Scene graph-based construction of CUDA kernel pipelines for XIP

Veronica Gidén[1], Thomas Moeller[1], Patric Ljung[1] and Gianluca Paladini[1]

[1]Department of Imaging and Visualization
Siemens Corporate Research, Inc.
755 College Road East, Princeton, NJ 08540
{veronica.giden.ext, thomas.moeller, patric.ljung, gianluca.paladini}@siemens.com

Abstract. We propose a framework which allows an application developer to construct and execute pipelines of existing CUDA kernel programs without programming the somewhat complex kernel configuration and setup. The framework is a new addition to the eXtensible Imaging Platform (XIP) of the National Cancer Institute. Pipeline construction is carried out through graphical construction of scene graphs in the XIP Builder tool. Complex pipeline structures as well as kernels of arbitrary structure and function are supported. The framework has been used to execute existing CUDA kernels from NVIDIA's CUDA SDK as well as a more complex image segmentation algorithm.

Keywords: General-purpose computation on GPUs (GPGPU), graphics processing units (GPUs), Compute Unified Device Architecture (CUDA).

1 Introduction

During recent years, the role of the graphics processing unit (GPU) has evolved from only handling graphics to functioning as a powerful co-processor to the central processing unit (CPU). This evolution is due to the GPU's rapid increase of performance in comparison to the CPU's, as well as its specialization towards compute-intensive, parallel execution. Medical imaging is one of many fields where these kinds of computations are used for other purposes than pure graphics processing. In this field, handling of large medical data sets is common, requiring high performance computations in order to assure speed and quality, thereby making the GPU a valuable asset. Previously, programming of GPUs was commonly carried out using high-level shading languages and it was necessary to map the algorithm to the graphics pipeline. Thus, utilizing the GPU for general-purpose programming was non-intuitive and cumbersome. Furthermore, there were several limitations on programmability which prevented usage of the GPU's full capacity. NVIDIA's Compute Unified Device Architecture (CUDA) [1], aims to solve these problems by introducing a new hardware and software architecture for the GPU, particularly suitable for general-purpose programming. Here, GPU programs are created using a C like syntax and a fast GPU cache can be explicitly accessed. However, executing an existing CUDA program on the GPU is non-trivial and requires extensive setup and configuration. In this paper a framework which facilitates this procedure is proposed. The framework can be used for CUDA application development and performs configuration and execution of pipelines of existing CUDA GPU programs without requiring any programming by the developer.

2 Background and related work

The purpose of the proposed framework is to simplify CUDA application development. The CUDA architecture is described briefly in section 2.1. The framework is implemented as a new addition to the eXtensible Imaging Platform (XIP) [2] of the National Cancer Institute [3], an overview of XIP is given in section 2.2.

2.1 CUDA

One of the main purposes of CUDA is to facilitate GPU programming by making the underlying architecture available to the developer through an API not related to graphics and thereby expose the GPU as a pure co-processor to the CPU. Contrarily to previous GPU architectures, the CUDA architecture allows *scattered writes*, that is, writing to arbitrary locations in DRAM. Furthermore, the architecture gives the developer access to a parallel data cache with short access time, the so-called shared memory. This decreases the need for costly GPU DRAM accesses, thereby potentially avoiding an otherwise significant bottleneck.

The CUDA execution model is based upon GPU programs known as *kernels*, which are executed over multiple threads in parallel. Threads are arranged into structures known as blocks, which in turn are arranged into grids. Threads within the same block can communicate with each other using the shared memory space and can furthermore also be synchronized.

Kernels are programmed using a C like syntax without references to the graphics pipeline. However, in order to execute a kernel, a quite complex CUDA setup and kernel configuration has to be carried out. There are two APIs available in CUDA for this configuration; the *Driver API* and the *Runtime API*. The former is more flexible but quite complex to use, whereas the latter generally requires less programming effort from the user but provides less control. However, only the Driver API completely separates GPU kernel code from code to be executed on the CPU and allows separate compilation of these. Using this API, kernel code is compiled into so-called *cubin objects* which can be loaded from disk and used to execute the contained program on the GPU.

