Jacket:
GPU Computing without CUDA programming

Summary

- In this paper, we describe a software platform, Jacket, for the rapid development of general purpose GPU (GPGPU) computing applications within the M language, more specifically the MATLAB® computing environment.
- In addition to providing low-level GPGPU graphics and computing capabilities, the platform includes a JIT compilation mechanism that translates M code to GPGPU machine code on demand. This arrangement allows users to interact with the *M*-language as they normally would (whether via the command-line or scripts/functions) while their computations are transparently compiled for and executed on the GPU as required.
- We show that the standard API provided with MATLAB can be utilized to convert entire M programs to run on the GPU by merely applying a ’g’ prefix to memory allocation commands.
- Finally, the platform also provides a path through which users can transform their M projects to C/C++ while retaining GPGPU capabilities after prototyping their GPGPU algorithms in the MATLAB environment.

Introduction

Over the past few years, specialized coprocessors from floating point hardware to field programmable gate arrays have enjoyed a widening performance gap with traditional x86-based processors. Of these, graphics processing units (GPUs) have advanced at an astonishing rate, currently capable of delivering over 4 TFOPS of single precision performance and over 300 GFLOPS [3] of double precision while executing up to 240 simultaneous threads in one low-cost package. As such, GPUs have gained signficant popularity as powerful tools for high performance computing (HPC) achieving 20-100 times the speed of their x86 counterparts in applications such as physics simulation, computer vision, options pricing, sorting, and search [13], [14].

Programming GPUs for scientific applications in general remains a difficult task. Currently, there are several paths programmers may take to interface with the GPU for general purpose computing, each of which require assimilation of new programming paradigms and/or APIs. However, each solution is essentially a library, middleware, or extension onto C or C++, both compiled languages which are not inherently parallel.

Invented in the 1970’s by Cleve Moler, MATLAB (*Mat*rix-*Lab*oratory) is an extensible interactive programming environment for numerical analysis built on a vector language called *M* [2] that has become almost ubiquitous in the scientific computing community with more than 1 million users [1] worldwide. The *M*-language, like other vector languages, provides users with a high level interface at which operations may be specified over large sets of data at once (foregoing iterators) making the expression of data-parallel algorithms natural. *M* is also dynamically typed, adheres to pass-by-value semantics, and is integrated into a well-developed interpreted environment. With these characteristics, *M* has proven to be a powerful, user-friendly language.

We show how the *M*-language and MATLAB almost transparently adapt to GPGPU computing through the description of the Jacket platform for the rapid development of GPGPU computing applications within the MATLAB computing environment. Unlike other GPU programming solutions, Jacket provides GPU computation and graphics ability from a language which is *inherently parallel* and *interpreted*, thereby providing a standard, extensible, and simple method of programming for the GPU in an already-proven rapid prototyping environment. We show that through the addition of GPU-specific data types to the M language with overloaded operators, entire CPU-bound M programs can be converted into GPU-enabled M programs through as little as adding a ’g’ prefix onto each memory allocation command. Otherwise, the user interacts with MATLAB as they normally would either from the command line or when running scripts. Pre-existing programs need no other modification (if they require modification at all).

Typically, MATLAB applications achieve native machine performance and hardware access through the construction of so-called

“MEX routines”: precompiled modules built from specialized C, C++, or FORTRAN source that are called as functions from the MATLAB interpreted environment. When a routine is called, the interpreter makes input data available to that routine from the MATLAB environment, concedes control to the routine, and expects computation results immediately upon termination of that routine. This facility is the only method through which MATLAB may call external system libraries and, therefore, interface with GPU hardware. Thus far, all attempts to link MATLAB to the GPU have utilized this approach [6].

However, the requirements of this method strike at the very caveat of GPU programming: the cost of memory transfer to and from the GPU is the key bottleneck for performance. The fact that each MEX call will force at least two memory transfers means that large strings of computations must be carried out entirely in one MEX routine in order to reach the full performance potential GPUs have to offer. This, in turn, forces all GPU programming and even higher-level algorithm development to be done in C, C++, or FORTRAN almost defeating the purpose of GPU for M development entirely.

