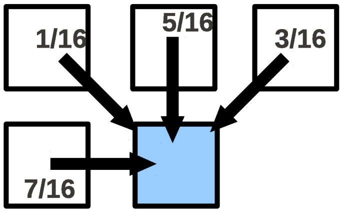


Figure 2. Data dependency of a pixel on its neighbors.

We parallelize the original FSD algorithm as it has perhaps the most challenging data dependency pattern that reduces parallelism. Thus, techniques developed for it and the lessons learned from it will be useful for many operations that are currently considered hard for parallel processing.

3. Floyd-Steinberg Dithering

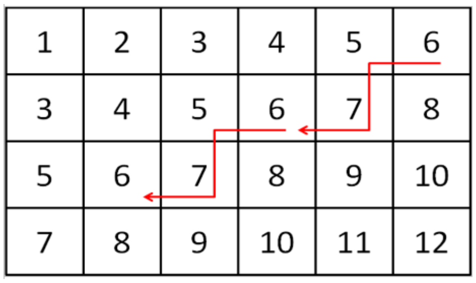
Floyd-Steinberg dithering is optimized for quality, compared to other similar schemes. In this section, we explain the basic algorithm as well as our parallel implementation schemes on multicore CPUs, GPUs, and hybrid computers.

3.1. Basic Algorithm

The basic FSD approach proceeds as follows: For each pixel the accumulated error from its neighbors

0	0	0
0	*	7/16
3/16	5/16	1/16

Table 1. FSD Error Distribution Matrix for the centre pixel.



1	2	3	4	5	6
3	4	5	6	7	8
5	6	7	8	9	10
7	8	9	10	11	12

Figure 3. Optimal scheduling order. Pixels with label n can be processed independently of other pixels, once all the pixels with labels less than n are processed.

(explained later) and pixel value is summed. This value is quantized to the nearest value available in the color palette. The residual quantization error is then distributed to the neighboring pixels in fractions shown in Table 1. Each pixel receives quantization errors from its neighbors and these form the pixel's accumulated error. The process starts with the top-left (or first) pixel. The quantization at each pixel makes the output a non-linear function of all past pixel values. The algorithm is summarized in Algorithm 1. We use Algorithm 2 as our $NearestColor$ implementation to obtain colors by a simple linear scaling. In the case of color palettes, a simple look-up table can be constructed for $NearestColor$.

It is interesting to see the flow of errors in FSD. Each pixel (except the ones on the border) needs the error values from 4 neighboring pixels before it can be processed (Figure 2). Thus, for any pixel, the 4 neighboring pixels need to be processed before processing itself. In a sequential implementation, the rows can be processed from left to right, starting with the first.

The error distribution pattern induces a long chain of data dependency as shown in Figure 2. The last pixel of the image in causal order depends on the first, in theory. The quantization at each pixel is a non-linear operation. The access pattern introduces the following scheduling constraint for each pixel (i, j):

$$\begin{aligned} T(i,j) > \max\{ \quad & T(i-1,j-1), T(i-1,j), \\ & T(i-1,j+1), T(i,j-1)\}. \end{aligned}$$

$T(i, j)$ denotes the time/iteration at which pixel (i, j) is scheduled for processing. Thus for a truly optimal scheduling, each $T(i, j)$ should be one unit greater than maximum of its dependencies, viz, $T(i-1, j-1), T(i-1, j), T(i-1, j+1), T(i, j-1)$. Optimal scheduling constraint results in an ordering pattern given in Figure 3. This pattern is the source of available parallelism, as the label indicates the iteration in which the pixel can be scheduled. All pixels with label n can be processed in parallel and independent of each other, provided that all pixels with label less than n have been processed. The number of pixels that can be processed in parallel increases as we approach the diagonal of the image. The pixels that can be processed in parallel are arranged in a knight's move order in chess. The maximum parallelism for an $M \times N$ image is $\min\{M, N/2\}$.

Algorithm 1 Basic Floyd-Steinberg Dithering

```
I = Input Image
OutputImage = $null$
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        OutputImage[i,j] = $NearestColor$(I[i,j])
        err = I[i,j] - OutputImage[i,j]
        I[i,j+1] += err*(7/16)
        I[i+1,j-1] += err*(3/16)
        I[i+1,j] += err*(5/16)
        I[i+1,j+1] += err*(1/16)
    end for
end for
```

Refer to Algorithm 2 for our implementation of $NearestColor$

Algorithm 2 Threshold(value, min, max, levels)

```
min = $min$(input color range)
max = $max$(input color range)
levels = output color levels
value = color value to be threshold
intervalLen = (max - min)/(levels - 1)
thresholdInd = $round$((value - min)/intervalLen))
thresholdVal = min + thresholdInd*intervalLen
return thresholdVal
```

Where min, max, $round$ have their usual meanings.

