Scientific GPU Programming with Data-Flow Languages

Daniel Goodman Mikel Lujan
The University of Manchester
Daniel.Goodman@Manchester.ac.uk

1. INTRODUCTION

Graphical Processing Units or GPUs are processors used primarily to render images from computer models for domains ranging from gaming to design engineering. As the generation of very accurate images often in real time is extremely computationally intensive, they have developed into extremely powerful processors. To achieve this they have relied on being able to specialise to this particular set of problems, specifically taking advantage of the fact that each pixel can be calculated independently. This has resulted in a processor with a high level of parallelism, and on which large numbers of threads run in small groups to compute independent results. These groups of threads take inputs and produce outputs, but there is no communication between groups. This lack of communication means that computations on GPUs are fundamentally controlled by the flow of data through pipelines as transferring control information between threads is not possible.

The level of performance offered by GPUs has for a long time been of interest to other areas of computation, and the development of the CUDA [8] programming language and later OpenCL [6] has provided a new means of programming GPUs to perform general purpose programming. While it had previously been possible to program GPUs to perform non graphics related computations, this was achieved through the use of domain specific shader languages [9] that required considerable skill to use for non graphics computations. The provision of a low level non-domain specific language greatly simplified the problem of programming GPUs. However, CUDA's low level nature, while saving the programmer from re-purposing a domain specific language to another unrelated purpose, does load the programmer with a lot of low level details that add time and complexity to the construction of these codes, and leaves them beyond many users. The effect of this can be seen in the number of domain specific languages that are now being produced for different areas of science that interface with GPU's [4, 5]. These are then augmented by more general purpose languages with differing levels of abstraction provided by companies such as the Portland Group [2] and MathWorks [10]. All of these languages though are derived from an imperative programming model where the user specifically describes the order that instructions are to be executed, instead of just describing the dependencies between instructions. This model was developed originally for single threaded applications, and then extended for multiple CPUs where coherency can be maintained between threads through communication. As a result of this extended evolution it is lacking in an intuitive way of handling the large levels of concurrency when communication between groups of threads is not possible. We argue that the restrictions on GPU's mean that they are better served by an alternative programming style known as data-flow programming [3, 7].

In this abstract we demonstrate that while CUDA is based on an imperative programming language, correctly constructed programs using CUDA's execution model map onto a coarse grained data-flow model. From this perspective we argue that the use of CUDA to simplify the programming of GPU's is a specific reoccurrence of the more general data-flow model. We then consider how languages for GPUs could develop into more fully featured data-flow language, and how this could further simplify the programming of these and other many-core devices.

2. CUDA PROGRAM DESIGN

Early CUDA-enabled GPUs could only execute one kernel at a time. While this restricted the class of algorithms that could be simply or efficiently executed on these cards, it also simplified the design constraints for programmers. With the early cards you simply needed to decompose your problem into a set of very large pieces of computation that could occupy the entire card, and ensured that any inefficiency due to the load balancing at the end of the kernel were minimal relative to the entire kernel execution time. This effectively turned the programs into a simple workflow where only one thing was happening at a time and if any branching in the control flow could be predicted the entire computation could be replaced by a pipeline without loss of performance. Coupled with the block structure of the kernels and the vector processing structure of the blocks, this model made an effective way of reasoning about problems from this restricted set.

The Fermi architecture made it possible to run multiple separately launched kernels at the same time, a pattern that has now been replicated by AMD with their Cayman architecture. This allows the time due to imbalances in the work load at the end of a kernel to be used on the next kernel to be executed, and for multiple small kernels to fully occupy the card. This removes the requirement for kernels to be large enough to occupy the entire card in order to get good performance. The relaxation of the requirements of the older cards does not prevent the construction of the more traditional styles of program, but it does raise the question of how best to take advantage of this new freedom to both reduce the complexity of kernels through an increased number of smaller kernels and to solve problems that previously were inefficient on GPU's. It is clear that in order for ordinary users to construct maintainable programs that truly take advantage of the ability to run multiple kernels, programming languages for GPUs, like languages for multi-core CPUs, are going to need to develop beyond their current relatively low

level.

3. CUDA PROGRAM EXECUTION

In this section we will look at how CUDA programs execute [8], identifying the different abstract memory types and how these interact with the computation. We will then go on to look at the data-flow programming model and how the interactions with these abstract types of memory map onto it.

A CUDA program consists of one or more kernels that are invoked on the GPU by the CPU. These kernels consist of one or more blocks of threads and in any real program there will be a large number of these blocks. Within each block there are a large number of threads that are able to communicate in order to achieve the overall aim of the block. The threads within the blocks are split into independent groups of threads called warps that execute in a SIMD model. As warps may execute at different speeds, communication within the block is achievable through the use of shared memory at synchronization points that provide a guarantee that every thread has executed this far. Away from these points, communication is only possible between threads in the same warp.

