CITK - an architecture and examples of CUDA enabled ITK filters

Release 0.00

Richard Beare[1], Daniel Micevski, Chris Share
Luke Parkinson, Phil Ward, Wojtek Goscinski1, Mike Kuiper[2]

May 25, 2011

[1]Richard.Beare@monash.edu, Monash University, Melbourne, Australia
[2]mike@vpac.org, Victorian Partnership for Advanced Computing, Melbourne, Australia.

Abstract

There is great interest in the use of graphics processing units (GPU) for general purpose applications because the highly parallel architectures used in GPUs offer the potential for huge performance increases. The use of GPUs in image analysis applications has been under investigation for a number of years. This article describes modifications to the InsightToolkit (ITK) that provide a simple architecture for transparent use of GPU enabled filters and examples of how to write GPU enabled filters using the NVIDIA CUDA tools.

This work was performed between late 2009 and early 2010 and is being published as modifications to ITK 3.20. It is hoped that publication will help inform development of more general GPU support in ITK 4.0 and facilitate experimentation by users requiring functionality of 3.20 or wishing to pursue CUDA based developments.

Contents

1 Introduction

Data must be resident in GPU device memory in order to be processed by the GPU. In order for an ITK filter to be accelerated using GPUs an image must be copied to the device memory and the result copied back if the next filter is not GPU enabled. Copying between host and device memory is quite slow and can easily offset any benefits achieved by faster GPU processing. It is therefore essential that redundant copies between host and device memory are eliminated. It is also desirable that new, GPU enabled, filters can be included in applications without changing programming style.

This article describes a simple modification to the itk::Image class that allows transparent use of CUDA enabled filters. A range of standard filters have been implemented and extensive testing performed.

2 CITK Architecture

The aim of the architecture outlined below was to allow GPU enabled filters to be included in an application without change of programming style or losing performance via redundant host to device memory copies.

A number of architectures were considered. These were derived from online discussions and small samples of code available online:

- Break the pipeline at the beginning of filter execution by copying data to device memory, processing, and then copying back after execution completes. This isolates the GPU code from the rest of the pipeline and requires no change to ITK infrastructure, but introduces redundant copies if subsequent filters are GPU enabled.
- Include interface objects between filters in the pipeline to manage copying. This can eliminate redundant copies but requires that the programmer be aware of which filters are GPU enabled. There is also a minor change of programming style.

Neither of these options require a modification to core ITK classes.

The approach used in CITK does require a modification to core ITK classes, but has a number of advantages. A similar approach has since been outlined on the ITK Wiki http://www.cmake.org/Wiki/ITK_Release_4/GPU_Acceleration.

The fundamental component of the pipeline is the *itk::Image* class. Within this class is a pixel container called *ImportImageContainer*, used to manage the image data. CITK includes a substitute pixel container

named *CudaImportImageContainer*. This pixel container has all the same functionality of the *ImportImageContainer* which results in full compatibility with existing ITK components.

The *CudaImportImageContainer* manages image data on both the host and device. When a standard filter requests the image data, such as through an iterator, the *CudaImportImageContainer* checks whether the most up to date image is on the device or the host. If it is on the device, it is copied back onto the host. This data is then supplied to the user. Similarly when a GPU filter requests the image data, the *CudaImportImageContainer* would check where the most up to date image is, and copy it to the device if required.

The *CudaImportImageContainer* can track where the most up to the date image is by which set command was used last, and assumes the data is modified when a standard iterator requests it.

The result of this is memory transfers are only performed when required and are completed transparent to both the developer and the user. This leaves all the responsibility on the architect, rather than the developer or the user such as in the other attempts.

Some exceptions have been uncovered during this development. Minor changes have also been made to the *AllocateOutputs* method of the *itkInPlaceImageFilter* to support pipelines that connect in place, multi-threaded CPU filters to a CUDA filter. Other exceptions are discussed below.

2.1 Weaknesses

- This framework only supports CUDA, and not OpenCL. The ITK 4.0 proposal supports OpenCL. Limitations of the CUDA development environment mean that even CUDA integration is less complete than hoped, with significant changes to compilation processes being necessary (see below).
- A copy between host and device always results in the source of the copy being considered redundant. This could be inefficient in some cases. The problem is correctly dealing with identical copies on both host and device. If, for example, a pipeline is branched such that one branch in on GPU and the other on CPU, then the branch point is likely to become a source of redundant copies.
- The need to copy between device and host memory breaks some of the usual assumptions, leading to some ugly use of *mutable* declarations in CudaImportImageContainer.
- Morphology filters included in the package use texture memory but are not as generic as the standard ITK versions.
- ITK filters are able to provide their own implementations of key methods, such as *AllocateOutputs*. This can lead to problems when connecting CUDA enabled components to a CPU pipeline. This problem has been observed in the *itkStatisticsImageFilter*, which is a multi-threaded *ImageToImageFilter* with its own AllocateOutputs method passing input through to output. This bypasses the trigger to copy device memory back to the host, leading to incorrect results. Other filters with unusual structure are likely to cause problems. The simple fix for such filters is to call *GetBufferPointer* in the *AllocateOutputs* method.