3.2. Block-based implementation on multicore CPUs

We now describe a block based implementation of FSD algorithm suitable for multicore CPUs with a small number (2-8) of cores. The pixels are grouped together into trapezoidal blocks, each block has a structure shown in Figure 4. The shaded region in the figure shows the pixels that need to have completed their processing before the region ECFG can be processed. Also, due to the pattern of error distribution

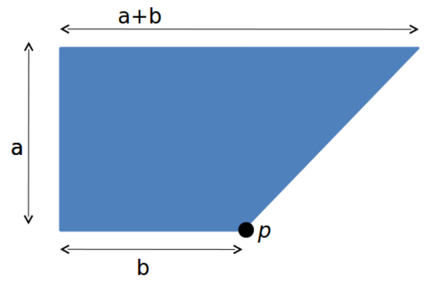


Figure 4. Trapezoidal block structure of a single block (used in Block-based CPU algorithm).

the bottom-right corner pixel (at the vertex F) can be processed only after all the remaining pixels are processed. If the triangle CDE is part of the neighboring trapezoidal block, then parallelogram ECFG defines the current block of pixels which needs to be dithered. (see Figure 5)

The flow of error between such blocks is the same as for the pixels (Figure 2). This induces a knight's move order for parallel scheduling of blocks. Thus, Figure 3 can be thought of as giving the optimal scheduling order for blocks, each of which can be processed by a thread using the sequential FSD approach. For a multicore CPU, the thread creation overhead is high. The optimal scheduling of a certain number of threads is also limited by the number of cores on the CPU. A lot of time may be spent on context switching if the number of threads is high compared to the number of cores. It is better to create a few heavy-weight threads as a result. We use each thread to process one or more blocks, keeping the overall number of threads small. This suits the CPU architecture well and gives good results.

We use OpenMP for thread creation and management on the multicore CPUs. The first occurrence of a block of label n, on a row-wise left to right and top to bottom traversal, is termed as the *primal block*. Hence, by definition, for all blocks with label n, the primal block is the one with the minimum row number. Once the location of primal block is obtained for an iteration, we can get the rows and columns for the other blocks by simply $row = row + k$ and $col = col - 2k$, where k is chosen such that row and col are within the range of the image. The pixels within a block are dithered sequentially in a scan-line manner. Any block requires only the errors from the boundaries of the neighboring blocks (see Figure 5). The pseudo-code for primal block location and block based dithering is given in Algorithm 3.

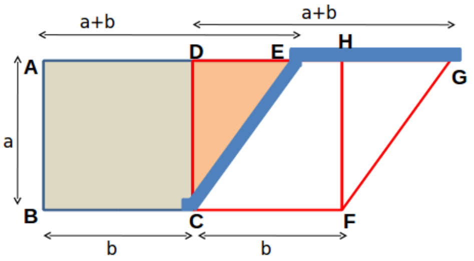


Figure 5. For the block-based CPU algorithm, consider two blocks ABCE and DCFG. The triangle DEC is the overlapping portion between these two blocks. If we have already processed block ABCE, then block DCFG needs to process only the parallelogram ECFG. We just need the error values along EC, EG (blue strips) and one error value along BC (blue block).

Algorithm 3 Primal Block (M,N,a,b,itr)

```
M,N = Image Height, Width
a,b = Block Height, Width
itr = current iteration
if $itr > 0$ and $itr <= N/b$ then
    primalBlock.row = 1, primalBlock.col = itr
else
    primalBlock.row = ceil((itr-(N/b))/2)
    primalBlock.col = (N/b)-[(itr-(N/b))($modulo$ 2)]
end if
```

Algorithm 4 Block-based Floyd-Steinberg Dithering

```
for $i = 1$ to $2 * (M/a) + (N/b)$ do
    get Primal Block (M,N,a,b,$i$)
    omp-set-threads(t)
    pragma omp parallel for
    for $j = 0$ to $totalBlocks$ do
        row = primalBlock.row - j
        col = primalBlock.col - 2*j
        ditherBlock(row,col,I,Out)
    end for
end for
```

3.3. Pixel-based implementation for the GPU

Pixel based implementation of FSD is better suited to the massively parallel architecture of the GPU. Due to availability of larger number of cores, it is best to create a large number of light-weight threads on the GPU. We let each thread process a single pixel of the image. The sequential dependence places strict constraints on how the threads can be scheduled. In iteration k, only as many threads can be actively used as there are pixels with label k (Figure 3).

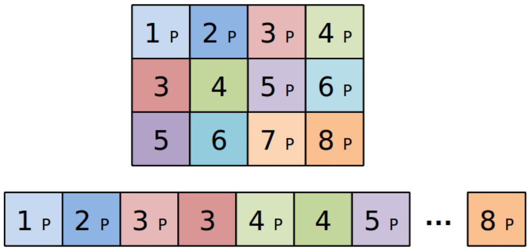


Figure 6. Knight order storage of an image. Here p indicates the $primalblock$.

