CUDA based implementation of DCT/IDCT on GPU

Team member: Peng Ye
Xiaoyan Shi
Instructor: Xiaoming Li
Contents

A. Introduction to DCT/IDCT
B. Direct matrix multiplication algorithm implementation
   1. Implementation on CPU
   2. Implementation on GPU
C. FFT based algorithm implementation
   1. Algorithm description
   2. Implementation on CPU
   3. Implementation on GPU
D. Conclusion
A. Introduction to DCT/IDCT

Discrete cosine transform (DCT) and inversion cosine transform (IDCT) are well-known signal processing tools that are widely used in compression standards because of their compact representation power. During compression stage, the image is divided into N by N blocks, 2-D DCT can be obtained by first performing 1-D DCT of the columns in the matrix, let’s name it as temp(in code), followed by 1-D DCT of the rows in the calculated matrix in the first 1-D DCT as below figure show:

![Diagram of 2D DCT computational model](image)

Fig. 1 2D DCT compute model [2]

The mathematical model and related coefficients used in our implementation are listed below:

\[
F(u, v) = c(u)c(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \left( \frac{(2x + 1) \pi u}{2N} \right) \cos \left( \frac{(2y + 1) \pi v}{2N} \right)
\]

\[
f(x, y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F(u, v) c(u)c(v) \cos \left( \frac{(2x + 1) \pi u}{2N} \right) \cos \left( \frac{(2y + 1) \pi v}{2N} \right)
\]

where

\[
c(u) = \frac{1}{\sqrt{N}} \text{ for } u, v = 0
\]

\[
c(u) = \frac{2}{\sqrt{N}} \text{ for } u, v = 1 \text{ through } N - 1
\]
For the purpose of image compression, we quarter small value coefficients in DCT domain to zero. We only keep a small number of coefficients, which contain most information of the image. Then we could recover the image using these coefficients. The whole compression is shown below. We do this simulation on MATLAB here.

Original picture

Transferred pixel matrix 1

Transferred pixel matrix 2

Recovered picture (compression rate: $64^2 / 256^2$)

Fig.2 Image compression example
B. Direct matrix multiplication algorithm implementation

1. Implementation on CPU

In order to demonstrate the efficiency of algorithm we have implemented on GPU, we have to establish the basis for comparison. Our first job in this project is to finish C program on CPU and optimization based serial processing that we have learned from lectures.

**Core code** (not have to be square matrix):

```c
for (n=C; n<N; n++)
    for (m=0; m<N; m++)
        for (x=0; x<N; x++)
            dTemp[m*N+n] += E[x*N+n] * coeffs[m] * cos((2*x+1)*PI*n/(2*N));

for (m=C; m<N; m++)
    for (n=0; n<N; n++)
        for (x=0; x<N; x++)
            f[m*N+n] += dTemp[m*N+x] * coeffs[n] * cos((2*x+1)*PI*n/(2*N));
```

Notice that coefficient matrix contains 2*N elements but spread into N*N space. Besides, cosine function in C is very computationally expensive. Therefore, the way to improve the performance is to pre-compute the coefficient and read them during DCT/IDCT algorithm.

**Performance:**

<table>
<thead>
<tr>
<th>Dimension of matrix</th>
<th>64*64</th>
<th>128*128</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for DMM algorithm on CPU (s)</td>
<td>0.031</td>
<td>0.266</td>
<td>2.156</td>
<td>23.781</td>
<td>1121.079</td>
<td>1979.625</td>
</tr>
</tbody>
</table>

Table 1 DMM algorithm on CPU

2. Implementation on GPU

GPU has parallel channels, which would enable us to run different sets of input on different channels and thus avoid wasting transistors on non-computing task.

In Matrix multiplications, the same program is executed on many data elements, which makes parallel computing become a good candidate to help to improve the efficiency. There are many successful applications using a data-parallel programming model to speed up the computation
of large data sets such as matrix. And as a streaming processor, the GPU has some features which are especially good to address problems for data-parallel computations.

Thus, we choose direct matrix multiplication (DMM) (Fig. 3) as our main algorithm to perform DCT/IDCT. In our project, we use GPU to solve problem in scientific computing (matrix multiplication) instead of rendering images. Since CPU part of project has demonstrated how we use MATLAB to get pixel matrix and recover the image from resulting matrix, so the GPU part of project would focus computation itself.

2.1 Computational model mapping

![Computational mapping model](image)

The computational model is showed above. The blocks are spreaded to cover the whole problem size. Two basic rules: each thread block is responsible for computing one square sub-matrix result matrix (square in green box); each thread within the block is responsible for computing one element.