CUDA has been used for a variety of applications, resulting in improvements such as considerable speed increases and reduced programming effort. For example, it has been used within medical imaging by Jeong et al. [4], specifically to perform optimal path analysis in diffusion tensor magnetic resonance imaging data from the brain. Scherl et al. [5] have used the technology for three-dimensional reconstruction of volumetric data from computed tomography. Another application comes from the bioinformatics field, where Manavski and Valle [6] have used CUDA to search for similarities in protein and DNA databases. An application from the scientific computing field has been demonstrated by Michalakes and Vachharajani [7], who have implemented the Weather Research and Forecast model in CUDA.

2.2 XIP

XIP is an Open Inventor-based open source environment for rapid development of medical imaging applications. Open Inventor [8] is an object-oriented API for three-dimensional graphics, built upon the scene graph concept and written in C++. It functions as an extra layer upon OpenGL [9], providing a library of modifiable and extendable scene graph objects which can be used to describe and control a three-dimensional scene. A scene graph is a general data structure, arranged as a tree or a graph and consisting of an ordered number of nodes which are connected to each other. The ordering determines how the graph is traversed, which might e.g. occur during rendering. The traversal state describes the current properties and can both be used and influenced by scene graph nodes.

A graphical editor, XIP Builder, is provided in XIP. This editor allows creating scene graphs representing medical imaging applications through a drag-and-drop visual programming interface.

This paper describes an addition to the XIP framework to be used in the scene graph-based XIP Builder tool, allowing the application developer to construct and execute pipelines of CUDA kernels without programming the quite complex kernel configuration required by the CUDA Driver API, while at the same time benefiting from the detailed control supplied by this API.

3 Scene graph-based CUDA application development framework

The created framework is an addition to the XIP framework, made available to application developers in the XIP Builder editor tool. It is focused on one scene graph entity in particular; the pipeline engine, which executes CUDA kernels and is further described in section 3.1. The framework also contains data types for accessing data stored in GPU memory, which are described in section 3.2. Moreover, scene graph objects which perform efficient data transfer between the OpenGL texture and the CUDA memory spaces are provided, and described in section 3.3. Finally, the CUDA application development process using the framework is described in section 3.4.

3.1 The pipeline engine

The pipeline engine is the central unit of the created framework. It takes data residing in CUDA memory space as inputs, executes one or several CUDA kernels in a pipeline, and produces outputs in the same memory space. Consequently, another pipeline engine can use these outputs as its inputs and no intermediate transfer to CPU memory space is necessary. The engine supports a wide variety of pipeline structures, including iteration. The idea of the pipeline engine is visualized in figure 1.

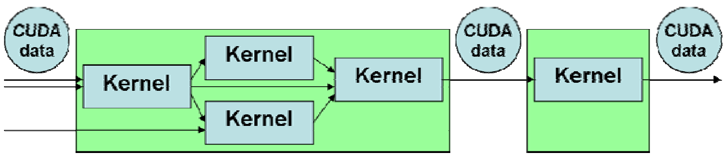


Fig. 1. The pipeline engine of the proposed framework executes one or several CUDA kernels in a pipeline, using inputs in CUDA memory space and producing outputs in the same memory space. The outputs can be connected as inputs to another pipeline engine.

Importantly, the pipeline engine completely handles all CUDA setup and kernel configuration and only leaves out the programming of the kernel itself. Thus, the XIP Builder developer does not need to edit the code of this object in order to execute a different CUDA kernel or to change the kernel configuration. Instead, the necessary configuration is specified through the fields of the pipeline object in XIP Builder, and then performed by the engine using the CUDA Driver API. The descriptions in the fields either consist of strings, using an intuitive, keyword-based syntax, or pure numbers. The fields that need to be filled out by the developer define the following properties:

- Which kernels that should be executed and in which order
- The input and output parameters of each kernel
- Texture properties for the used CUDA textures
- Properties of temporary CUDA linear memory *buffers*
- The outputs of the pipeline object
- The block and grid configuration used by each kernel
- The amount of shared memory used by each kernel
- The iteration condition for the pipeline
- Which cubin object that should be used to load kernels from

Here, *buffers* refer to chunks of CUDA linear memory which can be read from and written to by kernels. Each buffer is identified by an integer in the range *[0, number of buffers[*. A buffer can be initialized with input data supplied through the fields of the engine. It can furthermore be set as an external output of the engine, thereby becoming available to other scene graph objects. When describing the inputs and outputs of each kernel, the developer specifies which buffers that each kernel writes to or reads from. In this way, complex pipelines can be constructed. The idea is illustrated in figure 2.

