Parallel design of JPEG-LS encoder on graphics processing units

Hao Duan,a Yong Fang,a and Bormin Huangb

aNorthwest A&F University, College of Information Engineering, 22 No Xinong Road, Xianyang, Shaanxi 712100 China
yfang79@gmail.com

bUniversity of Wisconsin-Madison, Space Science and Engineering Center, Madison, Wisconsin 53706

Abstract. With recent technical advances in graphic processing units (GPUs), GPUs have outperformed CPUs in terms of compute capability and memory bandwidth. Many successful GPU applications to high performance computing have been reported. JPEG-LS is an ISO/IEC standard for lossless image compression which utilizes adaptive context modeling and run-length coding to improve compression ratio. However, adaptive context modeling causes data dependency among adjacent pixels and the run-length coding has to be performed in a sequential way. Hence, using JPEG-LS to compress large-volume hyperspectral image data is quite time-consuming. We implement an efficient parallel JPEG-LS encoder for lossless hyperspectral compression on a NVIDIA GPU using the computer unified device architecture (CUDA) programming technology. We use the block parallel strategy, as well as such CUDA techniques as coalesced global memory access, parallel prefix sum, and asynchronous data transfer. We also show the relation between GPU speedup and AVIRIS block size, as well as the relation between compression ratio and AVIRIS block size. When AVIRIS images are divided into blocks, each with 64 x 64 pixels, we gain the best GPU performance with 26.3x speedup over its original CPU code. © 2012 Society of Photo-Optical Instrumentation Engineers (SPIE). [DOI: 10.1117/1.JRS.6.061508]

Keywords: JPEG-LS; computer unified device architecture; graphic processing unit.

1 Introduction

The JPEG-LS is a standard established by ISO/ITU for lossless and near-lossless compression of continuous tone images.1 It is based on the low complexity lossless compression for images (LOCO-I) algorithm.2 LOCO-I algorithm is developed by Hewlett Packard (HP) laboratories, and is characterized by simple adaptive error modeling and the Golomb-Rice coding.3,4 JPEG-LS includes two compression modes (i.e., lossless and near-lossless). For the JPEG-LS near-lossless mode, the compression ratio is lower than the more widely used near-lossless baseline JPEG.5 Thus, JPEG-LS is mainly used for lossless compression. Lossless compression is often applied to remote sensing images, medical images, and professional photography where the integrity of the information is desired.

The airborne visible infrared imaging spectrometer (AVIRIS) is a hyperspectral sensor for remote sensing for Earth surfaces. It delivers calibrated images of the upwelling radiance in 224 contiguous spectral bands with wavelengths from 400 to 2500 nanometers (nm).6 There are 614 pixels per scan line and 512 lines set per scene. Every pixel needs two bytes to store. We choose four scenes from four AVIRIS images and trim them to 512 pixels as a line to test our implementation. Therefore the dimensions of a test image are 512 x 512 x 224. The four AVIRIS images are shown in Fig. 1(a) to 1(d) and can be found in Ref. 7.

In this work, we try to make use of the parallel computing power of GPUs to achieve high performance as compared to its CPU counterpart. GPUs are good at processing a vast amount
of data in parallel. Therefore, dividing the raw data into blocks and encoding them respectively is very applicable to GPU computing.

Several works have been done to parallelize JPEG-LS. Some of them were achieved by hardware implementation. An early hardware implementation was proposed in Ref. 8 where the run mode of the LOCO-I algorithm was ignored to reduce the complexity of implementation and the number of contexts was increased to 767 from the standard 365. This implementation is serial and thus the process speed is not high.