Block size is set to 16, the multiple of warp size and share memory bank number, which would avoid bank conflict when accessing data. The threads in each block would be 256 that is within the range allowed for each block (64,512).

Blue blocks show how the computation is carried out. The corresponding elements in original pixel matrix and transfer coefficients matrix are highlighted with different colors. For the second column transferring, matrixes would switch places but the algorithm stays same. The
registers would accumulate the value from each multiplication till the loop finishes the width of original matrix.

2.2 Memory model and usage

Each element is loaded from global memory to share memory. Since block size is set to 16, so the thread index would be the index number of banks. After finishing computation, the data in share memory y would be loaded back to global memory.

In order to adjust model to the picture size whose pixel matrix would not be the multiple of 16, we used the padding location to lead to aligned memory accesses. As showed in Fig.4 below:

![Memory padding](image)

Fig. 4 Memory padding

The problem size is covered by texture and the padding space is left in light blue. We set the elements in this region to zero which would not affect final results because we take the elements within problem size.

2.3 Performance Analysis
Fig. 5 DMM algorithm performance comparison: GPU vs CPU

<table>
<thead>
<tr>
<th>Dimension of matrix</th>
<th>64*64</th>
<th>128*128</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for DMM algorithm on CPU (s)</td>
<td>0.031</td>
<td>0.266</td>
<td>2.156</td>
<td>23.781</td>
<td>1121.079</td>
<td>1979.625</td>
</tr>
<tr>
<td>Time for DMM algorithm on GPU (s)</td>
<td>0.0023</td>
<td>0.0032</td>
<td>0.0079</td>
<td>0.0325</td>
<td>0.2</td>
<td>1.19</td>
</tr>
<tr>
<td>Speed Up</td>
<td>13.4783</td>
<td>83.1250</td>
<td>272.9114</td>
<td>731.7231</td>
<td>1216.2</td>
<td>1663.6</td>
</tr>
</tbody>
</table>

Table 2 DMM algorithm performance comparison: GPU vs CPU

Fig. 6 Speedup of DMM algorithm: GPU vs CPU
The time consumed by CPU program increases “exponentially” as the dimension of matrix goes up, while GPU uses much less time especially for large matrix (see the last row of speedup). We also noticed for small size of matrix, CPU works better. This will not undermine our efforts because the only large matrix computation needs GPU to speed up while the application only involves small size of picture does not have to go through all the trouble of GPU. In order to understand this problem better, we examine the efficiency of different parts of code. The major parts could be divided into coefficient generation (kernel: Cof_generator()) and matrix multiplication (kernel: mul()). Below cylinder diagram showed the time consumed by these two parts separately.

Furthermore, we found that the time for coefficient generator increases with dimensions slowly, basically keeps the same amount time. While matrix multiplication takes more time for large matrix. So optimization discussed below would be addressed these two parts. For small matrix, the optimization on coefficient generator routine makes more sense, while optimization for large matrix would make a difference on performance.

2.4 Optimization of coefficient generator

Only 2*N different cosine elements involved, we can use the same optimization in CPU part to avoid repeating calculation.

\[
Ds[ty][tx] = \sqrt{((2/N) + 1) * \cos((2*Py + 1) * Px * Pi / (2*N))};
\]

Therefore, the complexity of problem is reduced: \(O(N*N)\)–\(O(N)\).

As we learned from class, using constant memory for parameters used frequently in code would improve the performance.

\[
Ds[ty][tx] = \sqrt{((2/N))} * \text{myConstantArray}((2*Py + 1) * Px * Pi / (4*N));
\]

2.5 Optimization of matrix multiplication

At first stage, we use the clear implementation to make sure code runs correctly. After checking results matrix with those of CPU, we can move to next step to make it more efficient.

Fortunately, the last few lectures introduced some techniques that can apply to our problems.

a) Pre-fetch data

To avoid reading address of matrix, registers are used to speed up the access of data. The ideas are illustrated in below code part (only the related part).

\[
\text{float Atemp = A[a + wA1 * ty + tx];}
\]
float Btemp = B[b + wB1 * ty + tx];

// Shared memory for the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Shared memory for the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Load the matrices from global memory to shared memory;
// each thread loads one element of each matrix
As[ty][tx] = Atemp;
Bs[ty][tx] = Btemp;

b) Tile matrix

Access two matrixes at the same time because the step is block size. The idea is illustrated in below fig and sample code.