A GPU kernel operates on the pixel data and the error values from already processed pixels stored in the global memory. The output pixel value and the residual error are written to the global memory. The kernel at each thread reads the residual errors of its neighbors, adds the correct fraction to own pixel value, performs quantization, and writes the output pixel value and residual error. The thread number and the iteration number uniquely determine the pixel (i, j) processed by the thread. However, the reading of error values from the neighbors will be very inefficient if the image is stored in the usual row-major or column-major order due to the uncoalesced memory access pattern. For instance, in Figure 3, consecutive threads process the 3 pixels with label 6 as shown. Their neighbors are not consecutive in memory and the uncoalesced memory access will be very slow.

Optimum memory access times can be achieved by reordering the image. It can be seen that when the left neighbor is accessed by each thread, the memory locations accessed are related by a knight's move pattern, with difference in 1 row and 2 columns. We can reorder the image using the iteration number (labels in Figure 3) and an order within the pixels with the same label. Figure 6 shows the mapping from a 3×4 image to a 12-element 1D representation. This re-ordering can be done effectively on the GPU using shared memory by adapting the method used to transpose a matrix [10]. In practice, since the time required by other steps is significantly larger than this conversion time, we can make use of a simple kernel with MN threads.

After reordering, each thread of each iteration processes its pixel in a similar way. It can be seen that the access to error values from the neighbors as well as access to own pixel value, is totally coalesced in the new order, with consecutive threads always accessing consecutive memory locations. The output pixel and error values are written in the same order also, using coalesced write operations. This maintains efficient memory accesses for subsequent iterations also. The final resulting image needs to be transformed in the reverse using a similar kernel. The overall computation proceeds with each iteration of Figure 3 resulting in a separate invocation of the basic kernel.

4. Using the CPU and the GPU

The previous implementations used the CPU or the GPU alone. Better performance can be obtained by using both these resources. We now discuss two approaches for the same.

4.1. CPU-GPU handover algorithm

The number of pixels that can be processed in parallel (called $parallelism$) is low in the early iterations and in the later iterations. The parallelism slowly increases in the order $1, 1, 2, 2, 3, 3, \cdots$ until the maximum value of $\min\{M, N/2\}$. Depending on the image dimensions, many iterations can have this maximum value. Thereafter, the parallelism reduces slowly in the reverse order to $\cdots, 3, 3, 2, 2, 1, 1$. If the number of threads is low, the amount of time required for kernel setup is more than the actual processing time of the kernel. We therefore let the CPU process the iterations in the beginning and towards the end, before handing over the computations to the GPU (see Figure 7(a)).

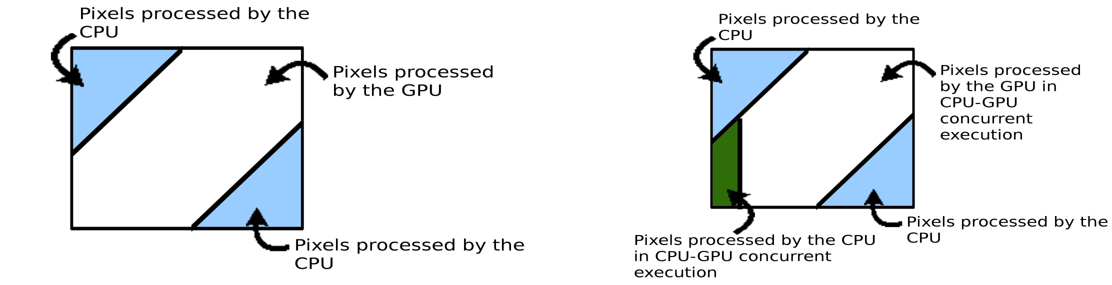
The simple CPU-GPU handover algorithm has the following three stages.

1) Process the initial part (top left of the image) on the CPU until the parallelism exceeds a threshold and then handover the computation to the GPU.
2) Process the image on the GPU until the parallelism falls below the same threshold (towards the bottom-right part of the image) and then handover the computation back to the CPU.
3) Process the end part on the CPU sequentially.

As will be demonstrated by the results later, this algorithm involving a CPU-GPU handover performed better than the algorithm where the GPU alone was used for processing all the iterations. Note that the GPU alone algorithm is a special case of the CPU-GPU handover, with the CPU immediately handing over the computation to the GPU.

