GPUs for Data Parallel Spectral Image Compression

Jarno Mielikainen[a], Risto Honkanen[a], Pekka Toivanen[a] and Bormin Huang [b]

[a]University of Kuopio, Department of Computer Science, P.O.Box 1627, FI-70211 Kuopio, Finland. tel. +358 17163522, fax: +358 17162595, e-mail: mielikai@uku.fi, rhonkane@uku.fi, pjtoivan@uku.fi [b]Space Science and Engineering Center, University of Wisconsin-Madison, bormin@ssec.wisc.edu

ABSTRACT

The amount of data generated by hyper- and ultraspectral imagers is so large that considerable savings in data storage and transmission bandwidth can be achieved using data compression. Due to the large amount of data, the data compression time is of importance. Increasing programmability of commodity Graphics Processing Units (GPUs) allows their usage as General Purpose computation on Graphical Processing Units (GPGPU). GPUs offer potential for considerable increase in computation speed in applications that are data parallel. Data parallel computation on image data executes the same program on many image pixels on parallel. We have implemented a spectral image data compression method called Linear Prediction with Constant Coefficients (LP-CC) using Nvidia's CUDA parallel computing architecture. CUDA is a parallel programming architecture that is designed for data-parallel computation. CUDA does not require the programmers to explicitly manage threads. This simplifies the programming model. Our GPU implementation is experimentally compared to the native CPU implementation. Our speed-up factor was over 30 compared to a single threaded CPU version.

Keywords: ultraspectral sounder data, lossless compression, graphics processing units (GPUs)

1. INTRODUCTION

Hyper- and ultraspectral imagers generate a very large amount of data. Therefore, considerable savings in data storage and transmission bandwidth can be achieved using data compression. Parallel processing enables the use of computationally intensive algorithms. Previously, Message Passing Interface (MPI) and OpenMP have been used for parallel compression of hyperspectral images in.[3] A comparison of clusters and Field-Programmable Gate Arrays (FPGA) for parallel processing of hyperspectral imagery can be found in.[7]

Increasing programmability of commodity Graphics Processing Units (GPUs) allows their usage as General Purpose computation on Graphical Processing Units (GPGPU). GPUs offer potential for considerable increase in computation speed in applications that are data parallel. Data parallel computation on image data executes the same program on many image pixels on parallel. GPUs have been used for wavelet compression in.[4] Lietsch[5] introduced a novel GPU-supported JPEG image compression technique. Simek[6] accelerated 2D wavelet-based medical image compression on MATLAB with CUDA.

Problems that are well suited for GPUs, also have have high arithmetic operation per memory operation ratio.[1] Certain types of hyper- and ultraspectral image data compression methods have both of these properties. Thus, we have implemented a spectral image data compression method called Linear Prediction with Constant Coefficients (LP-CC) using Nvidia's CUDA parallel computing architecture. CUDA is a parallel programming architecture that is designed for data-parallel computation. CUDA is an extension of C (offering an Application Programming Interface(API)) allowing programmers to use a GPU as a massively multi-threaded co-processor. Thus, CUDA hides the GPU hardware from the developers. Moreover, CUDA does not require the programmers to explicitly manage threads. This simplifies the programming model.[2] In this paper, our GPU implementation is experimentally compared to the native CPU implementation.

The rest of the paper is organized as follows. Section 2 describes the linear prediction with constant coefficients method. OpenMP shared memory multiprocessing programming application programming interface (API) is described in section 3. CUDA computing engine for NVIDIA graphics processing units is explained in section 4. Experimental results are given in Section 5. Finally, Section 6 concludes the paper.

Satellite Data Compression, Communication, and Processing V, edited by Bormin Huang, Antonio J. Plaza, Raffaele Vitulli,
Proc. of SPIE Vol. 7455, 74550C · © 2009 SPIE · CCC code: 0277-786X/09/$18 · doi: 10.1117/12.828135

2. LINEAR PREDICTION WITH CONSTANT COEFFICIENTS

The linear prediction with constant coefficients method is described in detain in.[8] Here we present only those details that are required by our computational kernel.