Csub += As[ty][0] * Bs[0][tx];
Dsub += As[ty][0] * Bs[0][tx+16];

c) Unrolling

This is a straightforward optimization to reduce the branches.

2.6 Performance Enhancement and Conclusion
The comparison of the direct matrix multiplication and optimized version is shown above.

- The optimization for generating cosine coefficients did not contribute much
- The optimization technique used for matrix multiplication significantly reduced computation time than previous version, especially for large matrix (highlighted in red).

Conclusion:

- Using appropriate compute and memory model to implement matrix multiplication on GPU will greatly speed up computation.
- To make the better use of computation resources, the optimization techniques are used, and the enhanced performances are demonstrated in the comparison table. It shows that the bits of techniques are worth of learning because they would contribute to better performance with same resources.
C. FFT-based DCT implementation

1. Algorithm description

1.1 The relationship between DCT and FFT

Firstly, let’s see the following equations. That’s the base of our algorithm.

**N-point DCT:**

\[
F(u) = c(u) \sum_{x=0}^{N-1} f(x) \cos \frac{(2x + 1)u\pi}{2N}
\]

\[
= c(u) \sum_{x=0}^{N-1} f(x) \left( \cos \frac{x\pi u}{N} \cos \frac{u\pi}{2N} - \sin \frac{x\pi u}{N} \sin \frac{u\pi}{2N} \right)
\]

\[
= c(u) \left[ \cos \frac{u\pi}{2N} \sum_{x=0}^{N-1} f(x) \cos \frac{x\pi u}{N} - \sin \frac{u\pi}{2N} \sum_{x=0}^{N-1} f(x) \sin \frac{x\pi u}{N} \right]
\]

**2N-Point FFT:**

\[
DFT[f(x)] = F(k) = \sum_{x=0}^{2N-1} f(x) \left( \cos \frac{\pi xk}{N} - j \sin \frac{\pi xk}{N} \right) (2N-point DFT)
\]

Comparing the above two equations, we get:

\[
F(u) = c(u) \cos \frac{u\pi}{2N} F(k)_{re} + c(u)\sin \frac{u\pi}{2N} F(k)_{im}
\]

From the above equation, we found that we could compute N-point DCT through 2N-point DFT. And for the computation, there is a famous fast algorithm—Fast Fourier Transform (FFT). As for the 2D DCT, based on the same method we have mentioned in the direct multiplication part, we just do the 1D DCT along the row direction and the column direction respectively. After analyzing the computational complexity of the Complex Multiplication for DFT, FFT, DCT and FFT-based DCT, we found that FFT-based algorithm could bring us much better performance than DMM-based algorithm on CPU. The analysis of the complexity is shown below (notice that the analysis is based on GPU, we haven’t considered parallelism here):
1) DFT versus FFT algorithm

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>Complex Multiplication in Original DFT</th>
<th>Complex Multiplication in FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>(N^*N)</td>
<td>((N/2)^*\log(N))</td>
</tr>
</tbody>
</table>

Table 3

2) Direct Matrix Multiplication based DCT versus FFT based DCT

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>Complex Multiplication in DMM based DCT</th>
<th>Complex Multiplication in FFT based DCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>(N^*N^*N)</td>
<td>(2N^<em>N^</em>\log(2N))</td>
</tr>
</tbody>
</table>

Table 4

Notice that the fast algorithm isn’t really fast when \(N\) is very small.

1.2 Introduction to FFT [1]

In the field of digital signal processing, Discrete Fourier Transform plays an important role in many applications. A major reason for its importance is the existence of efficient algorithms for computing the DFT. There are two major computationally efficient algorithms for evaluating DFT. We will discuss FFT here. It is a divide-and-conquer approach in which a DFT of size \(N\), where \(N\) is a composite number, is reduced to the computation of smaller DFTs from which the larger DFT is computed. To illustrate the basic notation, we will consider the computation of an \(N\)-point DFT, where \(N\) can be factored as a product of two integers, that is,

\[N = LM\]

Recall that:

\[X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}\]

\((0 \leq k \leq N - 1)\)

Suppose we select the column-wise mapping for \(x(n)\) and the row-wise mapping for the DFT:

\(n = l + mL\) and \(k = Mp + q\).
Then

\[ X(p, q) = \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} x(l, m)W_N^{(Mp+q)(mL+l)} \]

Furthermore, we get:

\[ X(p, q) = \sum_{l=0}^{L-1} \{ W_N^{iq} \left[ \sum_{m=0}^{M-1} x(l, m)W_M^{mq} \right] \} W_L^{lp} \]

Observing this equation, we found that we could subdivide the computation into three steps:

1. Compute the M-point DFTs

   \[ F(l, q) = \sum_{m=0}^{M-1} x(l, m)W_M^{mq} \]

   \((0 \leq q \leq M - 1)\)

   For each of the rows \(l = 0, 1, \ldots, L-1\).