A FPGA hardware implementation with limited pipeline was reported in Ref. 9. When the two neighboring pixels are under different stages of processing and the stages are data independent, these stages can proceed in parallel. The reported performance gain is only 17% on average, compared to the HP LOCO-I software implementation. Two implementations using the fully-pipelined technique were designed to avoid pipeline stalls and attained higher throughput. One used the double-frequency architecture to overcome context dependencies.\(^\text{10}\) Another used a look-ahead technique to compute the prediction residuals.\(^\text{10}\) Both methods can only encode one pixel per each clock cycle. Another parallel pipelined implementation was presented in Ref. 11 where three pipelines were used and a 2x speedup on average was obtained. More pipelines can increase the speedup but require more resources. Thus, it is difficult to achieve a speedup that is proportional with the number of pipelines. Another shortcoming is that no improvement was gained in the run mode because of the sequential nature of this mode. A novel hardware design by delaying the context update procedure was proposed in Ref. 12. This design breaks the data dependencies between sequential pixels, so that a certain degree of parallelism can be guaranteed without reducing compression performance in most cases.

The most recent software implementation based on GPU was proposed in Ref. 13. It takes advantage of the fact that the update of different contexts is independent. Therefore, pixels with different contexts could be processed in parallel. This literature presents a term, i.e., context rounds. Each round includes a set of pixels with different contexts, and these

![Fig. 1 The four AVIRIS images used in our GPU implementation.](image-url)
pixels are processed in parallel. The pixels with the same context are encoded serially in different rounds. After completing one round, the related contexts will be updated and then the next encoding round can start. This parallel software implementation is completely compliant to the JPEG-LS standard, but the degree of parallelism cannot be guaranteed and the reported average speedup is only 0.52 in the case of maximum round capacity, i.e., 366 pixels are scanned per round.

Modern commercial GPUs are far superior to CPUs in terms of processing capability, memory bandwidth, and power consumption. GPUs focus on intensive numerical computing, rather than data caching and flow control. Thus, more compute cores are developed for floating-point calculations. Modern GPUs can be used not only for graphical applications but also for general-purpose high performance computing (HPC). Several interesting HPC applications were reported in Refs. 14 to 17. In this paper, we design a massively parallel version of JPEG-LS to run on GPUs. Because lossless compression is our main purpose, so we only consider the lossless mode of JPEG-LS.

The rest of the paper is structured as follows. Section 2 describes JPEG-LS. Section 3 introduces CUDA. The detailed CUDA implementation of JPEG-LS and the performance results are discussed in Sec. 4. Finally, Sec. 5 summarizes the work and concludes this paper.

2 JPEG-LS for Lossless Compression

JPEG-LS standard consists of two main parts, i.e., context modeling and encoding. For encoding, there are two modes that can be selected. One is the regular mode and the other is the run mode. The simplified procedures of JPEG-LS are illustrated in Fig. 2. A probability distribution used to encode the current sample was gained through the context modeling procedure. The context is classified by three local gradients: \( D_1 = R_d - R_b \), \( D_2 = R_b - R_c \), and \( D_3 = R_c - R_a \). \( R_a \), \( R_b \), \( R_c \), and \( R_d \) are the neighbor pixels of current sample \( I_x \), as shown in Fig. 3. If not all three local gradients are zero (for lossless coding), the regular mode will be selected. In regular mode, the local gradients are mapped onto one of the 365 contexts.
by quantization and merging. A simple median edge detector is used to estimate $P_x$, which is used in latter steps.

$$
P_x = \begin{cases} 
\min(R_a, R_b), & \text{if } R_c \geq \max(R_a, R_b) \\
\max(R_a, R_b), & \text{if } R_c \leq \min(R_a, R_b) \\
R_a + R_b - R_c, & \text{otherwise}
\end{cases}
$$