To overcome this problem, we propose a platform approach to GPU programming with a lazy compilation scheme: operations on data are aggregated and only executed when absolutely necessary (such as when a user requests to display data). Concretely, our solution involves the introduction of several new classes into the MATLAB environment that mimic native MATLAB classes, but in fact, utilize the GPU for computation instead of the CPU. These classes overload the built-in set of base MATLAB functions which provide full GPU computation functionality in the MATLAB environment through polymorphism. This set of functions comprises a run-time to balance computation between the GPU and CPU: memory transfers, on-the-fly compilation, instruction caching, garbage collection, thread extraction, etc. Through this arrangement, the requirement for memory transfers is alleviated and the opportunity for end-to-end optimization of MATLAB programs for GPU execution is exposed.

Central to our contribution is the fact that MATLAB is a closed-source and interpreted environment with many features and behaviors beyond C and C++. Unlike other GPU-programming systems where almost universal knowledge of program state is known, the only information our system has access to is the limited state information available through the MATLAB API. We show several key algorithms and a data structure design that avoids this hindrance and opens the door to the addition of GPU capabilities to other interpreted environments, including MATLAB, without modification to the language or underlying environment.

Practically, however, due to the inherent parallel nature of the *M*-language and its function override behavior, programs may be transparently ported to the GPU from within MATLAB through only the use of GPU-bound data types much in the way sparse matrices are already supported. This arrangement also allows pre-written routines written with lower-level GPU programming methods to be integrated into Jacket. Several examples are provided on our website including partial differential equations, machine learning, and Fourier analysis -- many of which can run up to 75 times faster using Jacket.

Finally, the system gives the user an interactive way of utilizing the GPU for computation and visualization which has not been available before, all within the standard MATLAB API already adopted widely in the scientific computing community.

Related Work

Several efforts have been made in the past to bring parallel computing to MATLAB for the execution of scientific applications over distributed computing systems such as clusters [12]. However, unlike these approaches, our focus has been to maximize performance for non-distributed processing in MATLAB to facilitate the rapid development of specialized applications requiring near-real-time performance or prototyping at a high scale such as data-visualization, control, graphics, etc. Since initial product release, Jacket has become available for distributed applications given the amount of interest expressed in the market.

Also, through the inclusion of a load-sharing OpenGL API in our system, we capitalize on the fact that, unlike a distributed machine, the GPU is a device built for graphics synthesis capable of computing while visualizing at tremendous speeds. Ultimately, unlike previous projects, the purpose of Jacket is the development of software to be deployed to end-users with consumer-level hardware to make possible not only interactive, but also personal high performance computing.

Additionally, there have been several past efforts aimed at improving the accessibility of GPU’s to programmers for general purpose computing. These range from low to high-level and expose varying computation abilities and usability. Geared towards earlier hardware, the lowest-level method of achieving GPGPU computing is via extension and specific usage of drawing API's such as OpenGL or DirectX such that rendering graphics to a floating-point image reduces to computation [15]. This method relies on the construction of specialized routines known as either vertex or fragment shaders, clever rendering sequences, or memory layouts for the GPU.

Higher level approaches fall into three categories: stream computing, data-parallel, and parallel thread. Stream computing approaches, which include projects such as Brook for GPU's [8] and Sh [10], abstract data as a series of large blocks which are operated on by kernels with very little flow control. Both projects extend the C and C++ language with a modified build tool chain such that kernels are defined inline with regular C(++) code that orchestrates overall execution.

Other approaches including Acclerator [9] take a data-parallel approach where embarrassingly parallel computations and transforms are exposed as methods on GPU-specific classes in object oriented languages such as C# and C++. This approach abstracts away the kernels of stream oriented solutions by compiling vertex and fragment shaders transparently at run-time based on accrued operations on GPU data structures.