Blocks within a kernel are independent of each other and have no guarantees about when they will execute, which order that they will execute in, or even that two blocks will be executing at the same time. To handle this concurrency, normally blocks will maintain separation on outputs and will only share inputs. However, atomic statements can be used to ensure that interleaved of instructions do not affect the result written to areas of memory used by multiple blocks. But, because there is no guarantee that two blocks are executing at the same time, no form of complex interaction is possible. For example it is not possible to use the atomic statements for one block to communicate with another sending back and forth information about each other's computations. Instead, when it is necessary for a pair of blocks to communicate to successfully complete the computation, these blocks must be split into two separate kernels consisting of the computation before the communication and the computation after the communication. These are then invoked using two separate kernel calls from the CPU, and the information that the blocks wish to communicate is written out into separate memory locations by the first invocation and then read back in by the receiving block in the second kernel invocation. Semantically this splitting into two kernels is the same as the synchronization points within a block.

These restrictions mean that while GPU's have several types of physical memory such as shared, constant, main and texture memory used for different tasks, abstractly the memory used in a program can be grouped as follows:

Read only memory memory which can be read by one or more blocks in order to get their input. This memory will never change during the kernel invocation

Owner writable memory memory that can only be written to by a specific thread within a block, and will not be read or written to by other blocks during this kernel invocation or any other thread before the next synchronization point.

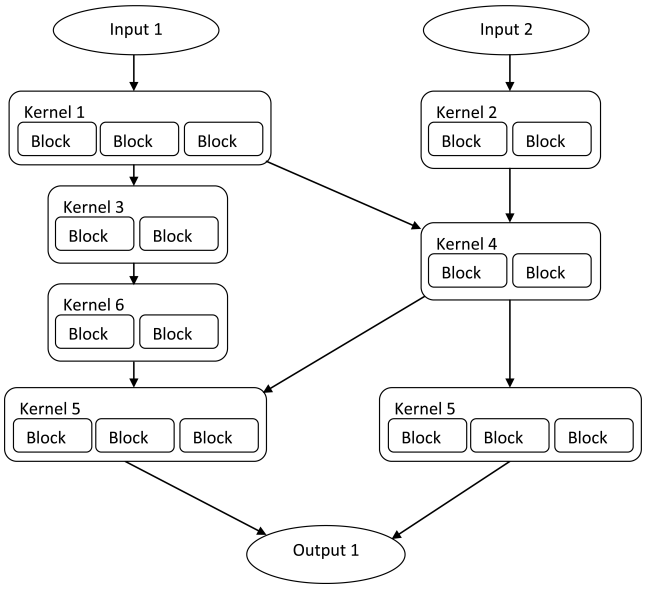
Atomic memory memory that is protected by atomic sections, so allowing writes from multiple threads at a performance cost.

Block local memory memory used to store temporary values used by a single block or thread within the block

It should be noted that owner writable memory and block local memory residing in the main memory can be allocated as a single call, and then divided algorithmically between many blocks. For example the first n bytes are used by block 0, the second n bytes by block 1 and so on. A table showing how and where these different abstract types of memory map onto the physical memories can be seen below.

Memory Type	Memory Usage
Main	Owner Writeable, Read only Atomic, Block Local
Shared	Block Local
Constant	Read only
Texture	Read only

The CPU is used to turn these kernel invocations into a workflow, keeping track of dependencies and deciding which kernel to execute next. This allows the scenario where the CPU will wait for several kernels to complete before starting another kernel, so providing the potential for complex synchronisation restrictions. Fermi provides further flexibility to this model by the ability to execute several smaller kernels at the same time. This allows for the creation of separate of groups of blocks within a computation based on parts of the computation that need to communicate, without concern about each kernel invocation containing enough computation to occupy the entire card. A dependency graph constructed from multiple kernels can be seen below, showing how different kernels can run concurrently and depend on each other's results



4. DATA-FLOW LANGUAGES

Originally developed in the late 1970's and early 1980's, data-flow languages [3, 7] were a moderately successful means of programming parallel computers. Their mainstream use was ultimately curtailed by the continuing development of faster single threaded processors making the need to write parallel code for the majority of applications superfluous. Now that physical limits have started to be reached forcing the move onto parallel computing, these languages are being looked at once more in order to program future multi and many core devices. Examples include the work on libraries such as LAPACK [1].