A linear spectral estimate for the current pixel, $p_{x,y,z}$, is formed by considering a linear combination of the corresponding pixels of the previous channels.The estimate can be computed as follows:

$$p'_{x,y,z} = \sum_{i=1}^{min(N,z-1)} \alpha_i p_{x,y,z-i}, \tag{1}$$

where x and y are spatial positions in channel z, α_i are prediction coefficients.For the first N bands the number of prediction coefficients is limited by the number of previous bands. Prediction of band z for the first N bands omits the first $N+1-z$ coefficients. Therefore, the number of prediction coefficients is a minimum of N and $z-1$.

The coefficients are constant in a sense that we only compute the coefficients once and then apply those same coefficients for all the other granules. The optimization procedure is performed on the whole set of granules together. The bandwise prediction coefficients for a set of granules are optimized in least squares sense as follows:

$$MIN_{arg\alpha} \sum_{x,y,g} \left\| p'_{x,y,z,g} - p_{x,y,z,g} \right\|_2 ,. \tag{2}$$

where g is a number of a granule, which belong to a set of base granules. Those linear prediction coefficients are used in the compression of all the other granules. That way, there is no need to store the prediction coefficients as the decoder can recompute the same coefficients based on base granules. The off-line calculation of the optimal prediction coefficients for the set of granules is also performed for the first N bands having a number of previous bands lesser than N.

3. OPENMP

OpenMP is a shared-memory application programming interface (API). OpenMP notation is added to the sequential program to facilitate shared-memory parallel programming. The notation describes work sharing among threads, which execute on different processor cores. Also, the access to shared data is ordered by the notation. OpenMP directive is a pragma to a C/C++ compiler. Thus, the program runs sequentially if a compiler is not OpenMP-aware.[9]

We only used *parallel for* directive in our program. It distributes the iterations of the loop among the executing threads. Those variables that have to be accessed by all threads are declared as *shared*. Variables declared *private* are private to thread. Private variables can be initialized by the value of the variable with the same name using *firstprivate* declaration.

C-language source code for OpenMP linear prediction in compression phase is shown in Figure 1. Compression phase distributes the work of different bands for different threads.

Decompression phase can not process band $z+1$ before band z. Thus, this type of work division is not possible for decompression phase. Therefore, we divide the work row-wise in the decompression phase. Source code for decompression phase is shown in Figure 2.

```
void predict_row_DPCM(short int **image, short int **prediction_error, int width,
                      int bands, CvMat **x, float *mult, float *bias, int predlen){
  int z, band, i, nb;
  float p;
  float elem;

#pragma omp parallel for shared(image,prediction_error)
                         firstprivate(predlen,bias,mult,width,x,bands)
                         private(z,p,elem,nb,i,band)
  for(z=1;z<bands;z++){
    nb = (predlen < z ? predlen : z);
    for(i=0; i < width; i++){
      p=0;
      for(band=1; band<=nb; band++){
        elem = CV_MAT_ELEM(*x[z], float, band-1,0);
        p += elem*(float)image[z-band][i];
      }
      prediction_error[z][i]=(int)p-image[z][i];
    }
  }
}
```

Figure 1. C-language source code for OpenMP linear prediction in the compression phase.

```
#pragma omp parallel for shared(ifp,ofp,img,pred_err)
                         firstprivate(predlen,height,width,bands,mult,bias,x)
                         private(h,w)
for(h=0; h<height; h++)
  for(w=0; w<width; w++)
    predict_row_DPCM2(img[h], pred_err[h], width, bands, x, mult, bias, predlen);
```

Figure 2. C-language source code for OpenMP linear prediction in the decompression phase.

Table 1. Key facts of the NVIDIA GeForce 9600 GT -card.