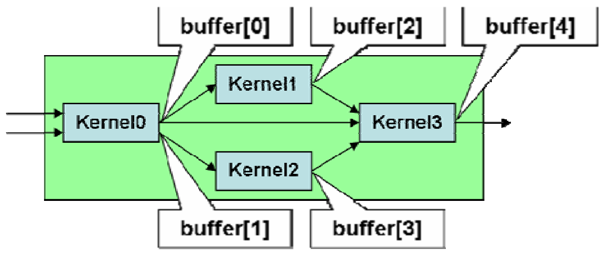


Fig. 2. Within the pipeline engine, CUDA linear memory which can be read from and written to by kernels is referred to as buffers. For example, in the pipeline above, Kernel0 writes to buffer 0, which is read by Kernel1 and Kernel3. Kernel3 writes to buffer 4, which is set as an external output of the engine.

As for inputs, the engine takes input data, existing in CUDA linear memory or as CUDA arrays, wrapped in data structures compatible with the XIP Builder environment. Data of smaller size stored in CPU memory is also supported, e.g. integer and float parameters.

The engine uses the information retrieved from its input fields to launch the kernels contained in the loaded, pre-compiled cubin object. The kernel configuration is checked with the system capabilities and the developer is warned if any hardware properties are violated.

3.2 CUDA memory data storage

In the implemented framework, the opportunity to access data stored in CUDA memory space, either in CUDA linear memory or as CUDA arrays, is provided by two different wrappers constituting an interface towards XIP Builder. CUDA linear memory can be read from and written to in kernels, whereas CUDA arrays can only be read from. In addition, CUDA arrays require reading through texture fetches, and the array needs to be connected to a CUDA texture reference. Only the texture reference itself can not be stored in the XIP Builder scene since it is specific to a certain cubin object and becomes invalid as soon as the cubin object is unloaded. Thus, a handle to the CUDA array is stored in the wrapper object. Before kernel execution is performed within a pipeline engine, these CUDA arrays are connected to texture references and thus become readable in kernels. Combining CUDA textures and CUDA arrays provide a number of benefits, such as boundary handling and interpolation schemes, further described in [1].

3.3 OpenGL interoperability

In order to allow efficient conversion between OpenGL textures and CUDA data four scene graph nodes have been implemented. These nodes employ CUDA Driver API functions for the internal interface between CUDA memory and OpenGL texture memory, relying on OpenGL buffer objects, see [10].

The first two nodes take input data in CUDA memory space (CUDA linear memory or CUDA arrays) and produce an OpenGL texture for each data set. An OpenGL pixel buffer object is registered with CUDA and mapped to a location in CUDA linear memory space. Then, the data contained in CUDA memory space is copied to the buffer location, and the data residing in the buffer object is used to create an OpenGL texture. The outputs of the nodes are OpenGL texture IDs representing the created textures. The final two nodes take one or several OpenGL texture IDs as inputs and produce data in CUDA memory space. Here, OpenGL texture data is copied to an OpenGL pixel buffer object which is mapped into CUDA memory space.

3.4 CUDA application development

As previously stated, one of the main aims of the proposed XIP Builder CUDA framework is to enable easy execution of existing kernels, by performing automatic CUDA setup and kernel configuration without requiring the developer to program this. The application development process is thus divided into two parts; kernel programming and scene graph construction.

CUDA kernel programming and compilation is carried out outside of the XIP Builder editor. CUDA kernels are programmed using the CUDA API [1]. In order to make the kernels loadable from scene objects in XIP Builder, the code is compiled into cubin objects using the NVIDIA nvcc compiler [11].

