OSGeo
Your Open Source Compass

OSGeo Journal

Volume 6 - September 2010

VISITDENVER.com
VISIT DENVER, THE CONVENTION & VISITORS BUREAU

THE MILE HIGH CITY

DENVER

COLORADO

2010

Official Visitors Guide

FOSS4G 2011 Denver, USA See You There!

More information coming soon..

WALK LIKE AN EGYPTIAN with King Tut at the Denver Art Museum Page 72

July 1, 2010 – January 2, 2011

JULY IS A "WESTERN HEMIS-FAIR" of art, culture and music at the Biennial of the Americas Page 76

July 1 – July 31, 2010

From the Editor...

by Tyler Mitchell

Welcome to the first edition of the OSGeo Journal for 2010! As a good kick-off to the new year this volume takes a few different perspectives on software development and design. Naturally the various issues related to typical development projects applies quite well to our open source geospatial specific interests. The articles cover a range of topics from a review of various software to a discussion of user-centered design. Along the way you'll also get to read some more technically meaty articles and some perspective pieces.

Each volume of the Journal takes several months of concerted effort by many individuals. Landon Blake played a lead editorial role in getting this volume pulled together so you can read it - thank you Landon! It's always a pleasure to have more section editors, LaTeX masters and reviewers come to help. Thank you to all the volunteers.

With our new online management system, any potential article can be submitted at anytime by simply filling in a form at http://osgeo.org/ojs. As well, over the next couple of months keep one eye open for the OSGeo 2009 Annual Report. Get your articles in soon if you have not already. Enjoy the articles!

Tyler Mitchell
Editor in Chief, OSGeo Journal
http://osgeo.org
tmitchell@osgeo.org

Contents of this volume:

Programming Tutorials

GPGPU With GDAL

Basics of GPGPU Interfacing

By Yann Chemin

Introduction

The new generation of graphic cards have Graphical Processing Units (GPU) installed on-board. These GPUs commonly have hundreds of processing cores, high speed parallel architecture, and RAM in the Gigabyte range. Development of GPUs was driven primarily by the processing power required to create and display virtual reality environments in the gaming industry. Now they are used to compute general physics accelerated algorithms for environmental modelling like fluids dynamics.

General-Purpose computation on GPUs (GPGPU) is a relatively new type of computation made possible by the increasingly varied types of computations available on those graphic cards. They are called "coprocessors" in computer engineering. Their high-speed high-parallel architecture makes them very attractive for computations requiring large amounts of operations on each dataset units.

The main point of GPGPU is to manage the memory allocation in the GPU itself. To do this, copy the data into the GPU memory before processing and copying it back to the computer resources outside of the GPU after processing is complete.

In this article we will consider an example of GPU programming. In this example we will use a language for NVIDIA GPUs called Compute Unified Driver Architecture (CUDA).

GPGPU Programming Example

This is a C/C++ language addendum that enables the code to send your data to your GPGPU-enabled NVIDIA graphic card for processing, after which you can retrieve your results from the graphic card RAM and send it to the computer hard disk.

For your computer to understand this code, it needs an additional compiler to be used with your C/C++ compiler. This is not your standard C or C++ compiler. The NVIDIA compiler can be downloaded from http://www.nvidia.com, Look for it in the "CUDA Zone". Read the documentation for your graphic card's driver to see if it already has the CUDA functionality. ION-based laptops are CUDA enabled too.

A CUDA-enabled file `cuda_ndvi.cu`, can be compiled with the NVIDIA compiler (nvcc) like this:

```
nvcc -o ndvicu cuda_ndvi.cu
```

A CUDA-enabled file can be run like this:

```
./ndvicu
```

If you don't have a CUDA capable GPU, compile it in emulation mode:

```
nvcc -deviceemu -o ndvi_cu ndvi_cuda.cu
```

Example Makefile:

```
ndvicu: cuda_ndvi.cu
        nvcc -o ndvicu cuda_ndvi.cu \
        -I/usr/include/gdal/ -L/usr/lib \
        -lgdal1.6.0
```

A typical raster program first loads datasets, define raster data holders for our Red (red), Near Infrared (nir) and Normalized Difference Vegetation Index (ndvi) image bands with GDAL. Then it loads the red and nir bands into line buffers. Using line buffers here is the choice, since the processing is not col/row dependent. GPUs have clear and easy ways to access and work with 2D and 3D pixel localization access within the (GPU) RAM matrix.

First we allocate arrays on the host (our computer itself) so we can fetch the data from our images.

```
 //N = col x row
 int N=nXSize*nYSize;
 float *red=(float *)malloc(sizeof(float)*N);
 float *nir=(float *)malloc(sizeof(float)*N);
 float *ndvi=(float *)malloc(sizeof(float)*N);

 //Load input datasets
 GDALRasterIO(hBandRed,GF_Read,0,0,\
    nX,nY,red,nX,nY,GDT_Float32,0,0);
 GDALRasterIO(hBandNir,GF_Read,0,0,\
    nX,nY,nir,nX,nY,GDT_Float32,0,0);
```