	GeForce 9600 GT
Number of multiprocessors	8
Number of cores	64
Global memory	512 MB
Shared memory per block	16 kB
Warp size	32

4. CUDA

We used NVIDIA GeForce 9600 GT graphics card as a test platform for our GPU/CUDA experiments. The card consists of 64 scalar processor cores and 512 MB of Graphics Double Data Rate 3 (GDDR3) memory space. The processor cores are divided among eight multiprocessor (MP) each having eight cores, a multi-threaded instruction unit, and 16 kB shared memory space accessible by the cores deployed at the MP. The key facts of the NVIDIA GeForce 9600 GT -card are presented in Table 1. A schematic visualization of the memory configuration of the GeForce 9600 GT is presented on the right hand side of Figure 3. CUDA is an extension

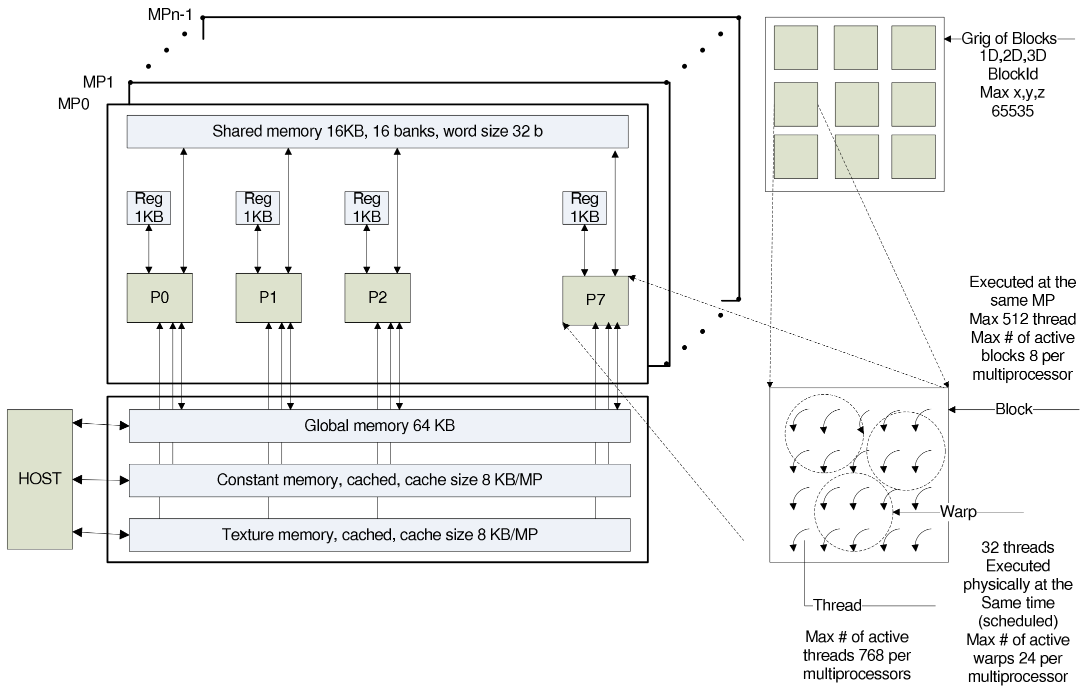
MPn-1
MP1
MP0
Shared memory 16KB, 16 banks, word size 32 b
Reg 1KB
Reg 1KB
Reg 1KB
Reg 1KB
P0
P1
P2
P7
Global memory 64 KB
Constant memory, cached, cache size 8 KB/MP
Texture memory, cached, cache size 8 KB/MP
HOST
Grig of Blocks 1D,2D,3D BlockId Max x,y,z 65535
Executed at the same MP Max 512 thread Max # of active blocks 8 per multiprocessor
Block
Warp
32 threads Executed physically at the Same time (scheduled) Max # of active warps 24 per multiprocessor
Thread
Max # of active threads 768 per multiprocessors

Figure 3. Schematic visualization of a GPU device.

to the C programming language offering programming GPU's directly. A CUDA application consists of two parts: a serial program running on the CPU and a parallel part running on the GPU. The parallel part is called a kernel. A C program program using CUDA extensions distributes a large number of copies of the kernel (denoted threads) into available multiprocessors to be executed simultaneously. A schematic visualization of a distribution of a 2D problem is presented on the right hand side of Figure 3. The problem is divided in a grid of blocks. Each block consists of a number of independently executed threads.