Scene graph construction, which is carried out in the XIP Builder editor, is centered on the pipeline engine which loads and executes pre-compiled CUDA kernels. The first step during application development is commonly to transfer data to CUDA memory space, since the kernels to be used most likely require some input data. This is carried out by converting data from XIP Builder data formats through framework-provided converters, or the nodes that convert OpenGL textures to CUDA data. Once CUDA data is available in the scene, the XIP Builder developer places one or several pipeline engines in the scene graph and connects the input data to the appropriate fields. The next step is to fill out the remaining fields in order to define the contained pipeline and set up the configuration for the pre-compiled CUDA kernels. This requires that the developer knows what grid, block and shared memory sizes that are appropriate for each kernel, as well as its inputs and outputs. However, it does not require any programming in CUDA or any other language. When the pipeline has been defined, the developer connects the outputs of the pipeline engine to input fields of other scene graph objects. If an output is connected to an input field of another pipeline engine, the data existing in CUDA memory space can be conveniently used as input data in kernels without requiring any extra data transfers. If an output is connected to other scene graph objects, the data can for example be used to create OpenGL textures which can be displayed on the screen, see figure 4.

4 Results

In order to evaluate the implemented CUDA XIP Builder framework, one simple and one advanced example application have each been developed in two different ways each. This includes an implementation using the pipeline engine from the new framework, as well as an implementation where a specialized XIP scene graph object is programmed using the CUDA Driver API. The purpose of the latter implementation is solely to function as a reference. In this section, the development processes for these applications, as well as the results in terms of scene graph structure and performance, are presented and analyzed. All execution time results can be found in table 3.

4.1 Box filtering

The simple implemented application smoothes an image by executing the two kernels from the *boxFilter* application from NVIDIA's CUDA SDK [12].

When using the implemented CUDA XIP Builder framework to perform box filtering, the input image data (existing in CUDA linear memory space) is set as an input to a pipeline engine. The engine loads the kernels contained in a pre-compiled cubin object. These are iterated over in order to perform repeated smoothing, and the number of iterations is controlled through a field of the pipeline engine. The field specification, excluding empty fields, is described in table 1. The main field is the CUDA kernel headers description field, where the execution order of the kernels as well as their parameter setup is defined. Two buffers are used internally, both of which are the same size and format as the image to be smoothed, see figure 3. One of the buffers contains the smoothed image and is set as an engine output, ready to be displayed on the screen. A snapshot of the application being developed and executed in XIP Builder can be seen in figure 4.

For evaluation purposes, box filtering using the NVIDIA SDK kernels has also been implemented by manually programming a specialized XIP Builder scene graph engine using the CUDA Driver API. The resulting scene graph is identical to the one in figure 4, with the exception of that the pipeline engine (the second scene graph object from the left) is replaced with this specialized engine. However, the field configuration is much simpler than when using the pipeline engine. Since kernel configuration and execution are programmed manually, the only fields that need to be filled out are the input data and parameter fields.

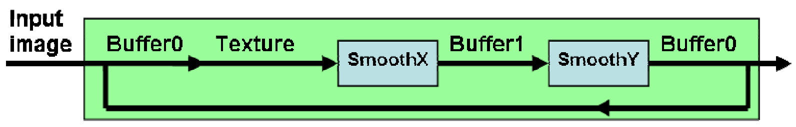
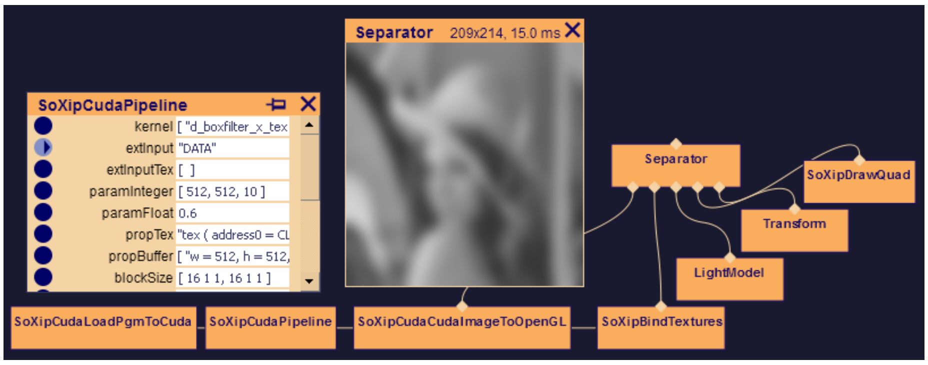