After $P_x$ is obtained, it is corrected by a context-based correction value $C[Q]$ and adjusted to an appropriate range if necessary. Then, the parameter $k$ of the Golomb-Rice code is computed by using the number of occurrences of this context ($N[Q]$) and the accumulated prediction error magnitudes ($A[Q]$):

$$
k = \left\lfloor \log_2 \frac{A[Q]}{N[Q]} \right\rfloor.
$$

The Golomb-Rice code was designed for coding nonnegative values, so the prediction error $Errval = I_x - P_x$ should be mapped onto a nonnegative value prior to the encoding procedure. The last step is updating the current context’s variables $A[Q], N[Q], B[Q], C[Q]$. The run mode is used to encode the flat regions of the image. In lossless coding, if all local gradients are zero, then the process will enter run mode. The run mode consists of two main steps: run scanning and run-length coding; and run interruption coding. More details about run mode coding can be found in Ref. 1. The pseudo code of the encoding procedure of JPEG-LS is illustrated in Fig. 4. The original C implementation of JPEG-LS is achieved by HP Labs, and the code can be downloaded from Ref. 18. We compiled the C code with -O2 optimization option and tested the four scenes of the AVIRIS images. Table 1 shows the execution time of the original C code for four AVIRIS scenes.

1. initialize $A[Q], N[Q], B[Q], C[Q], Nn[Q], PixelPosition$;
2. for(; ; ){
   if(counter == dim.x * dim.y) break;
}
3. GetNextSample();
4. Compute $D_1, D_2, D_3$;
5. if($D_1 == 0 \&\& D_2 == 0 \&\& D_3 == 0$){
   RunModeProcessing();
   RunlengthScanning();
   RunlengthCoding();
   Run INTERRUPTION Coding();
   Update $A[Q], N[Q], Nn[Q]$;
}
6. else {
   RegularModeProcessing();
   ContextClassification();
   PredictionErrorComputation();
   LimitedGolomb_RiceCoding();
   Update $A[Q], N[Q], B[Q], C[Q]$;
}

**Fig. 4** Pseudo code of encoding procedure of JPEG-LS.
3 Compute Unified Device Architecture

General purpose computing on GPUs can be realized by CUDA, OpenCL, and DirectCompute. CUDA is NVIDIA’s parallel computing architecture for use in NVIDIA’s GPUs. OpenCL was initially developed by Apple Inc. and now managed by the Khronos Group. The advantage of OpenCL is that the programs written by OpenCL can run across heterogeneous platforms. However, the literature demonstrates that CUDA is more efficient than OpenCL. DirectCompute is developed by Microsoft and only can be used with a Microsoft operating system. We adopt CUDA as our GPU development platform. Before 2007, graphics chips were difficult to use for general purpose computing because of the limited APIs. The release of CUDA makes it easier to develop general-purpose applications. This section introduces the hardware and software architectures of CUDA.

3.1 Hardware Architecture

CUDA-capable GPUs consist of a set of streaming multiprocessors (SMs) and each SM has a number of streaming processors (SPs). GPUs use single instruction, multiple data (SIMD) execution mode. It means that each SP of the same SM executes the same instruction in one clock cycle. For example, the NVIDIA GTX480, which has 15 SMs and each SM has 32 SPs, so there are 480 SPs in total. Each SM can execute one common instruction at a time.

GPU memory systems typically include an off-chip global memory and four different types of on-chip memory, namely, constant cache, texture cache, shared memory, and registers. Global memory can be accessed by all threads of a GPU. Global memory is large but with high latency. Hence, we should reduce the global memory access or use coalesced memory access pattern. Constant and texture cache are read-only. In some situations, using constant and texture cache can reduce the data transfer time. Shared memory can be accessed by all threads in the same thread block. If there is no bank conflict, shared memory access is almost as fast as register access. More details and usage about all kinds of memory have been summarized in Ref. 22.