3 Installation and building

3.1 CUDA compiler and software development kit

This framework requires CUDA 3.2 and the SDK. The *thrust* library is also included in the examples to implement more complex components of some sample filters.

3.2 Fetch this contribution from google code

The source code is included with this article and is also available from google code: http://code.google.com/p/cuda-insight-toolkit/. The patch for ITK is included.

3.3 Patch ITK 3.20

The code distributed with this article includes a patch to modify ITK 3.20, called *patch.3.20.0.dif*. This can be applied as follows:

- fetch http://voxel.dl.sourceforge.net/sourceforge/itk/InsightToolkit-3.20.0.tar.gz
- extract
- cd ITK-3.20
- patch -p0 < path/to/cuda-insight-toolkit/patch.3.20.0.dif

Alternatviely, this code may currently be retrieved via git, as follows:

- git clone git://github.com/richardbeare/ITK.git
- cd ITK
- git checkout v3.20.0_cuda

3.4 Build and install modified ITK

There are many options available when building ITK. This process has been tested under Linux and there are a number of changes to defaults required to avoid limitations to the CUDA development tools.

- Specify location of CUDA SDK. Note that if there is trouble locating libcutil.a, it may be set explicitly under advanced options - CUDA_CUT_LIBRARY.
- Turn off SSE options for VNL - see advanced/VNL. This avoids errors caused by multiple inclusion of SSE files.
- Enable ITK_USE_REVIEW under advanced options, in order to build all of the tests.

3.5 Build examples

The Examples subdirectory in the citk distribution includes a CMakeFile for building all examples and running tests. Location of the patched ITK, nvcc and CUDA SDK must be provided during configuration. It is also necessary to set CITK_USE_CUDA to ON. This option has been included to allow the Insight Journal to run some tests without requiring CUDA development tools.

3.6 Changes to standard processes for building ITK applications

Typical application development in ITK utilizes templates and generic programming and therefore does not require that the developer track new object code dependencies when adding new filters. In principle the same procedure should be possible when using CUDA enabled devices by compiling all application code with nvcc, leading to non-CUDA code being compiled with the host c++ compiler and CUDA code being compiled with CUDA compilers. This would also allow useful templating of CUDA kernels, leading to a relatively seamless integration with traditional ITK development. Unfortunately the current generation of CUDA tools is not able to cope with c++ of the complexity used in ITK. It is therefore necessary to compile CUDA kernels separately, which means the developer must specify the correct object dependencies to the linker. Examples of this can be seen in the CMake files included with this article.

This approach also implies that the CUDA kernels need to be compiled for the appropriate types, and this is currently achieved using a set of macros supporting a limited range of input and output types. Compiling application code with nvcc would eliminate these macros.

Alternatives, such as compiling all CUDA kernels into a library, are feasible but haven't been tested during this development.

4 Anatomy of a CUDA enabled filter

CUDA enabled filters can look very like a conventional ITK filter, with the main difference being a call to a CUDA kernel function from within the *GenerateData* method. CUDA-enabled filters should never have *ThreadedGenerateData* methods as threading is provided within the CUDA portion.

Pointers to device memory are obtained using the *GetDevicePointer* method and are passed to CUDA kernel functions.

4.1 Memory management

Two base classes have been provided to handle standard filter memory management - *CudaInPlaceImageFilter* and *CudaImageToImageFilter*. These filters allow the standard allocation structure to be used via *this→AllocateOutputs()*. Explicit allocation of device memory can be achieved using *Image→AllocateGPU()*

4.2 Templated kernel files

ITK filters are generic with respect to pixel type and dimension and hence CUDA kernels should offer the same flexibility. This is not currently possible. The structure outlined in this section is the best approxima-

tion we have been able to achieve and we hope it will evolve to be better integrated with ITK as the CUDA tools improve.

Each kernel function has a declaration - e.g. CudaAddImageFilterKernel.h contains

```
template <class T, class S> extern
void AddImageKernelFunction(const T* input1, const T* input2, S* output, unsigned int N);
```

The corresponding CudaAddImageFilterKernel.cu file contains:

```
template <class T, class S>
__global__ void AddImageKernel(T *output, const S *input, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N)
    {
    output[idx] += input[idx];
    }
}

template <class T, class S>
__global__ void AddImageKernel(T *output, const S *input1, const S* input2, int N)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx<N)
     {
     output[idx] = input1[idx] + input2[idx];
     }
}

template <class T, class S>
void AddImageKernelFunction(const T* input1, const T* input2, S* output, unsigned int N)
{


    // Compute execution configuration
    int blockSize = 128;
    int nBlocks = N/blockSize + (N%blockSize == 0?0:1);

    // Call kernels optimized for in place filtering
    if (output == input1)
      AddImageKernel <<< nBlocks, blockSize >>> (output, input2, N);
    else
      AddImageKernel <<< nBlocks, blockSize >>> (output, input1, input2, N);
}

#define THISTYPE float
template void AddImageKernelFunction<THISTYPE, THISTYPE>(const THISTYPE * input1,
                                                         const THISTYPE * input2,
                                                         THISTYPE * output, unsigned int N);
#undef THISTYPE
#define THISTYPE int
template void AddImageKernelFunction<THISTYPE, THISTYPE>(const THISTYPE * input1,
                                                         const THISTYPE * input2,
```

```
                                                    THISTYPE *output, unsigned int N);
#undef THISTYPE

#define THISTYPE short
template void AddImageKernelFunction<THISTYPE, THISTYPE>(const THISTYPE * input1,
                                                    const THISTYPE * input2,
                                                    THISTYPE *output, unsigned int N);
#undef THISTYPE

#define THISTYPE unsigned char
template void AddImageKernelFunction<THISTYPE, THISTYPE>(const THISTYPE * input1,
                                                    const THISTYPE * input2,
                                                    THISTYPE *output, unsigned int N);
#undef THISTYPE
```

This is a simple structure that allows a number kernels to be precompiled for different voxel types.

4.3 Using *thrust* algorithms

The thrust project, http://code.google.com/p/thrust/, is a source of templated, CUDA-enabled algorithms. The CudaStatisticsImageFilter makes use of these algorithms. It is also possible to implement simple arithmetic filters using the thrust *transform* algorithm (leading to more elegant code), but preliminary tests suggest a significant loss in performance. There are examples of thrust-based arithmetic in several sample filters that can be switched on with a cmake option.

5 Testing

The online testing within the Insight Journal does not support CUDA and therefore cannot be used to test this contribution.

5.1 ITK

Changes to ITK classes have been tested using standard ITK tests, producing the same results as an unmodified ITK.

5.2 Testing CUDA filters

A range of simple CUDA enabled filters have been developed and compared to the CPU equivalents. CMake based test are included in this contribution.

6 Performance

Improved computational performance is the reason for interest in GPU imaging applications. However there are many stories of how difficult this is to achieve in practice, and similar difficulties are likely to be experienced in the imaging domain. Some of the difficulties we forsee are:

- A real imaging application is likely to utilize a large number of ITK filters. It is likely to be a long time before a significant portion of ITK is CUDA (or GPU) enabled. Therefore a developer will only experience a nett performance gain if the CUDA enabled filter offers sufficient speedup to offset memory transfer costs and a small proportion of the application is particularly time consuming.
- Filters that are easy to port to the GPU tend to be fast on CPU anyway, and typically don't represent a large proportion of application time. Examples include all voxel-wise operations, such as masking and arithmetic.
- Many potentially time consuming operations, such as filtering with large kernels, have been highly optimised for CPU implementation. It is important the comparisons are made with these optimised CPU implementations.

We won't discuss the mechanics of CUDA performance profiling in this article - there are many resources available online.

One point worth noting when testing CUDA enabled ITK filters is that there is a per-process cost associated with running a CUDA-enabled application. This cost appears to relate to a number of things, including loading libraries and initializing the device. This cost can be very significant - as much as 2.5 seconds on one of our test machines - and can give an exagerated negative impression of the filter performance.

Finally, some positive performance results. These tests were carried out using a Tesla T10 in a 16 core, 2261.051MHz, Intel L5520 Xeon:

- Performance improvement of 190 times observed with simple arithmetic, such as adding or subtracting constants from images. See *simple_perf_test.cxx* for details.
- Image filtering with kernels, for example simple means, offer 32 times speedup for 3d kernels, radius 10 voxels on images size $500 \times 500 \times 500$ and 50 times speedup for kernels radius 20.. This is potentially interesting for applications requiring kernels of specific shape, because most accelerated CPU schemes can only implement a restricted range of shape kernels. Neither CPU nor GPU examples exploits redundancy in this test.

7 Conclusions

We have provided some simple modifications to ITK infrastructure that allow integration of CUDA enabled filters with ITK applications, and provided a number of examples and validation tests. We hope this framework will encourage immediate experimentation with GPU filters and inform some of the GPU development scheduled for ITK 4.

References

[1] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, 2003.