Fig. 3. When using the pipeline engine from the proposed framework to perform box filtering using kernels from the NVIDIA CUDA SDK, two CUDA linear memory buffers are used internally.



kernel	["d_boxfilter_x_tex
extInput	"DATA"
extInputTex	[ ]
paramInteger	[512, 512, 10]
paramFloat	0.6
propTex	"tex (address0 = CL
propBuffer	["w = 512, h = 512,
blockSize	[16 1 1, 16 1 1]

Fig. 4. A snapshot of the boxFilter application from the NVIDIA CUDA SDK being executed in XIP Builder. An image file is loaded from disk, transferred to CUDA memory space, processed by the boxFilter CUDA kernels and then used to create an OpenGL texture which is displayed on the screen.

Table 1. The field specification of a pipeline engine used to perform box filtering.

Field	**Content**
CUDA kernel headers description	["d_boxfilter_x_tex (IN [1,tex[0]], OUT [0], INT [0], INT [1], INT [2])", "d_boxfilter_y_global (IN [0], OUT [1], INT [0], INT [1], INT [2])"]
External input CUDA linear memory data	"DATA"
Integer parameters	[512, 512, 10]
CUDA texture properties description	"tex (address0 = CLAMP, filter = POINT, useNormCoord = FALSE, address1 = CLAMP, readAsInt = FALSE, components = 1, format = FLOAT)"
Internal buffer properties description	["w = 512, h = 512, d = 1, format = FLOAT, type = LUMINANCE", "srcType = EXT_INPUT, srcIdx = 0, dstType = IMAGE, dstIdx = 0"]
Kernel block sizes	[16 1 1, 16 1 1]
Kernel grid sizes	[32 1, 32 1]
Condition for iteration	"5"
Cubin object path	" cuSDKBoxFilter.cubin"

4.2 Segmentation using the Random Walker algorithm

The advanced example application performs image segmentation using the random walker algorithm presented by Grady [13]. This algorithm is used to perform image segmentation based on an existing seed image, where one or several pixels are labeled as belonging to one or several different objects in the image. Segmentation is achieved by placing a random walker at each pixel in the image, noting the label of the seed pixle to which it is most probable that it arrives to first, and then finally assigning this label to the start pixel. In practice, this is carried out by iteratively solving a large system of linear equations [13]. Apart from the iterative segmentation stage, the random walker algorithm also consists of an initialization stage.

When using the implemented CUDA XIP Builder framework, two pipeline engines are needed in order to perform random walker segmentation. The reason is that the pipeline engine only allows iteration over *all* its kernels, and it is not desirable to iterate over the initialization stage. Thus, the initialization stage is performed by one pipeline engine which takes CUDA linear memory data, representing the image to be segmented as well as a seed image, as inputs. Initialization is performed through execution of twelve kernels. The resulting output data sets are used as inputs to another pipeline engine which carries out the iterative stage. The pipeline engine performing the iterative stage iterates over eleven kernels until a certain condition, specified through a string in an engine input field, is fulfilled. The engine output is a CUDA linear memory data set describing the segmented image. The field specifications of the two pipeline engines are much more complex than for the box filter application. As an example of the increased complexity, the CUDA kernel headers description field of the pipeline engine performing iteration can be seen in table 2.

Table 2. The CUDA kernel headers description of a pipeline engine used to perform the iterative part of segmentation through the random walker algorithm.