2. Compute a new rectangular array \(G(l, q)\) defined as

   \[ G(l, q) = W_N^{iq} F(l, q) \]

   \[ \begin{cases} 
   0 \leq l \leq L - 1 \\
   0 \leq q \leq M - 1 
   \end{cases} \]

3. Finally, we compute the L-point DFTs

   \[ X(p, q) = \sum_{l=0}^{L-1} G(l, q)W_L^{lp} \]

For each column \(q = 0, 1, \ldots, M-1\), of the array \(G(l, q)\).

It seems that the computational procedure outlined above is more complex than the direct computation of the DFT. Let analyze the computational complexity then. The first step involves the computation of \(L\) DFTs, each of \(M\) points. Hence this step requires \(L \times M \times M\) complex multiplications and \(LM(M-1)\) complex additions. The second step requires \(LM\) complex multiplications and \(ML(L-1)\) complex additions. So the computational complexity is

Complex multiplications: \(N(M+L+1)\)

Complex additions: \(N(M+L-2)\)

Where \(N = ML\). Thus the number of multiplication has been reduced from \(N^2\) to \(N(M+L-1)\) and the number of additions has been reduced from \(N(N-1)\) to \(N(M+L-2)\). For example, suppose \(N = 1000\) and we select \(L = 2\) and \(M = 500\). This instead of \(10^6\) complex multiplications via direct computation of the DFT, this approach leads to 503,000 complex multiplications.
When \( N \) is a highly composite number, that is, \( N \) can be factored into product of prime numbers of the form \( N = r_1 r_2 \cdots r_v \), then the decomposition above can be repeated \((v-1)\) more times. This procedure results in smaller DFTs, which, in turn, leads to a more efficient computational algorithm. What we are using in our project is a method, namely, radix-2 FFT, where \( N = 2^v \). The diagram below shows how we compute a 8-point using this radix-2 FFT algorithm.

![Diagram](image)

**Fig. 8** Eight-point radix2 FFT

### 2. Implementation on CPU

We won’t talk a lot about the implementation. Just show our core code below:

**Core code for FFT:**

```c
for (k=0;k<k;++k) {
    for (i=0;i<k;i++)
        { 
            bsize=1<<(r-k);
            for (i=0;i<bsize/2;i++)
                { 
                    p=j%bsize;
                    b[p]=ADD(a[1+p],a[1+p+bsize/2]);
                    b[p+bsize/2]=SUB(a[1+p],a[1+p+bsize/2]),w[1<<(r-k)];
                }
            }
    x=a;
    a=b;
    b=x;
}
```

```c
for (j=0;j<count;j++)
    { 
        p=0;
        for (i=0;i<r;i++)
            { 
                if (j&1<<i))
                    p+=1<<(r-i-1);
            }
        f[j]=a[p];
    }
```

// adjust the order of the //result
Performance:

<table>
<thead>
<tr>
<th>Dimension of matrix</th>
<th>64*64</th>
<th>128*128</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for FFT based DCT on CPU (s)</td>
<td>0.016</td>
<td>0.078</td>
<td>0.375</td>
<td>1.657</td>
<td>7.375</td>
<td>30.875</td>
</tr>
</tbody>
</table>

Table 5 FFT based DCT Performance on CPU

![Time for FFT based DCT on CPU](image)

**Fig. 9** FFT based DCT performance on CPU

3. Implementation on GPU

3.1 FFT \( (N = 2^r) \)

a. \( N \) – point FFT \((N \leq 256)\)

Using the radix-2 FFT algorithm directly, in our project, the myDCT_FFT1.cu realizes this function. We compute the \( N \)-point FFT within one block.
We associate each point of FFT with a thread in one block. We need double buffer here. For every stage in FFT, the input is As and the output is Bs, because to compute Bs, we need to know the value of more than one element of As. They are not point to point. After finishing the computation of Bs, let As = Bs as the input of next stage. Notice that the GPU code does parallel the two inner-loop of the CPU code. This should bring us better performance for bigger data sequence.
b. N – point FFT (N≥512)

If N≥512, we couldn’t use the above method to compute the FFT, because of the limitation of the maximal thread per block and the limitation of shared memory per block. So we couldn’t use radix-2 FFT algorithm directly. Recall the more general break-and-conquer method we have introduced before. It still works here. For example, if N=512, we will break the data sequence into two part, each part includes 256 points. Firstly, we perform the 256-point FFT using radix-2 FFT algorithm on each part within one block. So we need 2 blocks to finish this respectively.