3.2 GPU Computing with CUDA

CUDA has gone through a rapid development in the past few years. In the CUDA programming model, the GPU operates as a coprocessor of a CPU host. The execution of a typical CUDA program consists of three steps: data transfer from host memory to device memory; kernel execution; and data transfer from device memory to host memory. When a kernel is invoked, a lot of threads are generated and each thread runs a copy of a kernel in parallel. Thread blocks consisting of many threads are executed on SMs concurrently. Each thread block can have no more than 1024 threads in GTX 480. Threads in the same thread block can communicate through shared memory, however, threads in different thread blocks can only exchange data via global memory. Once a thread block is assigned to a SM, it is further divided into 32 threads units known as warps. In fact, a warp is the unit of thread scheduling in SMs. At any time, the SM executes only one of its resident warps. This allows the other warps to wait for long latency operations without slowing down the overall execution throughput of the massive number of execution units.

### Table 1: Execution time of C code.

<table>
<thead>
<tr>
<th>File name</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jasper.raw</td>
<td>8753.6</td>
</tr>
<tr>
<td>moffett.raw</td>
<td>8564.3</td>
</tr>
<tr>
<td>lunar lake.raw</td>
<td>8408.8</td>
</tr>
<tr>
<td>cuprite.raw</td>
<td>8341.6</td>
</tr>
</tbody>
</table>

Duan, Fang, and Huang: Parallel design of JPEG-LS encoder on graphics processing units

In our implementation, we use NVIDIA GTX480 as our hardware platform and CUDA 4.0 as our software platform. The CPU of our desktop computer is Intel Core i7 running at 3.07 GHz.

4 JPEG-LS Encoder on GPUs with Block Parallel Strategy

For an AVIRIS image scene, if we use traditional JPEG-LS encoder, we have to encode the bands one by one. In our parallel design of JPEG-LS encoder, we partition a scene into subsets called blocks and let each thread processes one block. In our early development, we used a two-layer structure. In the higher layer each CUDA block is arranged to process a band of a scene while in the lower layer each thread encodes a block of a band. This structure is showed in Fig. 5.

In general, too small a block size will reduce the compression ratio. As a trade-off between the degree of parallelism and compression ratio, we set $64 \times 64$ pixels as a block. Hence, a band has $8 \times 8$ locks. For the entire 224-band AVIRIS scene we need 224 thread blocks, each with 64 threads. The position of the first pixel of each block should be determined by block ID and thread ID. Lines 6 and 7 of Fig. 6 expresses this relationship. Each thread performs independent JPEG-LS encoding on one block.

![Fig. 5 Illustration of thread arrangement.](image)

```
1. #define height 64 //the height of a tile
2. #define width 64 //the width of a tile
3. int tidx = threadIdx.x;
4. int bidx = blockIdx.x;
5. int firstx ; //the position of the first pixel of a tile
6. firstx = bidx*(height+1)* width + tidx/8 *height*
7. width +tidx%8*width +width+1;
```

![Fig. 6 Partial kernel code for fetching data from global memory.](image)
After the encoding procedure is complete, the coded data blocks are stored in global memory separately. They are compacted prior to being transferred to the host. Before concatenating the scattered coded data blocks, we calculate the starting position of each data block in the final bit stream. The process can be carried out by a method called parallel prefix sum (scan). In this approach, if there are \( N \) starting positions needed to be computed, we need \( O(\log_2 N) \) ops in total. The scan procedure can be described with pseudo code shown in Fig. 7.

After the starting positions are calculated, two methods can be used to finish the compaction of JPEG-LS coded data blocks. One is to iteratively invoke the CUDA runtime API function `cudaMemcpy (device to device)` to compact coded data blocks. When each data block size is large and the number of data blocks is small, the performance of this method is good. But there are some drawbacks when using this method. One is that the size of each coded data blocks must be transferred from GPU device to CPU host before invoking `cudaMemcpy()` which requires additional transfer time. The other is that this method is not suitable when using CUDA stream to hide data transfer time (or kernel execution time). Hence, we adopt another way to compact coded data blocks. The method is to launch a new kernel, which we call `compactionKernel()`. Then we compact data blocks in this kernel. This method is preferable in the situation where there are vast data blocks with a small data block size.