The CUDA code consists of three computational phases: transmission of data into the global memory of GPU, execution of the GPU kernel, and transmission of results from the GPU into the memory of CPU. During the first computational phase the host program receives the 2 dimensional matrices x and $image$ consisting prediction coefficients and image values respectively. Each row i' of the matrix x consists of prediction coefficients for band i'. The maximum number of prediction coefficients is $predlen$ and the total number of bands is $bands$. Thus, the size of x is $predlen \cdot bands$. The matrix $image$ has $bands$ rows each containing $width{\cdot}height$ elements. Each element at the row i' represents the corresponding pixel value of band i' in the original spectral image. The 2D presentations of the matrices are redrawn on the left hand side of Figure 4. Mappings of the matrices are presented on the right hand side of Figure 4.

We allocated linear memory on the device using *cudaMallocPitch()* function. The function pads the allocation to ensure that the alignment requirements for coalescing are fulfilled as the address is updated from row to row.[11] This is made due to optimization of data transmission during the run time of kernel. Thus, each element (i', j') of matrix $\mathtt{x}$ resizes in the memory location $(j' \times predlen + i')$ of the shared memory of GPU and each element

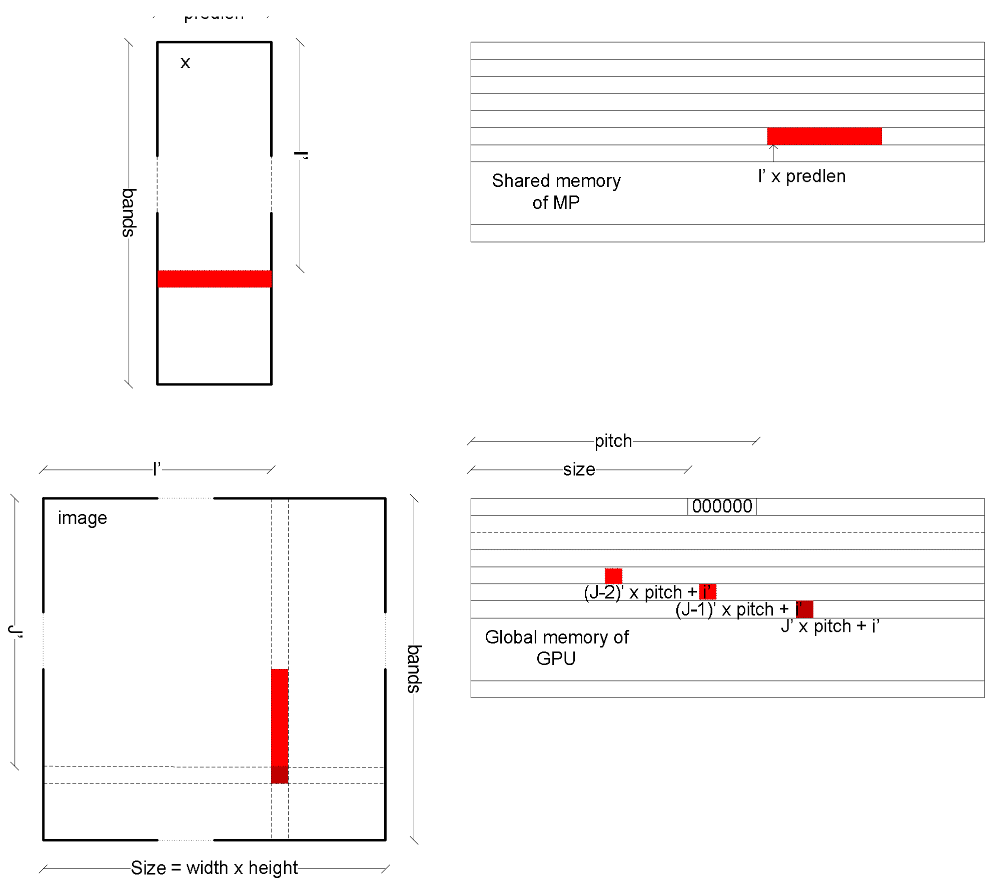

Figure 4. Schematic visualization of the mapping of spectral image into the device memory.