Finally, thread-based approaches wholly include the Compute Unified Device Architecture (CUDA) [3], which strives to move GPGPU computing beyond the composition of embarrassingly parallel operations on large blocks of data by exposing the GPU as a device capable of executing a large number of possibly independent threads in parallel, each communicating via specialized memory. As with other approaches, however, this is an extension upon C/C++. It is ultimately a low-level approach, albeit powerful enough to facilitate complex algorithms such as BLAS3 [4] and the FFT [5] more efficiently than all other approaches, thus far.

Jacket serves as a high-level method of programming the GPU utilizing an approach similar to that of data-parallel approaches, but exposing methods up to the full capabilities that MATLAB provides as primitives. Additionally, due to the function override behavior of MATLAB and the parallel nature of *M*, the approach of adding new GPU-specific classes to MATLAB while overloading operators allows M code to be ported to the GPU with no other code changes other than a change in data primitives (prefixing memory allocation functions with a 'g'). The approach is also the first interpreted solution of GPGPU programming with a fully integrated visualization package included, allowing user-friendly, interactive personal supercomputing.

Background

A. GPU Architecture and Programming Model

A GPU is a highly parallel computing device designed for the task of graphics rendering. However, the GPU has evolved in recent years to become a more general processor, allowing users to flexibly program certain aspects of the GPU to facilitate sophisticated graphics effects and even scientific applications. In general, the GPU has become a powerful device for the execution of data-parallel, arithmetic (versus memory) intensive applications in which the same operations are carried out on many elements of data in parallel. Example applications include the iterative solution of PDE's, video processing, machine learning, and 3-D medical imaging.

With the release of nVidia's Compute Unified Device Architecture (CUDA), the hardware architecture of modern GPU's can be viewed as a series of Single-Instruction-Multiple-Data (SIMD) multiprocessors, each capable of processing one set of instructions on different elements of memory in one clock cycle. As of this writing, up to 240 cores are available on a single GPU, grouped onto a series of multiprocessors. Each of these cores has available to it four sources of local memory within its multiprocessor unit:

- A set of registers for each processor
- Shared memory available to all processors
- Shared constant memory available to all processors
- Read-only texture cache available to all processors

Beyond the scope of each multiprocessor, read write access is available to the local video memory present on the GPU. Texture memory is considered a special case of device memory and is accessed by each processor via a texture unit which applies any special filtering or addressing requested depending on memory data format. Typically, the GPU is accessed for computation or graphics rendering through its device driver via either a graphics API (such as OpenGL or DirectX) or through specialized APIs such as CUDA (which shares the same name with nVidia's hardware architecture). GPU computing with CUDA is a more modern and capable GPGPU computing approach than utilizing the former two APIs, so in this paper we focus on CUDA and utilize its terminology and programming model from here on out.

CUDA is a software architecture and API geared towards the utilization of the GPU as a computing device rather than a graphics rendering device. The CUDA software includes a GPU device driver, a runtime system that serves as an abstraction over the driver, and also runtime libraries that CUDA applications may link to in order to provide GPU-enabled FFT and BLAS support, among others. CUDA also includes a compiler tool-chain which provides extensions onto the C and C++ languages for the construction of GPU applications. While programming the GPU with the CUDA tool chain, the GPU is viewed as a coprocessor to the CPU, or host, which orchestrates the executions carried out by the GPU as needed. In order to utilize the GPU to its fullest potential, the CPU must minimize data communication with the GPU (due to limited bus bandwidth) and maximize data parallelism in the tasks given to the GPU (to maximize usage of GPU processors). Though the GPU can be viewed as capable of executing a large number of general threads in parallel thanks to CUDA, GPU programming is still typically accomplished through the specification of kernels which operate across an array of data elements such is done in the following CUDA-extended C-code:

```
¨global¨ void sign˙kernel( float *b, float *a ) –
b[ blockIdx.x ] = sign( a[ blockIdx.x ] );
˝

void do˙sign( float *host˙src, float *host˙dest ) –
float *a, *b;
a = cudaMalloc( 100 );
b = cudaMalloc( 100 );
cudaMemcpy( a, host˙src, 400, cudaMemcpyDeviceToHost );
sign˙kernel¡¡¡100,1¿¿¿( b, a );
cudaMemcpy( host˙dest, b, 400, cudaMemcpyHostToDevice );
˝
```