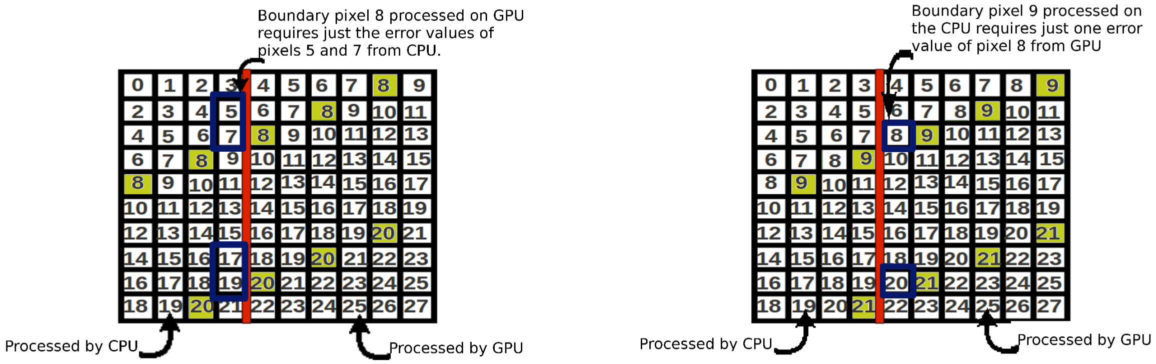
4.2. CPU-GPU hybrid algorithm

In the CPU-GPU handover algorithm (Section 4.1), the CPU and GPU do not operate concurrently. This results in the GPU remaining idle when the CPU does


Pixels processed by the CPU
Pixels processed by the GPU
Pixels processed by the CPU
Pixels processed by the CPU
Pixels processed by the GPU in CPU-GPU concurrent execution
Pixels processed by the CPU in CPU-GPU concurrent execution
Pixels processed by the CPU

(a) CPU-GPU Handover (b) CPU-GPU Hybrid

Figure 7. CPU+GPU Dithering. The pixels processed by the CPU are shown in blue and green, and those processed by the GPU are shown in white.


Boundary pixel 8 processed on GPU requires just the error values of pixels 5 and 7 from CPU.
Processed by CPU
Processed by GPU
Boundary pixel 9 processed on the CPU requires just one error value of pixel 8 from GPU
Processed by CPU
Processed by GPU

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16	17
10	11	12	13	14	15	16	17	18	19
12	13	14	15	16	17	18	19	20	21
14	15	16	17	18	19	20	21	22	23
16	17	18	19	20	21	22	23	24	25
18	19	20	21	22	23	24	25	26	27

(a)

0	1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16	17
10	11	12	13	14	15	16	17	18	19
12	13	14	15	16	17	18	19	20	21
14	15	16	17	18	19	20	21	22	23
16	17	18	19	20	21	22	23	24	25
18	19	20	21	22	23	24	25	26	27

(b)

Figure 8. Errors transferred between the CPU and GPU for hybrid implementation. The pixels to the left of the red line are processed on the CPU and those to the right are processed on the GPU. (a) To process pixels in iteration 8, the GPU needs errors from iterations 5,7 (highlighted by blue border) from the CPU. Similarly for iteration 20, errors from iterations 17 and 19 are needed. (b) To process pixels in iterations 9, the CPU needs errors from iteration 8 (highlighted by blue border) from the GPU. Similarly for iteration 21, errors from iteration 20 are needed.

Algorithm 5 Pixel-based Floyd-Steinberg Dithering

```
for $i = 1$ to $2 * M + N$ do
    get Primal Block (M,N,1,1,$i$)
    //Note we use a=1,b=1 for pixel based approach
    threadBlocks = ceil(totalPixels/MAX-THREAD-PER-BLOCK)
    gridDim(threadBlocks, 1, 1)
    blockDim(MAX-THREAD-PER-BLOCK,1,1)
    ditherKernel(gridDim, blockDim)($i$,primalBlock.row, primalBlock.col,I,Out)
end for
```

the processing and vice versa. On machines like a laptop with a GPU, considerable computing resources are idle at all times. Fast processing of large images needs the mobilization of all resources.

Step 1 and 3 of the handover algorithm is best done on the CPU alone as described above. The step 2 can be jointly performed by CPU and the GPU. We do this by partitioning pixels of each iteration (recollect that pixels of each iteration can be processed totally independently) between the CPU and the GPU, with the CPU processing a fixed number of pixels. After each iteration, along the border separating CPU and GPU execution, a few residual error values need to be sent either from the CPU to the GPU or in the reverse direction. These errors need to be sent before the next iteration begins (Figure 8). The transfer of error values satisfies the data dependency for the next iteration. We use the zero copy feature of Nvidia GPUs for the transfer as the data involved is only a few bytes. This is a feature that enables the GPU threads to directly

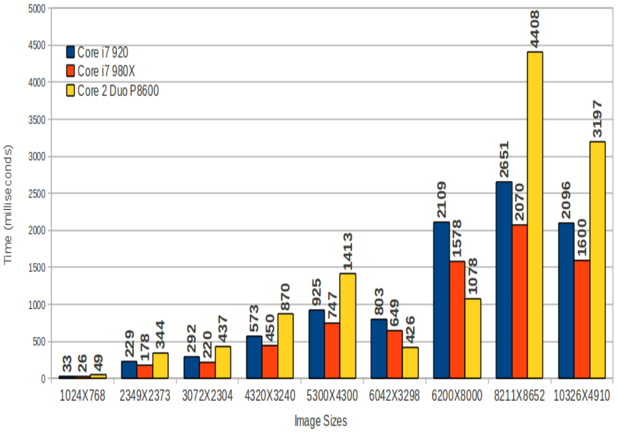


Figure 9. FSD Sequential implementation timings.