On the GPU side, we start by allocating variables with the suffix `_d` (d for device) to remind us of their location of use, in the GPU device. As everything is a grid (2D or 3D matrix) in a GPU, we allocate an integer N as our image total dimension, this is actually the total length of the grid allocated inside the GPU (it could be written like this: `N=row_d x col_d`).

```
/* pointers to GPU device memory */
float *red_d, *nir_d, *ndvi_d;

/* Allocate arrays on GPU device*/
cudaMalloc((void **)&red_d,sizeof(float)*N);
cudaMalloc((void **)&nir_d,sizeof(float)*N);
cudaMalloc((void **)&ndvi_d,sizeof(float)*N);
```

Once the data is in our computer memory and GPU memory has been prepared to receive the data, it is time to send the row data into the GPU. To do that, we have to use a specific function called:

```
cudaMemcpy(GPU_memory,PC_memory,\
  size_of_data,cudaMemcpyHostToDevice)
```

The last argument indicates the direction of the copy of the data. In this case we send the data from the computer to the GPU, so the direction is `cudaMemcpyHostToDevice`, to retrieve the data after computation, we will use the opposite direction `cudaMemcpyDeviceToHost`.

```
/* Copy data from CPU host to
   GPU device memory */
cudaMemcpy(red_d,red,sizeof(float)*N,\
    cudaMemcpyHostToDevice);
cudaMemcpy(nir_d,nir,sizeof(float)*N,\
    cudaMemcpyHostToDevice);
```

When the data has reached the GPU memory, it is time to apply calculations on it. This is done in GPGPU computing by applying a kernel. The incantation for doing so is shown below:

```
/* Add arrays red, nir and
   store result in ndvi */
ndvi_cu<<<dimGrid,dimBlock>>>(red_d, \
     nir_d, ndvi_d, N);
```

In this case, the kernel to be used is called `ndvi_cu`, it is applied on a grid. That grid is sub-divided in blocks, which are the units of work allocation within the GPU. The grid in our example is N, our row-x-col image dimension. It is visualized as a grid by the GPU and is split into computing blocks limited by the architecture of the GPGPU.

At the time of the writing of this document, lower-end graphic cards are capable of 256 or 512 size blocks. The bigger the block, of course, the less number of job distributions to complete the processing of our row data. If the row data has a size of N=nXSize*nYSize=1024, it will take 4 blocks of 256 or 2 blocks of 512 to compute it. As you can see, Block and Grid allocation are defined into dim3. This is because GPUs, being graphical devices, operate natively in 3 dimensions.

```
/* Compute blocks of data to send to GPU */
// On GeForce 8600 Galaxy x=256
// On GeForce 9500 Galaxy & 9800 GT x=512
int x=512;
dim3 dimBlock(x);
dim3 dimGrid((N/dimBlock.x)+ \
     (!(N%dimBlock.x)?0:1));
```

The kernel was initially a standard C function. It was modified to appropriately take benefit of the GPGPU architecture.

```
__global__ void ndvi_cu(float *red, \
   float *nir,float *ndvi, int N)
{
 int i = blockIdx.x * blockDim.x \
      + threadIdx.x ;
 if ( i < N )
  ndvi[i]=(nir[i]-red[i])/(red[i]+nir[i]);
}
```

Synchronize the data movement with the job completion in the GPU to streamline the processing. This is done by applying the function `cudaThreadSynchronize()`:

```
/* Block until device completed processing */
cudaThreadSynchronize();
```

From this point forward, data is copied back to the computer from the device using the `cudaMemcpy` function. You can then free the memory on the GPGPU device.

```
/* Copy data from GPU device
   to CPU host memory */
cudaMemcpy(ndvi,ndvi_d,sizeof(float)*N, \
           cudaMemcpyDeviceToHost);
cudaFree(red_d);
cudaFree(nir_d);
cudaFree(ndvi_d);
```

After this point, our program will look like a standard C program, as seen earlier, where GDAL takes over the row data and writes it to the disk. We then close the output file and free the memory.

```
GDALRasterIO(hBandNdvi,GF_Write,0,0, \
   nX,nY,nir,nX,nY,GDT_Float32,0,0);
if( red != NULL ) free( red );
if( nir != NULL ) free( nir );
if( ndvi != NULL ) free( ndvi );
GDALClose(hDatasetRed);
GDALClose(hDatasetNir);
GDALClose(hDatasetNdvi);
```

Conclusion

This article is a short path to implementing GDAL-based raster processing in the scalable and highly parallel environment of GPUs.

More information on GPGPU in general can be found at http://www.gpgpu.org.

Yann Chemin
International Centre of Water for Food Security
http://www.csu.edu.au/research/icwater
ychemin AT csu.edu.au

Imprint

Editor in Chief:
Tyler Mitchell - tmitchell AT osgeo.org

Assistant Editor:
Landon Blake

Section Editors:
Daniel Ames
Scott Mitchell
Barry Rowlingson
Jorge Sanz
Micha Silver
Zachary Woolard

Acknowledgements
Various reviewers & writers

The *OSGeo Journal* is a publication of the *OSGeo Foundation*. The base of this journal, the LaTeX 2ε style source has been kindly provided by the GRASS and R News editorial boards.

Journal online: http://www.osgeo.org/journal

OSGeo Homepage: http://www.osgeo.org

Mail contact through OSGeo, PO Box 4844, Williams Lake, British Columbia, Canada, V2G 2V8

9 771994 189002 00

ISSN 1994-1897

OSGeo

Your Open Source Compass

osgeo.org