In the example above, two functions are declared: *sign kernel*, a function which executes on the GPU and is callable from both the GPU and the CPU (specfied by the *global* keyword), and *do sign*, a function which resides on the CPU and is only callable from the CPU. Overall, *do sign* computes the sign of the 100 element array *host src* and places the result in *host dest*.

Above, all major aspects of GPU programming are illustrated. First, programs to be executed on the GPU must be specified as kernels (known in OpenGL and DirectX as shaders). These kernels are limited in their length and the amount of local memory they use. Second, GPU programs, or kernels, must operate on memory that currently resides on the GPU. This is insured in *do sign* by allocating GPU memory and copying *host src* to *a* via *cudaMalloc* and *cudaMemcpy*. Third, as stated previously, the CPU must orchestrate GPU executions as is done in *do sign* with the *sign kernel* call. The number of threads that will run the kernel is given by a grid-size specifier, in this case 100 threads by 1 thread giving 100 threads in all to process the entire array, *a*. Finally, in order for the CPU to utilize the results of GPU executions, the results must reside in the CPU's memory. This is insured by the final *cudaMemcpy* command in *do sign*.

The example also illustrates the bottlenecks involved in computing with the GPU: memory allocation, memory transfer, and kernel execution. In the most ideal situations, each of these tasks is done very occasionally to ensure that minimal overhead is accrued over the lifetime of an application. Jacket, which we describe in the following sections minimizes these tasks transparently and yields high GPU/CPU performance for MATLAB applications.

The M-language

M is an object-oriented vector programming language that is particularly well-suited to parallel programming. In M, users may manipulate typed data and arrays within MATLAB through iterators (such as *for* and *while* loops), but it is recommended to perform executions in vectorized form in which case iterators are avoided and the M interpreter may perform vectorized optimizations. Vectorized computations are achieved in the M-language through the use of object methods that operate on entire arrays at a time and vectorized indexing which allow multiple elements of arrays to be accessed and then operated upon. Additionally, through the use of the colon operator, entire dimensions of arrays may be selected for computation. Possible computations may be anything defined as M-code, but a standard set of commands are made available through the MATLAB interface which are widely utilized by the MATLAB programming community. The M-language also supports objected oriented programming up to the definition of classes, class instances, object fields, and object methods. Class inheritance is also supported along with polymorphism for correct method selection for a particular object.

Mapping M to the GPU

Our approach to extending the M-language to support GPU programming is to define a new set of classes dubbed *g* objects where each element of this set corresponds to a base class of the MATLAB standard interface: single, uint16, ones, etc. map to gsingle, guint16, gones, etc. These new classes function exactly as their CPU-based counter parts. In order to facilitate this functionality, each standard method currently present on the standard classes is made available on the *g* objects via GPU-enabled mex code, the architecture of which is described below. The set of standard methods include basic functionality such as displaying the object in text to assigning to the variable, but also much more involved functionality such as the fast fourier transform, matrix multiplication, singular value decomposition, etc. Since a standard MATLAB environment and thus programmer expect these functions to be present by default, large portions of already existing MATLAB programs can be run on the GPU in this manner by only changing data types from their base class such as *single* to the GPU base class equivalent *gsingle*.

As with programming the GPU in C or C++, care must be taken by the programmer to ensure minimum memory transfer to the GPU and maximum data-parallelism i.e. the maximum number of homogeneous operations are performed on the maximum number of data elements at a time. As discussed, above, it is preferred that MATLAB code be written in vectorized form in which large parts of data are operated upon by only a few operations in parallel. This vectorized paradigm of programming is exactly the style of programming necessary to meet the data-parallel requirement of GPGPU programming for maximum performance. Thus, if a piece of MATLAB code is written in conforming style to the vectorized paradigm (which most programmers do), then the code is already nicely designed for data-parallel execution. However, in some instances, certain computations are not expressible in vectorized form. In this case, automatic thread extraction must be utilized.

Unfortunately, a major feature of the M-language which makes the language quite suitable for beginning programmers and prototyping algorithms, but bad for GPU computing is M’s standard convention of utilizing pass-by-value semantics: copies of objects are passed to functions instead of references. Thus, in many instances, multiple copies of objects are made throughout the execution of a MATLAB program. This convention in the M-language thus makes it impossible to effectively utilize standard MATLAB classes with completely exposed data-stores for use with the GPU. Many copies of objects on the GPU would be created, inefficiently utilizing bus bandwidth and GPU memory. Additionally, as copies of objects are made, there is no way within the MATLAB environment to intercept these events, making it impossible to achieve coherence between MATLAB and GPU memory state.

To bypass M's pass-by-value calling convention, the Jacket Architecture uses object-oriented-programming to handle references to data. Such objects retain information about the location, size, and type of underlying data in memory, as well as any computations that have been performed on this data.

Copy-on-write and memory state coherence

Unfortunately, even with the object oriented programming approach to interacting with the MATLAB environment and to carry out commands, memory state coherence between MATLAB and the GPU is still a problem. For instance, if a copy of a GPU object is created, there is no way to know if that object has been duplicated and that data currently stored by a referenced GPU/CPU data store must be maintained for correctness of the duplicate which may be required in another part of the program. For example, without the knowledge of duplicated GPU objects, it is unknown whether the third statement should write over the GPU object data of *A* on line 3, or to maintain a separate copy to maintain the expected MATLAB behavior of *B* on line 4 of the following example:

```
A = gones(3) +3;
B =A;
A = gones(3) *5;
B =B -2;
```

To solve this problem, we utilize MATLAB's copy-on-write optimization to detect such duplications: when a duplicate of a particular object is to be created during the execution of M-code, a duplicate is not created immediately. Instead, the duplicate (such as *B* in the above example) is made to reference the information contained in the original object (*A* from above). The creation of duplicate information is forced only when the duplicate object is written to (i.e. *B(2,2)=5*) in which case the data is copied and the assignment is affected. Jacket's design takes into full consideration these memory and data challenges to deliver results with minimal impact on users.

Lazy Execution, JIT and Garbage Collection

A. Computation Tree Construction

As operations on GPU and MATLAB objects are recorded over the course of a MATLAB program, no computations are carried out until absolutely necessary. This strategy encompasses the lazy execution approach of GPU computing with MATLAB. Instead of carrying out computations, new GPU objects are created which record the operations on other objects to aptly describe the resulting object data without actually carrying out computations on the GPU.

For instance, consider the following piece of MATLAB code:

```
A = gones(3) +3;
```

The events that ensue to build the computation tree which is eventually stored in the variable *A* are as follows:

1) The instantiator, gones, calls the Jacket architecture which returns a GPU object of type *constant*.

2) Since *gones(3)* is now replaced by a GPU object, the plus operator calls the plus method on that object with the object and the MATLAB variable 3 as operands.

3) A new GPU object representing a plus operator is created and *gones(3)* and *3* are made its children. The update dimensions of the new GPU object are computed based upon the children and the operation.

B. Computation Tree Evaluation

Actual computation is demand driven: the user wants to print or visualize the data on the screen. To render the requested values, Jacket examines and executes the recorded computations taking into account various considerations including the availability of memory, memory hierarchy latencies, cached instructions, system load, opportunities for parallelism, etc.