Before invoking `compactionKernel()`, the starting position of every coded data block has been obtained. We just need to copy scattered coded data blocks to a continuous global memory space. We use the coalesced memory access for the copying. For devices with CUDA compute capability 2.0, a cache line has 128 bytes and maps to a 128-byte aligned segment in the device memory. If the size of the words accessed by each thread is 4 bytes and threads in a warp access the data in the same cache line, the accessing request can be coalesced into a one time memory transaction. Hence, we arrange threads in a thread block to read continuous data sequentially from a coded data block and then write them sequentially back into a continuous global memory space. This is illustrated in Fig. 8 and expressed at lines 5 and 6 in Fig. 9. Because the number

---

### Fig. 7 Pseudo code for scan.

```c
1. __shared__ sum[N], temp_sum[N];
2. load the elements to sum;
3. load the elements to temp_sum;
4. __syncthreads();
5. for(i=0;i<N;i=i+2){
6.   if(threadIdx.x >= i){
7.     sum[threadIdx.x] += sum_temp[threadIdx.x - i];
8.   sum_temp[threadIdx.x] = sum_temp[threadIdx.x];
9. __syncthreads();
10. outputsum;
```

---

### Fig. 8 Coalesced global memory access.
of bytes of a data block may not be an integral multiple of the number of threads, we add a special process, as shown at lines 7 and 8 in Fig. 9. If we use one thread block to compact one coded data block, the number of thread blocks will be too many and the workload of a thread block too few. Hence, we arrange one thread block to compact more than one data block (as line 4 in Fig. 9), which can increase the executive efficiency of GPUs.

The performance of this development is shown in Table 2. The speedup is the execution time of the C code in Table 1 divided by the execution time of its CUDA counterpart.

There are multiple memory controllers for memory access in GPU DRAM. In order to make all memory controllers work simultaneously, we make some modifications to keep

![Partial kernel code for compacting the JPEG-LS coded data.](image)

**Table 2** Speedup in early development.

<table>
<thead>
<tr>
<th>File name</th>
<th>Execution time of GPU implementation (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>jasper.raw</td>
<td>620.9</td>
<td>14.1x</td>
</tr>
<tr>
<td>moffett.raw</td>
<td>646.4</td>
<td>13.2x</td>
</tr>
<tr>
<td>lunar lake.raw</td>
<td>645.1</td>
<td>13.0x</td>
</tr>
<tr>
<td>cuprite.raw</td>
<td>573.8</td>
<td>14.5x</td>
</tr>
</tbody>
</table>

![Illustration of the modified thread arrangement.](image)
the load balance among global memory access. In early development, we arrange each thread block to process one band of a scene. But in this modified development, a thread block will handle one block in each band. It means that the $i$th thread of a thread block processes one block of $i$th band. Through this method, we only need 64 thread blocks and the dimension of a thread block increases from 64 to 224. The method is illustrated in Fig. 10 and the key code is expressed at lines 2 and 3 in Fig. 11. Table 3 shows the improved performance of this modification.

Since global memory access has the property of spatial locality, we can use texture cache to improve the bandwidth of global memory access. However, the 224-band AVIRIS scene has $114912 = 224 \times 513$ pixels which is larger than 65536, the maximum allowed linear dimension of 2D texture reference. Thus, we have to divide 224 bands into two parts, and use one texture reference to bind the two parts in order. The procedure of reading pixel values through texture cache is expressed at lines 4 to 8 in Fig. 12.

Figure 13 shows the texture binding and the corresponding kernel invocation. The performance of using texture cache is shown in Table 4. The speedup values are calculated by the execution time of the $C$ code in Table 1 divided by the execution time of the CUDA code in the corresponding scene.