CUDA kernel headers description
["updateQcoalesced(EXTIN[2], EXTIN[5], EXTIN[1], INT[0], INT[1], INT[2], EXTIN[8])", "updateAlphaInit(EXTIN[8], EXTIN[5], INT[3], INT[0], EXTIN[9])", "reduceAddFloat(EXTIN[9], EXTIN[13], INT[4], INT[4], INT[4])", "updateAlphaFinalize(EXTIN[13], EXTIN[10], EXTIN[11], EXTIN[13])", "updateX(EXTIN[7], EXTIN[5], EXTIN[13], EXTIN[7], INT[3])", "updateRcoalesced(EXTIN[2], EXTIN[7], EXTIN[3], EXTIN[1], INT[0], INT[1], INT[2], EXTIN[4])", "updates(EXTIN[4], EXTIN[2], EXTIN[1], INT[0], INT[1], INT[2], FLOAT[2], EXTIN[6])", "updateRhoInitPreconditioned(EXTIN[4], EXTIN[6], INT[3], INT[0], EXTIN[9])", "reduceAddFloat(EXTIN[9], EXTIN[11], INT[4], INT[4], INT[4])", "updateBeta(EXTIN[10], EXTIN[11], EXTIN[14])", "updateD(EXTIN[6], EXTIN[5], EXTIN[14], EXTIN[5], INT[3])"]

The internal buffer structures, defined through appropriate string fields, are also considerably more complex than for the simple example. The resulting XIP Builder scene graph is almost identical to the one in figure 4, except for that there are two pipeline engines instead of one.

When the random walker application is implemented by programming a specialized XIP engine executing the appropriate kernels, the whole algorithm can be contained in one scene graph object, making the scene graph identical to the one in figure 4. The field specification is simpler than when using the pipeline engine, since the only input fields needed are the input data and parameter fields.

4.3 Analysis of results

The results described in the previous section show that the proposed framework supports development of both simple and complex applications. It provides a great advantage compared to when specialized XIP Builder engines are programmed for each application; i.e. that no CUDA Driver API programming is necessary in order to execute precompiled CUDA kernels. Also, since the entire pipeline is controlled through input fields, updating the pipeline structure does not require any recompilation. Instead, this is achieved by directly changing the input field values in XIP Builder. However, for more complex examples such as the random walker algorithm, filling out the engine fields describing the pipeline becomes quite cumbersome. Furthermore, complex pipelines are more likely to require a larger number of pipeline engines, resulting in increased scene graph complexity. Moreover, as can be seen in table 3, the pipeline engine introduces overhead resulting in decreased performance. This is a result of its general structure, which e.g. includes interpretation of input string fields (parsing), checking of CUDA requirements and buffer handling, see table 4. Also, some delay is introduced during kernel execution since parameters are accessed through lists that need to be traversed. The performance decrease tends to be more severe for complex applications with a larger number of parameters, kernels and internal buffers, which all influence the time consumption of these operations. Also, manually programmed specialized engines provide better optimization possibilities, since the kernel execution and configuration as well as memory allocation can be adapted to each specific application.

Table 3. Example execution times for applications developed using the pipeline engine from the proposed framework or through programming of a specialized XIP Builder engine. All tests were performed using the following setup: Pentium 4 (2.80GHz and 2.79 GHz) with 2.00 GB RAM and an NVIDIA Geforce 8800 GTS graphics card.

Application	**Execution time using pipeline engines (ms)**	**Execution time using specialized engines (ms)**
Box Filtering (100 iterations) of a 512x512 floating point data set.	107.4954780	104.7613884
Random Walker Segmentation of a 128x128x128 floating point data set.	9 532.2740510	6 842.9303450

Table 4. Example time consumption results for specific parts of the implementation of the boxFilter example from the NVIDIA CUDA. The filtering was performed 100 times over a 512x512 floating point data set with the same setup as in table 3.

Operation	**Time using pipeline engines (ms)**	**Time using specialized engines (ms)**
Clean up	0.51129200	0.76651833
Retrieve and check input data	0.00887561	0.00963296
Check CUDA requirements	1.50713500	0
Parse input field strings	0.60431100	0
Handle cubin object and handles	0.01559400	0.00970700
Init buffers	0.44454400	0
Allocate CUDA memory	0	0.90882400
Kernel setup	0.00918971	0.00655589
Launch kernels	104.06700000	103.04100000
Set outputs	0.32753600	0.01915070

5 Limitations

Although the implemented framework allows execution of a wide variety of CUDA kernels in complex pipeline structures, it has limitations that restrict its usage.