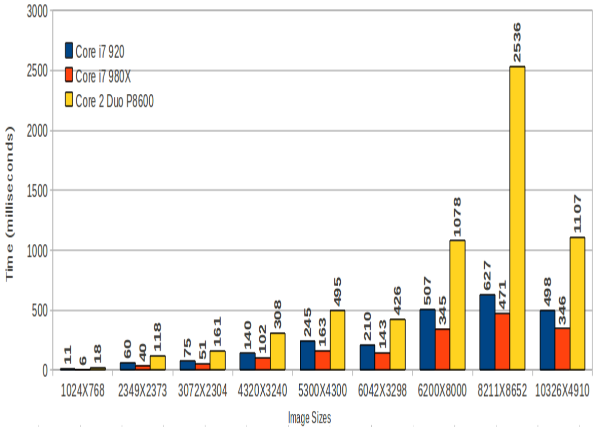


Figure 10. Running times for the block-based implementation on multicore CPUs.

access the pinned memory on the CPU. This feature is better than stream transfer methods like `cudaMemcpy` for transferring small amounts of data [5].

5. Results

We present the the results for the different methods discussed above. The focus is on real-time processing of large images on ordinary computers using available computing resources in parallel. We also show results on relatively high-end computing resources such as high-end GPUs and 6-core CPUs to show the efficacy of our methods. The CPUs we use are the Intel Core 2 Duo P8600 (2 physical cores), Intel Core i7 920 (4 physical cores with HyperThreading [3]), Intel Core i7 980x (6 physical cores with HyperThreading). We disabled the TurboBoost feature on the Core i7 processors to ensure a steady clock speed. We use the GPUs Nvidia 8600M GT (32 stream processors), Nvidia GTX 480 (480 stream processors, Fermi architecture), and the Tesla T10 (240 stream processors). The input is an 8-bit gray level image and the output is a binary image. The conversion to knight order (Figure 6) for the 1024×768 and 6042×3298 images takes 4 ms and 99 ms respectively on 8600M. To compare the basic algorithms, we do not include this in the total time.

5.1. Block-based implementation on CPU

We used OpenMP for handling thread creation on multicore CPUs. We varied the block size and the number of threads independently of each other and checked timings for the combinations. A strong dependence between the number of cores in the CPU and the maximum number of threads used is seen, and hence we can assume the timings to be independent of the block size. For a given image, we calculate the block size from the number of threads. e.g. For n threads, we use the block size which will have n blocks in parallel for the maximum number of iterations. We are able to get a 3-4 times speed-up over the purely sequential implementation. The number of threads which gives minimum timing is roughly 1.5 to 2 times the number of CPU cores for CPUs with Intel HyperThreading Technology [3], and exactly equal to the number of cores for other CPUs. By using this observation, we can compute the optimal number of threads needed for a certain CPU (depending on number of cores) and from this we can calculate the block size for any image automatically. Consider the 1024×768 image (Figure 9 and Figure 10). For the Core 2 Duo P8600, we see that the speed-up is 2 times compared to the sequential code, when 2 threads are used. For the Core i7 980x, the speed-up is around 4 times (with 9 threads).

5.2. Pixel-based implementation on GPU alone

As noted earlier, the pixel based implementation on the GPU alone is a special case of CPU-GPU handover algorithm. The time required for 1024×768 image on 8600M is about 48 milliseconds (ms). Similarly for a 6042×3298 image, we need about 576 ms on the 8600M. These values are higher than the corresponding values needed for a CPU-GPU handover. In fact, as discussed later, we observe that to improve performance some of the initial and final iterations (with low parallelism) must be offloaded to the CPU.

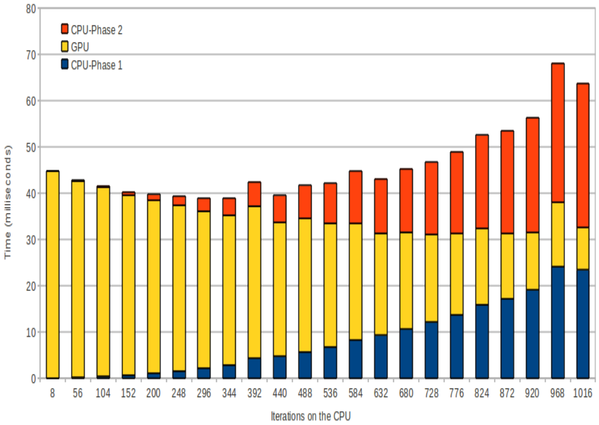


Figure 11. CPU-GPU handover algorithm times for different number of pixels processd by the CPU for a 1024×768 image on a Nvidia 8600M.