(i', j') of matrix `image` resizes in the memory location $(j' \times pitch + i')$ of the global memory of the GPU.

The CUDA kernel is shown in Figure 5. For illustrations sake we have omitted the exact boundary check. At the beginning of execution, prediction coefficients for the current block are stored in the shared memory as presented at Line 6. In order to perform coalesced memory transfers, values of image pixels and values of prediction errors are transferred from/to memory space eight bytes at a time. The structure implementing that is presented at Lines 7 – 9. The work in kernel is divided on current block and thread indexes `blockIdx.x`, `blockDim.x`, and `threadIdx.x` as shown at Line 11. The main loop starting at Line 16 begins with syncronization to ensure memory coherence. After that, all the threads in a block take part in transferring prediction coefficients to the shared memory and the threads are syncronized as shown at Lines 18 – 21. The inner loop beginning at Line 24 firstly performs a coalesced memory read of four image pixels. It then performs memory accesses of prediction coefficients in shared memory using broadcast access pattern and calculates estimation values as presented at Lines 26 – 29. The rest of the main loop performs coalesced memory read of four image pixels, calculates prediction errors for the pixels, and performs coalesced memory write of four residual image pixels.

5. EXPERIMENTAL RESULTS

For test data, we used NASA's AIRS radiance data. The AIRS data covers 2378 infrared channels in the 3.74 μm to 15.4 μm region of the spectrum. The data gathered during 24-hour period is divided into 240 granules. Each granule consists of 135 scan lines and 90 cross-track footprints per scan line. Ten selected test data granules are available via anonymous ftp*. In order to make the test data more generic to other ultraspectral sounders, 270 bad channels have been excluded from the publicly available test data.[10] The bit depth of a channel ranges from 12 to 14 bits for different channels. The original size of a granule is 41,249,250 bytes.

We evaluated the efficiency of our OpenMP and CUDA implementations on a 2.50 GHz Intel(R) Xeon(R) X3353 running on 64-bit Fedora 10 Linux distribution. Our compiler options for gcc version 4.3.2. were following: -O3 -ftree-vectorize -march=core2 -mfpmath=sse -mmmx -msse -msse2 -msse3 -funroll-loops -fopenmp -lgomp.

Table 2 depicts the computation times for OpenMP and CUDA versions of the linear prediction phase of the LP-CC algorithm.

OpenMP version gave over three fold increase in speed of the linear prediction phase using four or more thread in quad core processor compared to the single thread case. CUDA version increased the speed by a factor of 30 over single threaded CPU version.

Range coding phase using a single processor takes 1.6s. Thus, linear prediction using CUDA is currently faster than the entropy coding phase. In practice, entropy coding speed could be increased by using separate threads for encoding different bands.

Table 2. Prediction times. A number after a '-' refers to a number of OpenMP threads.

Method	time [s]
OpenMP-1	32.9
OpenMP-2	21.7
OpenMP-3	16.3
OpenMP-4	14.4
OpenMP-5	13.2
OpenMP-6	14.3
CUDA	1.1