A notable limitation concerns the required CUDA knowledge of the XIP Builder developer. Although kernel programming and pipeline construction are separated, the developer is in practice required to know the structure of the kernel function header in order to provide correct input data. Moreover, the amount of shared memory to allocate has to be known, as well as what block and grid configuration that is suitable to use. Inappropriate specification of these properties may result in incorrect results or failure of kernel execution. Although CUDA errors are reported to the developer the error messages available in CUDA tend to be very general, and without an understanding of the underlying hardware they might therefore be of little use. Importantly, there are no warnings concerning violations of the parallel programming model since these do not necessarily result in CUDA errors. To summarize, it is vital that the XIP Builder developer has some knowledge and understanding of CUDA, the parallel programming model of GPUs and of the kernels that are used.

Furthermore, although complex pipelines can be specified through the pipeline engine's fields, there are indeed scenarios that are not supported. First, in order to use this engine, all calculations in the pipeline are required to be carried out through CUDA kernels. If CPU calculations are needed, specialized scene graph objects need to be created. Second, scenarios where data does not fit in GPU memory are not supported. Third, the functionality of the pipeline engine is restricted to pipeline structures that can be described by the syntax used for its input fields. For example, it does not support all functionality provided by the CUDA Driver API. This includes the recently provided support for three-dimensional CUDA textures as well as usage of streams [14].

6 Conclusion

The presented CUDA XIP Builder framework completely separates CUDA kernel coding from pipeline construction and automatically handles kernel configuration and CUDA setup according to user specifications through string fields. It thereby supports creation of CUDA pipelines through scene graph construction without requiring any CUDA programming except for of the kernels to be executed. Furthermore, it provides data structures for accessing data stored in CUDA memory space in XIP Builder, thereby avoiding costly transfers between host and device between scene graph objects. Also, it supports conversion between OpenGL textures and CUDA data without intermediate CPU memory storage.

As seen in section 4, the generality of the framework causes disadvantages which are more evident for complex pipelines, the most serious one being the decrease in performance. The created framework is thus more suitable for prototyping purposes where the XIP Builder developer quickly wants to test a CUDA kernel pipeline rather than to optimize it.

There are still possibilities for improvement of the created framework. Since the string-based kernel configuration and setup tend to become impractical for complex applications, it would certainly be beneficial to let the user control these through a graphical drag-and-drop editor. A more user-friendly interface could also allow further functionality extension of the pipeline engine, e.g. including new functionality provided in CUDA 2.0 [14].

References

1. NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.1, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf
2. Siemens Corporate Research: Xipwiki, https://collab01a.scr.siemens.com/xipwiki/index.php/Main_Page
3. National Cancer Institute: Comprehensive Cancer Information, http://www.cancer.gov/
4. Jeong, W., Fletcher, P.T., Tao, R., Whitaker, R.: Interactive Visualization of Volumetric White Matter Connectivity in DT-MRI Using a Parallel-Hardware Hamilton-Jacobi Solver. In: IEEE Transactions on Visualization and Computer Graphics, vol. 13, no. 6, pp. 1480--1487. IEEE Computer Society, Los Alamitos (2007)
5. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast GPU-Based CT Reconstruction Using the Common Unified Device Architecture (CUDA). In: Nuclear Science Symposium Conference Record, vol. 6, pp. 4464--4466 (2007).
6. Manavski, S.A., Valle, G.: CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. In: BMC Bioinformatics, vol. 9, suppl. 2 (2008)
7. Michalakes, J., Vachharajani, M.: GPU Acceleration of Numerical Weather Prediction. Technical report (2008). http://www.mmm.ucar.edu/wrf/WG2/michalakes_lspp.pdf
8. Wernecke, J.: The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2. Addison-Wesley (1994)
9. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2. Addison-Wesley, Stoughton (2006)
10. Wright, S.R. Jr., Lipchack, B., Haemel, N: OpenGL SuperBible: Comprehensive Tutorial and Reference. Addison-Wesley, Ann Arbour (2007)
11. NVIDIA Corporation: The CUDA Compiler Driver NVCC, http://www.nvidia.com/object/io_1195170069217.html
12. NVIDIA Corporation: Download CUDA Code, http://www.nvidia.com/object/cuda_get.html
13. Grady, L.: Random Walks for Image Segmentation. In: IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.28, no.11, pp.1768—1783 (2006)
14. NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.0, http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf