A Higher-Level and Portable Approach for GPGPU Programming

Andrew V. Adinetz

Abstract—**With the growth of GPU programmability and processing power, GPGPU has become a topic of intensive research recently. This research, however, has been impeded by the lack of adequate instruments for GPGPU programming.**

In this paper we perform an overview of existing higher-level means for GPGPU programming. We also propose an idea of our own language, C$, to be used for it and report some of the achieved results.

Index Terms—**Computer Languages, Parallel Programming, GPGPU**

I. Introduction

General purpose computations on graphics processor units (GPGPU) [1] have been a major topic of research in recent years. GPUs have been used, besides others, for acceleration of linear algebra computations [2], image processing [3], CFD numerical methods [4] and ray tracing [5].

In most of these works, GPGPU programming has been done directly using an API to interact with programmable graphics hardware, like OpenGL [6]. Such an approach, however, has a number of drawbacks. First, although GLSL shader language [7] can be thought of as a high-level language, graphics hardware interfaces tend to provide lower-level access to graphics hardware, and it requires a large number of operations to be performed (texture loading and binding, frame buffer binding, shader binding etc.) to execute a single step (a single shader) of a GPGPU program. This complicates development and debugging of a GPGPU application. Second, although OpenGL tends to be portable between various GPUs and operating systems, different GPUs differ in the subset of the shader language they support in hardware. Moreover, various GPUs have different hardware architectures: e.g., ATI X1k [8] and NVIDIA GeForce 6 and 7 series process quadruples of floats, while NVIDIA GeForce 8 series use scalar pipelines. Therefore, array operations, which are very typically used in GPGPU programming, need to be translated differently to various architectures to achieve higher performance. Third, a GPGPU program written in terms of OpenGL and shader language is not portable to non-GPU architectures, which, however, have some similar properties, such as CELL [9].

Therefore, a higher-level means for GPGPU programming is required. This being realized, a number of such approaches appeared, starting from Brook for GPU [10].

The rest of the paper is organized as follows. We overview existing approaches to higher-level GPGPU programming in section II. In Section III, we present the C$ language, while in section IV we discuss the architecture of the underlying system and how the language can be translated to a GPU program. In section V, we report some of the results currently achieved, followed with a brief discussion.

II. Existing Approaches to Higher-Level GPGPU Programming

After the tediousness of GPGPU programming using bare OpenGL had been realized, higher-level approaches started to appear. Typically, they provided a certain level of abstraction, varying from simple classes wrapping OpenGL or DirectX shaders and textures into arrays and kernels, to array processing libraries with support for lazy evaluation and JIT shader generation. Approaches can also be classified as general approaches to GPGPU and domain-specific approaches, e.g. image processing libraries on GPU, such as CoreImage [11]. Here we shall primarily consider general approaches.

Historically, Brook GPU [10] has been the first higher-level GPGPU approach. Brook GPU is an extension to C which adds stream types to the language (written as T<>) and kernel functions which operate on streaming types. As streaming programming is the basic approach to GPGPU [11], the concepts of Brook language (with some limitations) mapped relatively straightforwardly to the GPUs available at the time, while providing a significantly higher level of abstraction than bare OpenGL. A number of GPGPU applications, e.g. [5] and [12]. A compiler has been created which compiled the Brook code to a sequence of SM 2.0 shaders and respective API calls. With the appearance of SM 3.0, however, the language failed to support new features of the new shader model, and therefore fell out of use.

Brook has been a higher-level approach, and has been relatively successful for its time. As it worked relatively well with streams, it did not provide efficient means for array programming and for application of shaders to arrays. The complexity of its integration with existing programming languages also contributed to its decline, as well as failure to support SM 3.0.

Manuscript received April 10, 2007. This work was supported in part by ATI (now division of AMD) under its fellowship award for Ph.D. students.

A. V. Adinetz is a Ph.D. student at Moscow State University, Faculty of Computational Mathematics and Cybernetics, Moscow, 119992 Russia (corresponding author to provide phone: 926-2833921; e-mail: adinetz@ cs.msu.su).

Sh [14] is a different approach to GPGPU. Instead of providing a new language, it uses an existing one (C++) and wraps shaders and textures in C++ classes, providing a higher-level streaming programming abstraction. Sh uses metaprogramming to embed shaders directly into C++ programs as C++ code. This is achieved by surrounding shader code with macros (such as SH_BEGIN_FRAGMENT_PROGRAM and SH_END_PROGRAM), which, in fact, wraps a shader into quotes and a call to shader object constructor. As a subset of C++ supported for shader code is relatively small, the resulting shaders are relatively easily converted into GLSL and passed to graphics hardware. It allows to specify shader inputs and outputs directly in C++ code.

Sh did not aim at providing a very high-level abstraction of a GPU; instead, it provided a simple library supported on a large number of platforms, which hides tedious calls to OpenGL behind easier C++ concepts. It does not, however, provide array programming abstractions or higher-level abstractions, and shaders for various architectures need to be coded individually.

The next generation of GPGPU libraries started with Accelerator [15] library developed at Microsoft Research by Tarditi *et al.* The idea behind the library was to provide a library of array programming primitives which are executed on GPUs. The class of a 2D GPU-stored floating-point array, together with arithmetic and boolean operator overloads for it form the basis of the library. As GPUs require computationally-intensive shaders in order to run efficiently, the array operators are not executed immediately; instead, they are executed lazily, and a tree of array operations is formed. When the resulting array is read back, the tree is analyzed, and a sequence of shaders is generated for it. This sequence of shaders is then executed, and the result is read back. A number of tests for the Accelerator library has been written, which demonstrated that Accelerator-generated code is typically within 50% efficiency compared to hand-written shader code, being still up to 18 times faster than the same code executed on CPU.

The first version of Accelerator provided only simple arithmetic and relational operators on arrays. The second version already provides whole-array operations, such as inner and outer products together with array shifts and rotations, which allows it to be used in practical applications, such as image processing or linear algebra operators.

The Accelerator library is a research project, and is freely available for non-commercial applications. A number of similar commercial approaches exist, such as RapidMind [16] and PeakStream [17]. They use an approach similar to Accelerator, combining array programming with lazy computations. Unlike Accelerator, those are commercial libraries. They also provide support for platforms other than GPUs, such Cell BE or multi-core CPUs.

CUDA [18] is an implementation of a subset of C language which allows to write programs using both GPU and CPU. It has been designed by NVIDIA for its GeForce 8 series GPUs, and works with these cards only. CUDA adds a set of specific libraries to manipulate NVIDIA GPUs. These calls in fact resemble library calls used in GPGPU libraries to set up and execute kernels. In addition, it provides implementations of subsets BLAS and FFT libraries, which can be used to provide familiar programming interface for programmers.

The success of the GPGPU programming approach with the use of array programming combined with lazy computations motivated us to take a similar approach to GPGPU programming. However, integrating array programming into a language which does not support it natively causes a number of conceptual inconsistencies. First, for each array type and dimensionality, a separate class should be declared in the language, which requires writing redundant code, together with defining each of the operations two times: once for arrays and once for simple types. Second, while it is possible to override arithmetic and relational operators for such arrays, it is impossible to override conditional and loop statements as well as conditional operators, which leads to inconsistencies in expressing control structures in shader code and in CPU code of the program. Moreover, pure application of arithmetical operators to arrays in not always the best way to perform array operators; it is often easier (e.g., for ray tracing) to express GPU algorithms as functions without side effects applied to arrays, which is not provided by existing approaches. Finally, pure dynamic approaches lack static code analysis, which could provide clues for better memory representation and breaking the expression tree into shaders.

III. Our Language for GPGPU Programming

We therefore opted to design a language which would provide primitives for GPGPU programming, which we called C$ [19]. In the language, we attempted to combine features of C# language together with array programming primitives which would allow higher-level GPGPU programming. Such an approach is quite common for modern parallel programming languages, such as Titanium [20] or Chapel [21], so we follow it. Technically, our language is based on .NET, and array and functional types are translated into .NET interfaces. On the one hand, it allows better integrating our language with existing languages, such as C#, and thus write whole GPGPU programs together with I/O and user interaction with a single language. On the other hand, this would allow to develop a language and a library simultaneously, and use the programming library from other .NET languages.

In our language, we decided to merge arrays and functions; thus, an array is a special case of a function. This is achieved by interpreting functional types as .NET interfaces, which can be implemented by various classes which represent GPU-mapped arrays, functions associated with methods etc. Although the idea is not new, it seems logical in the context of lazy computations, which is more common for functional programming. This also makes possible better integration of arrays with other data structures, such as maps; we also allow for user-defined classes to inherit from functional types, which allows to represent files, database queries etc. as functions, integrating them with lazy evaluation framework and potentially reducing the amount of reads from files.

Merging arrays with functions also allows to express such operations as indexing of arrays with arrays as a more natural operation of superposition of functions. It also allows to avoid introducing a separate type of index sets and regions into the

language. Each function in C$ has a domain (possibly finite or infinite), which is a special function of type **int** => T, where T is the type of the argument tuple of a function.

For each standard and user-defined function, functional and reduction promotions are applied. Assume there is a **float op**+(**float** a, **float** b) addition function (which is, of course, a standard one). It is then allowed to pass to it as arguments functions (or arrays) which return floating-point values, and obtain a new function as a result, which takes arguments of the first two functions, and returns the sum of their results. It is also possible to pass to the + function a single function returning a floating-point value, and reduction operations shall be performed on the entire set of values returned by this function.

In order to perform lazy computations, facilities for side-effect specification are needed, which are introduced into a language in the form of a standard **void** keyword, which means a function having side effect in our language. We also use C# structural types (which we rename here to final classes with certain limitations) with immutability specification to support simple elements of OOP in high-performance computations.

As each function has a domain associated with it, it is possible to deduce the range of values of loop variables rather than explicitly specifying it. So we introduce a concept of a *bound variable* into our language, which can conceptually be understood as a loop variable of a parallel loop without any loop header. For example, this is how dense matrix multiplication is expressed with our language:

```
type matrix = float (int, int); // matrix functional type alias
class Utils {  // no global functions or variables
   matrix mul(matrix a, matrix b) {
      matrix c(j, l) = + (a(j, k) * b(k, l));
      return c;
   }  // end of mul
}  // end of class Utils
```

Here, a matrix is treated as a floating-point function of two integer values. *a*, *b*, and *c* are variables, while *j*, *k* and *l* are bound variables. The range of bound variables is inferred dynamically from the domains of *a* and *b* functions, the same is true for the domain of the *c* variable. Because *k* occurs in both *a* and *b* indexing (which can be done with brackets as well), respective dimensions of *a* and *b* matrices are run though simultaneously. The * operator performs element multiplication, while the + actually performs reduction by all bound variables declared inside it and all non-bound dimensions (variables *j* and *l* are declared outside the reduction, and reduction is not performed by them, while *k* is declared inside, and the expression is reduced by it). Bound variables are allowed to be used only in functions without side effects.

This makes it relatively easy for the language compiler to extract parallel segments of the program. Parallel segments are those in which either bound variables or functional or reduction promotion is used. The main source of parallelism in the language is *evaluation of a function without side effects on its entire domain*. The parallel segments thus perform lazy evaluation and construct functional trees, which are compiled and translated at the *points of actualization.* Such points can also be defined relatively easy as points where the constructed function is passed as an argument to a method with side effects, such as methods which save results of computations to files, or points where a function is converted to an array (an array, unlike a function, is understood strictly imperatively, and is always associated with an object inside backend memory).

IV. APPROACHES TO TRANSLATION OF C$ LANGUAGE

The proposed structure of the C$ system is outlined in figure 1.

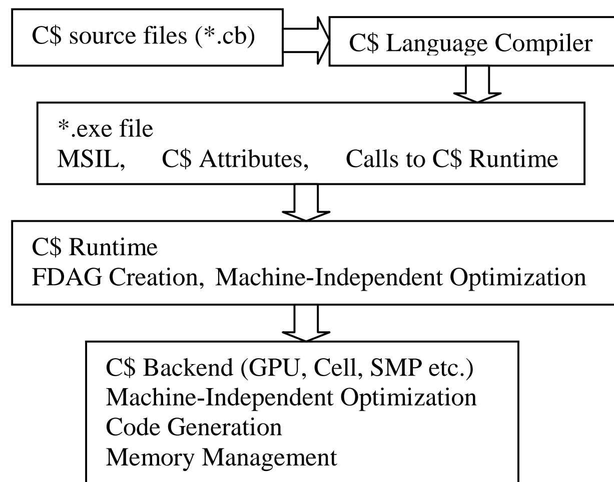

Fig. 1. The structure of C$ system.

which builds .NET assemblies. An assembly shall contain the MSIL code, the classes and attributes which reflect properties of C$ entities otherwise not representable in .NET (such as whether or not the function has a side effect etc.). Segments of parallel code are actually translated in calls to C$ runtime which is responsible for lazy computations.

At the runtime, the DAG of lazy functional computations (FDAG) is built. Its leaves contain non-functional values, functions (arrays, maps, member functions etc.) and bound variables, while its inner nodes contain functional operations (apply, reduce etc.). This allows us to treat standard and user-defined functions alike. After the evaluation of the DAG is requested, runtime optimization (such as common subexpression elimination, constant propagation, reordering etc.) is performed, and the DAG is passed on to the backend.

The backend is responsible for memory management, code generation for target architecture, and machine-dependent optimization. The DAG is executed at the backend, and the result is made accessible to the runtime.

Note that a backend is responsible for storing arrays used in high-performance segments of the program (while .NET memory management is used for other objects). When a program issues a request to allocate a memory for a functional object (array, map etc.), it is checked whether the backend can serve the request. If it can, the memory is allocated on the backend, and further computations are also performed there. If not, the memory is allocated by .NET and the code is actually interpreted in runtime.