---

**Table 3** Performance of modified development.

<table>
<thead>
<tr>
<th>File name</th>
<th>Execution time of GPU implementation (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>jasper.raw</td>
<td>398.1</td>
<td>22.0x</td>
</tr>
<tr>
<td>moffett.raw</td>
<td>387.8</td>
<td>22.1x</td>
</tr>
<tr>
<td>lunar lake.raw</td>
<td>386.7</td>
<td>21.7x</td>
</tr>
<tr>
<td>cuprite.raw</td>
<td>387.2</td>
<td>21.5x</td>
</tr>
</tbody>
</table>

---

1. Texture<unsigned short, 2> rT; // the declaration of texture reference
2. int idx = threadIdx.x, bidx = blockIdx.x;
3. int coor_x, coor_y; // The coordinate of 1x relative to rT texture reference
4. lx = tex2D(rT, coor_y, coor_x);
5. Ra = tex2D(rT, coor_y -1, coor_x);
6. Rb = tex2D(rT, coor_y, coor_x -1);
7. Rc = tex2D(rT, coor_y -1, coor_x -1);
8. Rd = tex2D(rT, coor_y +1, coor_x -1);
NVIDIA GTX 480 supports asynchronous transfer which allows simultaneous execution of a CUDA kernel and data transfer between CPU host and GPU device. We can use more than one CUDA stream to achieve asynchronous transfer. A CUDA stream is a sequence of GPU operations that get executed in order. GPU operations in different streams can be executed at the same time. To use CUDA stream, we need to use `cudaHostAlloc()` to allocate the pinned CPU memory space and use `cudaMemcpyAsync()` to queue the memory copies. In our work, we make use of this feature to accelerate our CUDA code. We partition 224 bands into three chunks whose volumes are 64, 96, and 64 bands, respectively, and create three streams. First, we copy data chunk 0 to GPU and then invoke a kernel to use JPEG-LS to encode data chunk 0 in stream 0. Then we copy data chunks 1 and 2 to stream 1, and then we invoke a kernel to use JPEG-LS to encode data chunk 1 in stream 1, followed by another kernel to encode data chunk 2 in stream 2. As soon as a kernel finishes JPEG-LS coding in a stream, it is immediately followed by copying the encoded data to the host in the same stream. This procedure is illustrated in Fig. 14 and expressed at lines 2 to 12 in Fig. 15.

![Fig. 13 Texture binding and kernel invocation in the main function.](image)

**Table 4** Speedup improvement: using texture cache.

<table>
<thead>
<tr>
<th>File name</th>
<th>Execution time of GPU implementation (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>jasper.raw</td>
<td>348.5</td>
<td>25.1x</td>
</tr>
<tr>
<td>moffett.raw</td>
<td>334.8</td>
<td>25.6x</td>
</tr>
<tr>
<td>lunar lake.raw</td>
<td>347.6</td>
<td>24.2x</td>
</tr>
<tr>
<td>cuprite.raw</td>
<td>337.2</td>
<td>24.7x</td>
</tr>
</tbody>
</table>

NVIDIA GTX 480 supports asynchronous transfer which allows simultaneous execution of a CUDA kernel and data transfer between CPU host and GPU device. We can use more than one CUDA stream to achieve asynchronous transfer. A CUDA stream is a sequence of GPU operations that get executed in order. GPU operations in different streams can be executed at the same time. To use CUDA stream, we need to use `cudaHostAlloc()` to allocate the pinned CPU memory space and use `cudaMemcpyAsync()` to queue the memory copies. In our work, we make use of this feature to accelerate our CUDA code. We partition 224 bands into three chunks whose volumes are 64, 96, and 64 bands, respectively, and create three streams. First, we copy data chunk 0 to GPU and then invoke a kernel to use JPEG-LS to encode data chunk 0 in stream 0. Then we copy data chunks 1 and 2 to stream 1, and then we invoke a kernel to use JPEG-LS to encode data chunk 1 in stream 1, followed by another kernel to encode data chunk 2 in stream 2. As soon as a kernel finishes JPEG-LS coding in a stream, it is immediately followed by copying the encoded data to the host in the same stream. This procedure is illustrated in Fig. 14 and expressed at lines 2 to 12 in Fig. 15.