*ftp://ftp.ssec.wisc.edu/pub/bormin/Count

```
__global__ void evaluate_enc(const signed short int *image_d, \
              signed short int *prediction_error_d, const float *x_d, \
              size_t pitch, size_t pitch_image, int size, int bands){
  short unsigned int z, i, band;
  float p1, p2, p3, p4;
  __shared__ float x[predlen];
  struct __align__ (8) im{
    short int i1, i2, i3, i4;
  } image, prediction_error;

  i = (blockIdx.x * blockDim.x  + threadIdx.x) * 4;

  pitch /= sizeof(float);  // pitch in bytes
  pitch_image /= sizeof(short int);

  for(z=1;z<bands;z++){
  __syncthreads();
  for(int j=0; j<predlen/BLOCKSIZE+1; j++)
    if(threadIdx.x+j*BLOCKSIZE < predlen)
      x[threadIdx.x+j*BLOCKSIZE]=x_d[threadIdx.x+j*BLOCKSIZE+z*pitch];
  __syncthreads();
  p1=p2=p3=p4=0;  // linear predictions for four image pixels
  if(i<size){
    for(band=1; band<=(predlen < z ? predlen :  z); band++){
      image = *((struct im *)& image_d[i+(z-band)*pitch_image]);
      p1 += x[band-1] *  image.i1;
      p2 += x[band-1] *  image.i2;
      p3 += x[band-1] *  image.i3;
      p4 += x[band-1] *  image.i4;
    }
    image = *((struct im *)&image_d[i+z*pitch_image]);
    prediction_error.i1 = ((int)p1)-image.i1;
    prediction_error.i2 = ((int)p2)-image.i2;
    prediction_error.i3 = ((int)p3)-image.i3;
    prediction_error.i4 = ((int)p4)-image.i4;
    *((struct im *)&prediction_error_d[i+z*pitch_image])=prediction_error;
    }
  }
}
```

Figure 5. The CUDA kernel. For illustrations sake the exact boundary check has been omitted.

6. CONCLUSIONS

Shared memory parallelism was achieved using OpenMP directive, which distributed the iterations of the loop among the executing threads. This trivial change in code gave over three fold increase in speed using four or more thread in quad core processor compared to the single thread case.

Programming a GPU using CUDA is more challenging. It requires the programmer to specify the data to be transferred to shared memory for faster access. Furthermore, it also requires the access to the global memory to be coalesced for higher effective bandwidth. Altough our algorithm does not have a high arithmetic operation per memory access ratio, our programming effort increased the speed of the linear prediction by a factor of 30 over single threaded CPU version.

In the future, we will experiment with PGI compiler [†]. It should automatically analyze the program and generate a parallel program to utilize hardware threading and vector capabilities of modern GPUs. Our future work also includes using data compression for the image data, which is used for linear prediction. The data compression would lead to a reduction in bandwidth requirements. This in turn would increase the compression speed of the algorithm.

Acknowledgment

This work was supported by the Academy of Finland.

REFERENCES

1. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide 1.1, www-site: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf
2. T. Halfhill, "Parallel processing with CUDA, Microprocessor", Report, www-site: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf
3. J. Mielikainen, P. Toivanen, "Parallel implementation of linear prediction model for lossless compression of hyperspectral airborne visible infrared imaging spectrometer images", Journal of Electronic Imaging, 14, 013010 (2005).
4. C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, F. Tirado, "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting", IEEE Transactions on Parallel and Distributed Systems, 19(3), 299–310 (2008).
5. S. Lietsch, P. Lensing, "GPU-Supported Image Compression for Remote Visualization - Realization and Benchmarking", Lecture Notes in Computer Science, 5358, 658–668, (2008).
6. V. Simek, R. Asn, "GPU Acceleration of 2D-DWT Image Compression in MATLAB with CUDA", Second UKSIM European Symposium on Computer Modeling and Simulation, 274–277 (2008).
7. A. Plaza, "Clusters Versus FPGA for Parallel Processing of Hyperspectral Imagery", International Journal of High Performance Computing Applications, 22(4), 366–385 (2008).
8. J. Mielikainen, P. Toivanen, "Lossless Compression of Ultraspectral Sounder Data Using Linear Prediction with Constant Coefficients", IEEE Geoscience and Remote Sensing Letters, 6(3), 495–498 (2009).
9. B. Chapman, G. Jost, R. van der Pas, "Using OpenMP: Portable Shared Memory Parallel Programming", The MIT Press, (2007).
10. B. Huang, A. Ahuja, H. Huang, T. Schmit, R. Heymann, "Lossless compression of three-dimensional hyperspectral sounder data using context-based adaptive lossless image codec with bias-adjusted reordering," *Opt. Eng.*, 43(9), 2071–2079 (2004).
11. NVIDIA CUDA Compute Unified Device Architecture Reference Manual, Version 2.0, June 2008.

[†]http://www.pgroup.com/resources/accel.htm