Fig. 10 FFT algorithm compute model (1)

After we get the output of the two blocks, which are two 256-length data sequences, we compute 2-point DFT for the m-th element of the first sequence and the m-th element of the second sequence. (Notice: we simply compute DFT instead of FFT here). The reason why we just use direct DFT algorithm for the 2-point DFT is that for small data sequence, FFT doesn’t provide better than direct algorithm. And since we only focus on image’s DCT, generally speaking, the size of image is at most 2048*2048, which means we need to perform 4096-point FFT. 4096/256=16, 16 is not that big. So the direct DFT algorithm doesn’t decrease the performance compared with FFT.
3.2 Performance

We could see using the same algorithm for DCT computation the implementation on GPU provides us better performance for matrix with large size (>256*256).

<table>
<thead>
<tr>
<th>Dimension of matrix</th>
<th>64*64</th>
<th>128*128</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for FFT based DCT on CPU (s)</td>
<td>0.016</td>
<td>0.078</td>
<td>0.375</td>
<td>1.657</td>
<td>7.375</td>
<td>30.875</td>
</tr>
<tr>
<td>Time for FFT based DCT on GPU (s)</td>
<td>0.1</td>
<td>0.13</td>
<td>0.43</td>
<td>0.9</td>
<td>2.38</td>
<td>7.93</td>
</tr>
</tbody>
</table>

Table 6 performance comparison of FFT based algorithm: GPU vs CPU
3.3 Optimization

The only optimization we do for this FFT-based algorithm is using constant to store the FFT coefficients instead of download the coefficients in global to shared memory to perform FFT. But the performances of the two methods are almost the same. So we won’t discuss more about this issue.
D. Conclusion

In the project, we implement two algorithms (DDM and FFT based) of DCT on CPU and GPU respectively. In part B, we provide the experimental results to evaluate the performance of DDM algorithm on GPU and CPU. In part C, we provide the experimental results to evaluate the performance of FFT-based algorithm on GPU and CPU. Now we would like to consider the results from part B and part C together to see which approach has the best performance.

<table>
<thead>
<tr>
<th>Dimension of matrix</th>
<th>64*64</th>
<th>128*128</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for DMM algorithm on CPU (s)</td>
<td>0.031</td>
<td>0.266</td>
<td>2.156</td>
<td>23.781</td>
<td>1121.079</td>
<td>1979.625</td>
</tr>
<tr>
<td>Time for DMM algorithm on GPU (s)</td>
<td>0.0023</td>
<td>0.0032</td>
<td>0.0079</td>
<td>0.0325</td>
<td>0.2</td>
<td>1.19</td>
</tr>
<tr>
<td>Time for FFT based DCT on CPU (s)</td>
<td>0.016</td>
<td>0.078</td>
<td>0.375</td>
<td>1.657</td>
<td>7.375</td>
<td>30.875</td>
</tr>
<tr>
<td>Time for FFT based DCT on GPU (s)</td>
<td>0.1</td>
<td>0.13</td>
<td>0.43</td>
<td>0.9</td>
<td>2.38</td>
<td>7.93</td>
</tr>
</tbody>
</table>

Table 7

It is obvious that the DMM method on GPU performs best and the DMM method on CPU performs worst. Performance: DMM on GPU > FFT on GPU > FFT on CPU > DMM on CPU

![Graph](image)

Then we would like to compare the best CPU version (FFT based on CPU) and the best GPU version (DDM on GPU) to see what kind of speedup GPU brings us.
In the end, we would like to talk about the alternative approaches. There are many other fast algorithms for DCT. We only implement the FFT-based fast algorithm in our project. For example, the scaled DCT algorithm [4] is very suitable to be performed on OpenGL based GPU implementation. We may try this algorithm for CUDA based GPU implementation. In our project, we write the code for FFT implementation by ourselves. In fact, there is CUFFT library provided by NVIDIA. We may use the function in CUFFT directly, which should have higher efficiency than our FFT code.
Reference:


[2] “Application Note Discrete Cosine Transform with the LF3320”