![Fig. 14 Execution timeline for the implementation of three CUDA streams.](image)
In our CUDA implementation, the JPEG-LS kernel execution takes more time than host-device data transfer. Hence, asynchronous transfer can hide part of data transfer time. Finally, we obtain our best performance as shown in Table 5.

JPEG-LS applied to different AVIRIS block sizes will lead to different speedup and compression ratios. Figure 16 shows the relation between three different AVIRIS block sizes (128 × 128, 64 × 64, and 32 × 32) and the corresponding compression ratios. As seen, the compression ratio increases with block size.

---

**Fig. 15** The pseudo code of JPEG-LS coding using three CUDA streams.

```plaintext
1. chunk0 = 64; chunk1 = 96; chunk2 = 64; gridDim = 64;
2. Memcpy2D(d_data0, buffer, MemcpyHostToDevice);
3. cudaBindTexture2D(d_data0);
4. encode <<< gridDim,chunk0,0,stream[0] >>(d_data0);
5. cudaMemcpy2DAsync(data1,buffer,MemcpyHostToDevice,stream[1]);
6. cudaBindTexture2D(d_data1);
7. encode <<< gridDim,chunk1,0,stream[1] >>(d_data1);
8. MemcpyAsync(coded0,d_coded0,MemcpyDeviceToHost,stream[0]);
9. cudaBindTexture2D(d_data2);
10. encode <<< gridDim,chunk2,0,stream[2] >>(d_data2);
11. MemcpyAsync(coded1,d_coded1,MemcpyDeviceToHost,stream[1]);
12. MemcpyAsync(coded2,d_coded2,MemcpyDeviceToHost,stream[2]);
```

**Table 5** Final speedup improvement: adding asynchronous transfer.

<table>
<thead>
<tr>
<th>File name</th>
<th>Execution time of GPU implementation (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>jasper.raw</td>
<td>325.6</td>
<td>26.9x</td>
</tr>
<tr>
<td>moffett.raw</td>
<td>332.4</td>
<td>25.8x</td>
</tr>
<tr>
<td>lunar lake.raw</td>
<td>327.3</td>
<td>25.7x</td>
</tr>
<tr>
<td>cuprite.raw</td>
<td>311.1</td>
<td>26.8x</td>
</tr>
</tbody>
</table>

**Fig. 16** The relation between AVIRIS block size and compression ratio.
Figure 17 shows the relation between the three different AVIRIS block sizes and the corresponding GPU speedups. The best average speedup 26.3x is obtained when the AVIRIS block size is $64 \times 64$.

5 Conclusions

In this paper, a GPU-based block-parallel implementation of JPEG-LS encoding is accomplished with the CUDA programming language. However, the bottleneck of JPEG-LS acceleration lies in the concatenation of coded pixels, which has to be done byte by byte due to the fact that the MSB of the current byte must be 0 if the previous byte is 0xFF in each coded data block. The JPEG-LS algorithm is known for its low complexity and good compression ratio. The low complexity makes it easy for hardware implementation. The outstanding compression performance makes it preferable in wide fields. We use the block parallel strategy as well as such CUDA techniques as coalesced global memory access, parallel prefix sum, and asynchronous data transfer. We study the relation between GPU speedup and AVIRIS block size and the relation between compression ratio and AVIRIS block size. Finally, an average speedup 26.3x for four AVIRIS data is obtained when the AVIRIS block size is $64 \times 64$.

Acknowledgments

This research is supported by National Science Foundation of China (NSFC) (Grant Nos. 61001100, 61077009, and 60975007) and Provincial Science Foundation of Shaanxi, China (Grant Nos. 2010K06-15 and 2010JQ8019). We thank the anonymous reviewers for their constructive comments.

References


Biographies and photographs of the authors are not available.