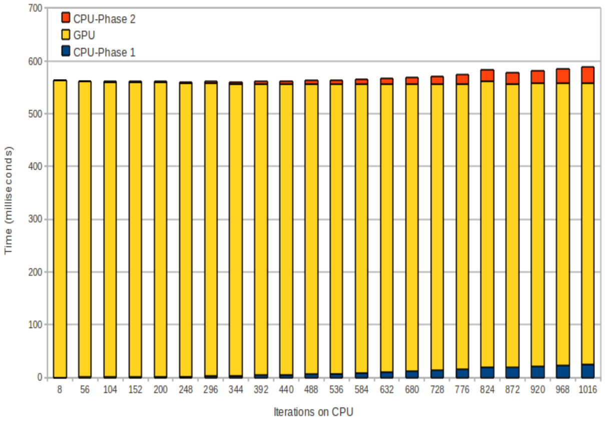


Figure 12. CPU-GPU handover algorithm times for different number of pixels processd by the CPU for a 6042×3298 image on a Nvidia 8600M.

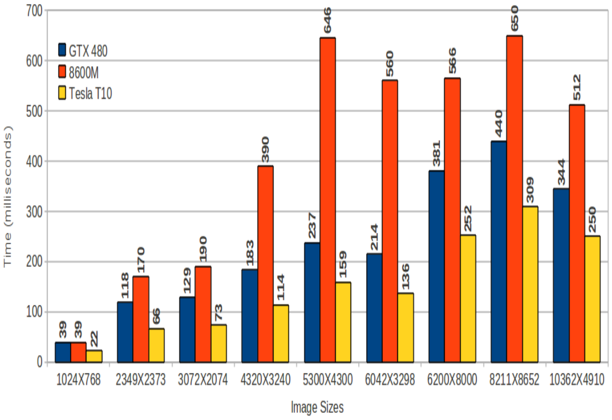


Figure 13. Best timings for CPU-GPU handover algorithm on different images and GPUs.

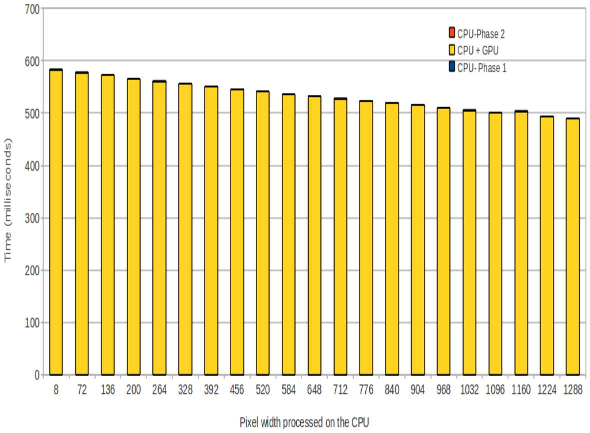


Figure 14. CPU-GPU hybrid algorithm times for different number of pixels processd by the CPU for an 6042×3298 image on a Nvidia 8600M.

5.3. CPU-GPU handover

The CPU-GPU handover algorithm uses the GPU resources well. We vary the number of iterations handled by the CPU and GPU. In essence we modify the parameter iteration number n, which signifies the number of iterations after which the CPU hands over control to the GPU, and the number of iterations the CPU handles towards the end. So an iteration value of n means that the CPU handles n iterations at the beginning, n iterations at the end and the GPU handles the iterations in between. Figures 11 and 12 have plots of time v/s iteration number. Figure 13 shows the optimum time required for various image sizes on various GPUs. The nature of the graphs 11 and 12 indicates that there is an optimum value of the number of iterations which should be performed on the CPU. Increasing or decreasing this value, results in an increase in total time. Also as seen in graph 13, the time required to dither various images is the lowest for the Tesla T10. These timings are considerably lower than the sequential timings (Figure 9).

5.4. CPU-GPU hybrid

The hybrid algorithm uses both the CPU and the GPU during the step 2 of the processing. We vary the number of pixels handled by the CPU in this step. This changes the boundary for the CPU-GPU concurrent execution part of the hybrid algorithm. Figures 14 and 15) give the times for two different images for different amounts of load handled by the CPU. The results demonstrate the existence of an optimum value for pixel width processed by CPU that reduces the total