Currently we mainly target GPUs as backends. We're planning to have several GPU backends (one for DirectX, one for OpenGL, one for ATI boards and one for NVIDIA ones) in order to get the best of each of the boards. Although the exact format of the FDAG and backend-runtime interface is currently under development, we're planning to make this interface open in order for other developers to be able to write backend for other architectures, such as Cell BE or multi-core.

V. Current State of Work and Discussion

We have currently implemented a simple interpreter for a subset of C$ language, in order to be able to perform backend development together with the development of the first version of language specification and the language compiler. We've also implemented a backend for ATI Data-parallel virtual machine (DPVM) [22]. We're currently able to translate simple expressions with floating-point arrays and reductions; we do not currently support bound variables.

The work is currently going on with writing a draft language specification together with language compiler to .NET (currently we compile the language into custom intermediate representation, which shall be deprecated). The work is also going on with implementing a DirectX backend. The first version of the draft language specification will be available soon. We're also starting development of multi-core backend soon.

We think that, given .NET integration, a C#-like language for GPGPU programming and a number of backends, at least for GPU and multi-core, will make the system usable enough, and will contribute to the popularization of GPGPU programming.

References

[1] *GPGPU.org.* http://www.gpgpu.org/.

[2] *Linear algebra operators for GPU implementation of numerical algorithms.* **Kruger, Jens and Westermann, Rudiger.** Los Angeles, California : ACM Press, 2005. International Conference on Computer Graphics and Interactive Techniques.

[3] *GPUCV: A framework for image processing acceleration with graphics processors.* **Farrugia, J. P., et al.** Toronto, Ontario, Canada : IEEE, 2006. IEEE International Conference on Multimedia & Expo (ICME 2006).

[4] *A multigrid solver for boundary value problems using programmable graphics hardware.* **Goodnight, Nolan, et al.** San Diego, California : Eurographics Association, 2003. SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware. pp. 102 - 111. ISBN ~ ISSN:1727-3471 , 1-58113-739-7.

[5] *Ray tracing on programmable graphics hardware.* **Purcell, Timothy J., et al.** Los Angeles, California : ACM Press, 2005.

[6] OpenGL.org. http://www.opengl.org.

[7] **Kessenich, John, Baldwin, Dave and Rost, Randi.** *The OpenGL Shading Language.* September 7, 2006.

[8] **ATI Technologies Inc.** *Radeon X1800 Shader Architecture.* 2005.

[9] **IBM Corporation.** *Cell Broadband Engine Architecture.* s.l. : IBM Corporation, October 3, 2006.

[10] *Brook for GPUs: stream computing on graphics hardware.* **Buck, Ian, et al.** Los Angeles, California : ACM Press, 2004. pp. 777 - 786.

[11] **Apple Inc.** Mac OS X Core Image. http://www.apple.com/macosx/features/coreimage/.

[12] *Mapping computational concepts to GPUs.* **Harris, Mark.** Los Angeles, California : ACM Press, 2005. ACM SIGGRAPH 2005 Courses.

[13] *KD-tree acceleration structures for a GPU raytracer.* **Foley, Tim and Sugerman, Jeremy.** Los Angeles, California : ACM Press, 2005 . SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware. pp. 15 - 22. ISBN:1-59593-086-8.

[14] Sh Web Site. http://libsh.org/.

[15] *Accelerator: using data parallelism to program GPUs for general-purpose uses.* **Tarditi, David, Puri, Sidd and Oglesby, Jose.** San Jose, California, USA : ACM Press, 2006 Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 325 - 335. SESSION: Embedded and special-purpose systems.

[16] Rapid Mind Inc. Web Site. http://www.rapidmind.net/.

[17] PeakStream Inc. Web Site. http://www.peakstreaminc.com/.

[18] **NVIDIA Corporation.** NVIDIA CUDA Complete Unified Device Architecture. February 12, 2007.

[19] **Adinetz, Andrew V.** C$ Project Web Site. http://www.codeplex.com/cbucks.

[20] **Bonachea, Dan, et al.** Titanium Language Reference Manual version 2.20. [ed.] P. L. Hilfinger. August 2006.

[21] **Cray Inc.** Chapel Language Speicifcation 0.702. s.l. : Cray Inc., 2005.

[22] *A performance-oriented data parallel virtual machine for GPUs.* **Peercy, Mark, Segal, Mark and Gerstmann, Derek.** Boston, Massachusetts : ACM Press, 2006 . ACM SIGGRAPH 2006 Sketches. ISBN:1-59593-364-6.