C. Garbage Collection

Since all variables contained within the computation tree are actually MATLAB objects, when a tree is disconnected from its parents and there are no references to the root of that tree in the MATLAB workspace, then the tree is considered orphaned and returned to the free memory heap automatically by MATLAB. Thus nodes within the computation tree need no extra cleanup or duplication checks in order to be cleared properly from memory after a computation has been performed. A method of reference counting is used to determine when it is safe to reuse GPU/CPU memory objects.

Conclusion

The emergence of new software tools and cost-effective GPU-based computers are ushering in a new era for scientists, engineers and analysts working on technical problems. The platform, Jacket, described in this paper offers a more productive platform for non-computer scientists looking for ways to leverage the computational capabilities of GPUs. Jacket utilizes the M-language, and the flexibility of MATLAB from The MathWorks, to provide much larger population of users the benefit of GPU technology. Jacket abstracts all of the complexity of GPU and CUDA programming from the user and addresses the underlying challenges associated with programming GPUs. The Jacket architecture takes into account all of the possible roadblocks associated with high performance GPU applications so that the technical computing user can focus on science, engineering or analytics.

About AccelerEyes

AccelerEyes launched in 2007 to commercialize Jacket, the first software platform to deliver productivity in GPU computing. With advanced language processing and run time technology to transform CPU applications to high performance GPU codes, Jacket extends desktop workstation performance and fully leverage GPU clusters. Based in Atlanta, GA., the privately held company markets Jacket for a range of defense, intelligence, biomedical, financial, research and academic applications. Additional information is available at www.accelereyes.com.

Index Terms— GPGPU, stream computing, data-parallel, Mat-lab.

Acknowledgements - References

[1] Goering, R., “MATLAB edges closer to electronic design automation world,” EE Times, Oct. 2004.
[2] The Mathworks, “The Origins of Matlab” Available at http://www.mathworks.com/company/newsletters/news notes/clevescorner/dec04.html
[3] nVidia, http://www.nvidia.com/object/tesla_computing_solutions.html
[4] nVidia, “CUDA CUBLAS Library 1.1,” Available at http://developer.download.nvidia.com/compute/cuda/1 1/ CUBLAS Library 1.1.pdf
[5] nVidia, “CUDA CUFFT Library 1.1,” Available at http://developer.download.nvidia.com/compute/cuda/1 1/CUFFT Library 1.1.pdf
[6] nVidia, “Accelerating MathWorks MATLAB with CUDA,” Available at http://developer.download.nvidia.com/compute/cuda/1 0/AcceleratingMatlabwithCUDA.pdf
[7] nVidia, “PTX: Parallel Thread Execution ISA Version 1.1”, Available athttp://www.nvidia.com/object/cuda develop.html
[8] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., “Brook for GPUs: Stream computing on graphics hardware.” *Transactions on Graphics and Visualization* vol. 23, no. 3,Aug. 2004.
[9] Tarditi, D., Puri, S., and Oglesby, J.. Accelerator: Using data parallelism to program GPUs for General-Purpose uses. In *International Conference on Architectural Support for Programming Languages and OperatingSystems* (2006)
[10] McCool, M. and Toit, S.D., *Metaprogramming GPUs with Sh*. A K Peters, 2004.
[11] Mark, W.R., Glanville, R.S., Akeley, K., and Kilgard, M.J. “Cg: A system for programming graphics in a c-like language.” *Transactions on Graphics and Visualization* vol. 22, no. 3, pp. 896-907, 2003
[12] Choy, R. and Edelman, A. “Parallel Matlab: Doing it Right.” *Proceedings of the IEEE* vol. 93, no. 2, pp. 331-341, 2005
[13] Matt Pharr, ed., *GPU Gems 2*, Addison-Wesley, 2005.
[14] Hubert Nguyen, ed., *GPU Gems 3*, Addison-Wesley, 2007.
[15] D. G¨oddeke, *GPGPU Basic Math Tutorial*, tech. report 300, Fachbereic Mathematik, Universitt Dortmund, 2005