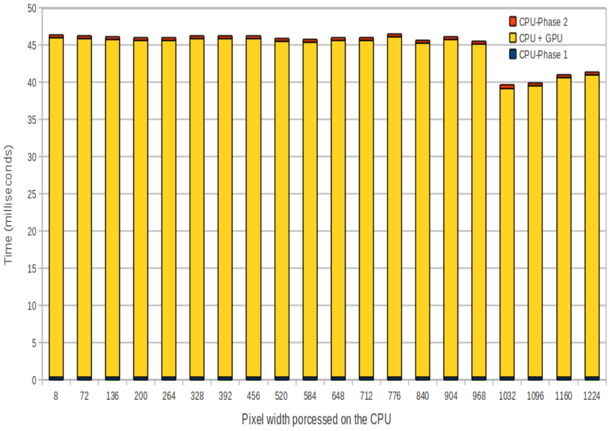


Figure 15. CPU-GPU hybrid algorithm times for different number of pixels processd by the CPU for an 1024×2048 image on a Nvidia 8600M.

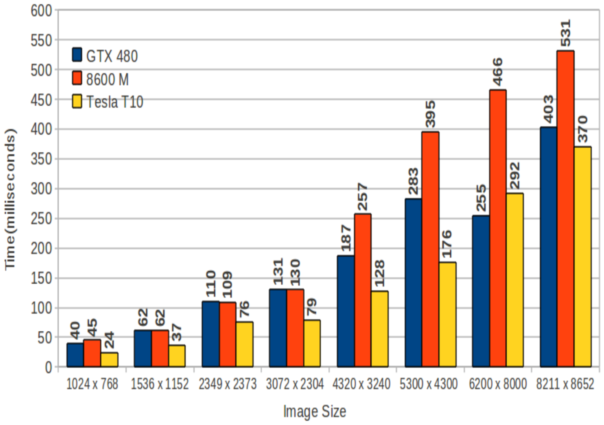


Figure 16. Best timings for hybrid algorithm.

overall processing time. As seen from the graph in Figure 15, the hybrid approach is not beneficial for small image sizes. This is because the time required for synchronization (zero-copy memory access time and kernel setup time) is much higher in this case as compared to the actual time for computation. The overall timings for various images on the different GPUs are shown in Figure 16.

5.5. Discussion

A summary of the timings for all our approaches is presented in Figure 17. The CPU-GPU handover and CPU-GPU hybrid approaches perform better than the GPU alone approach, as we are able to utilize all available computational resources of the CPU and the GPU. On high-end hardware, our speed up is around 10 and on low-end hardware, after exhaustive utilization of resources, the speed up is 3-4, compared to a standard sequential implementation, shown in Figure 9. Our approaches involving the GPU work better for large images, since the kernel set-up time, the memory copy time, etc., become overheads for small images. Although it may seem that in some cases, the block-based CPU algorithm performs better than the CPU-GPU handover or the CPU-GPU hybrid algorithm, it must be rememembered that the multicore CPUs used in the block-based CPU results are relatively high end CPUs (with 4-6 cores), while the GPU as well as the CPU in the CPU-GPU handover or CPU-GPU hybrid results is a commodity, low-end hardware.

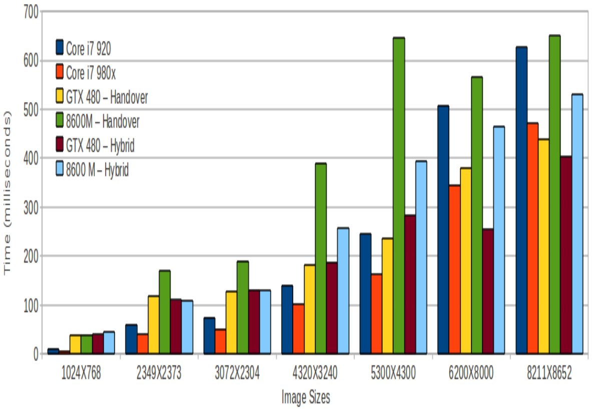


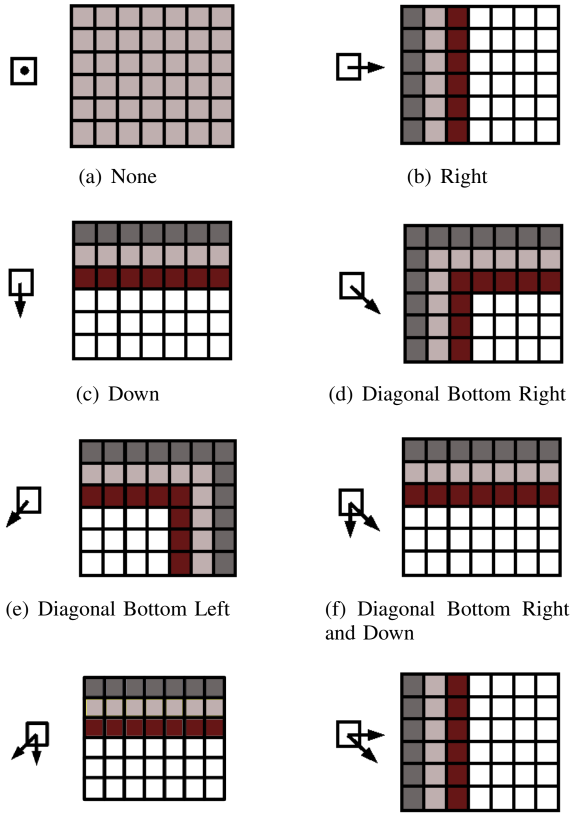
Figure 17. Summary of the best timings across all our implementations.

6. Conclusions

We demonstrated that problems with long sequential dependency, like Floyd Steinberg Dithering, can be efficiently implemented in parallel even on low-end hardware. We can handle various types of data dependency by just analyzing the pixels that can be processed in parallel in a given iteration.

The non-linear dependence on the outputs of some past pixels, on the computation of each pixel reduces parallelism greatly. We can study pixel independence as a function of data dependence, generalizing on the dependence pattern of FSD. If a pixel does not depend on any of its neighbors (Figure 18(a)), all pixels can be processed independently, similar to the case where input and output are two different copies of the image.

When a pixel sends its error to its right neighbor, each pixel depends on the output of its left neighbor. In such a case, the pixels of each column are independent, but each column is dependent on its previous one. Thus, all M pixels of each column can be processed


(a) None

(b) Right

(c) Down

(d) Diagonal Bottom Right

(e) Diagonal Bottom Left

(f) Diagonal Bottom Right and Down

(g) Diagonal Bottom Left and Down

(h) Diagonal Bottom Right and Right

Figure 18. Various types of data dependences and how they can be resolved in parallel. Pixels in same color indicate that they can be processed in parallel in a single iteration.

in parallel and hence the available parallelism is M (constant), as seen in Figure 18(b). Similarly, when a pixel depends on its upper neighbor (i.e., error is sent downwards), the available parallelism is N (constant), the number of columns (Figure 18(c)). When a pixel depends on its top-left neighbor, the first row and column can be processed in parallel in the first step, the next row and column in the second step, etc. Thus, the usable parallelism is $M + N, M + N - 2$, etc., with average parallelism of $(M + N)/2$. If a pixel depends on the top and top-left neighbors, the available parallelism is N (Figure 18(g)).

The parallelism available is N if a pixel depends on the top-right neighbor (Or $M + N, M + N - 2$, etc., if the top-right pixel is the earliest). If a pixel depends on its left and right neighbors, no solution is possible, as there are no causal ordering. If a pixel depends on the top-right and left neighbors, the pixels that can be computed together in each step is shown in Figure 6. This figure also shows the reordering of the pixels which will result in coalesced memory access. Each parallel execution structure given in Figure 18 corresponds to a pixel reordering. For row-parallelism of Figures 18(c), 18(f) and 18(g), row-major ordering of pixels will suffice. For column-parallelism (Figures 18(b) and 18(h)), a column-major order is necessary. A similar reording can be associated with each of the figures.

Thus we can use this strategy to parallelize problems with any kinds of non-linear data dependency by reordering the data and exploiting the parallelism available in these computations.

References

[1] Bruce Waltery, George Drettakis, Steven Parker, *Interactive rendering using the render cache*. Eurographics Workshop on Rendering, 1999.

[2] Chang Chou Lin and Wen-Hsang Tsai, *Visual crytpography for gray-level images by dithering techniques*. Pattern Recognition Letters, Vol 24.

[3] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, Michael Upton, *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, Vol 6.

[4] Metaxas, P. T., *Optimal parallel error-diffusion dithering*. Proceedings of SPIE, 1999.

[5] NVIDIA CUDA Best Practices Guide 3.2

[6] P Li and J P Allebach, *Block interlaced pinwheel error diffusion*. Journal of Electronic Imaging, 2005.

[7] Pavel Slavik and Jan Prikryl, *Dithering as a method for image data compression*. Winter School of Computer Graphics, 1995.

[8] R Floyd and L Steinberg *An adaptive algorithm for spatial grey scale*. Digest of the Society of Information Display, 1976.

[9] Sam Hocevar and Gary Niger, *Reinstating Floyd-Steinberg: Improved Metrics for Quality Assessment of Error Diffusion Algorithms*. Proceedings of the International Conference on Image and Signal Processing, (ICISP) 2008.

[10] Greg Ruetsch and Paulius Micikevicius, *Optimizing Matrix Transpose in CUDA*. NVIDIA CUDA SDK Application Note, 2009

[11] Yao Zhang, John Ludd Recker, Robert Ulichney, Giordano B. Beretta, Ingeborg Tastl, I-Jong Lin, John D. Owens, *A parallel error-diffusion implementation on a GPU*. Proceedings of SPIE, 2011.

[12] Y Zhang, *Line diffusion: a parallel error diffusion algorithm for digital halftoning*. The Visual Computer, Vol 12.