Unlike control flow programs or imperative programs that view a program as a sequential list instructions that must be followed, a data-flow program consists of either dynamically or statically generated graph, where the nodes are a fragments of sequential code that can be run in parallel by separate threads and edges are data dependencies. Each of these fragments has a set of inputs that are required before it can begin executing and a set of outputs that it generates. Once all the required inputs for a fragment of code have been generated its associated thread can be passed to the scheduler for execution. So the organisation is controlled by the flow of data through the program, not the flow of control. Aside from the data dependencies there is no guarantee of the order that the threads will execute in and multiple threads may make use of a single output from an earlier thread. This means that it is necessary for these threads to be functional in their construction. That is, that for a given set of inputs the output of the thread is always the same, and there are no side effects. Collectively this means that once a piece of data has been written to it will remain the same for the entire execution of the program. This functional purity prevents race conditions and so allows the code to execute in parallel without the programmer providing further details about the algorithm. However, the removal of mutable memory restricts the class of programs that can be described, and the way programs can be described. For example while loops have to demonstrate the same semantics as recursion, and systems such as a booking service for an airline that allows multiple concurrent bookings are not possible. The restriction to loops is trivially overcome by allowing mutable thread local memory within a thread's code, and only making it read only if it is passed out of the thread to be used by other threads. The construction of a parallel booking system is more complex. This restriction is as a result of the need to have a single piece of data describing the current state of all bookings. This single piece of data can then only be modified by a single thread at a time. To overcome this restriction it is necessary to add shared mutable state. This addition breaks the model as now the ordering of the execution of the segments of code can affect the result, however as long as all orderings still generate correct although possibly different results, the program will still be correct. Shared state also adds the potential for the state to be changed by one thread while another thread is executing, so protection for mutable areas of memory that are shared between threads is required. This protection can be provided by either the addition of locks, or the use of transactions, but the overall effect is there are areas of memory that are restricted such that only a single thread may access them at a time when they are being modified. When these areas are only being read from there is no need to apply restrictions to the number of accessing threads.

This leaves us with the strikingly similar set of four types of memory:

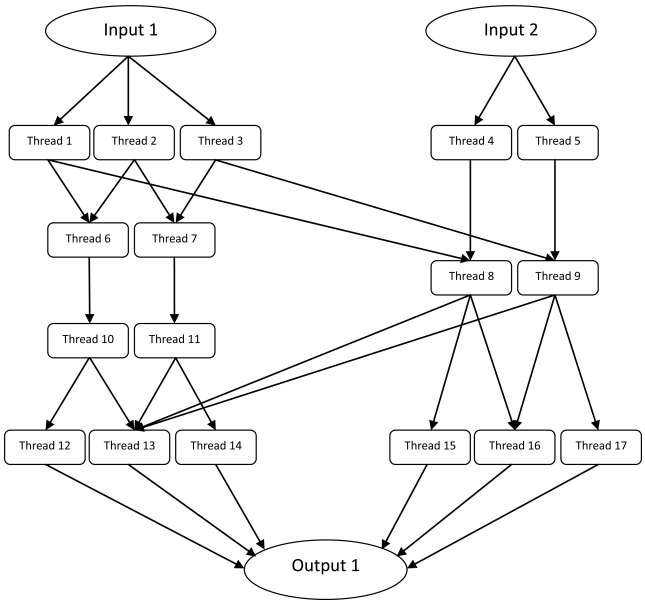
Read only memory memory which can be read by multiple threads in order to get their input.

Owner writable memory memory that can only be written to by a specific thread, and will not be read by other threads during this thread's execution. Once this thread has executed this memory can become read only memory.

Atomic memory memory that is protected by atomic sections, allowing multiple threads to safely modify it, but potentially at a performance cost.

Thread local memory memory used to store temporary values used by a thread.

Once again the scheduling of the computation is based on the data dependencies, not the control flow, although in data-flow languages this is overt. Taking this view, the dependency graph from the earlier CUDA program may convert to the following data-flow graph.



5. LIFE BEYOND CUDA

The similarities illustrated raise an interesting possibility using a data-flow model to develop future languages for programming both single and multiple GPU's, and in the not too distant future hybrid CPU-GPU chips. The development of such a language would be able to take advantage of the historical and current work on data-flow languages. Unlike the frequent observation about the similarity of programming within a block to historic vector processing, this

will not only allow for the reuse of algorithms and algorithm development techniques, but will provide insight into and a starting position for the languages that might replace CUDA and OpenCL as low level languages for many-core machines or augment them as a high level language. Such languages would simplify the invocation of kernels, the handling of data, and creating the potential for better correctness checking.

From this point of view let us now consider what a future high level data-flow GPU programming language may look like. Features that such a language may include are:

- Managed Memory
- Abstracted Kernel Invocations
- Complex Type System

A language supporting these could either be achieved by building a new language from scratch, or it could be achieved through the extension of an existing data-flow language. While making a fresh start may in many ways be the better option, it requires a large amount of work, where as the second option can be incrementally added. Taking the incremental approach, the first step would be to integrate the ability to call kernels into an existing data-flow language, with the scheduler automatically handling the data transfers much like in other higher level GPU languages. This would provide the ability to call kernels as and when their data becomes available, without the user having to handle scheduling, allowing a GPU execution to form parts of a data-flow graph. The construction of the kernel code can be achieved either through the direct use of CUDA, or through an adaptation of the data-flow language and a compiler that compiles down to CUDA code. Given the restrictions of kernel code include the absence of recursion, constructing languages that map onto CUDA and provide all its functionality should not be problematic and will provide the potential to add additional support for types etc.

This integration can then be furthered by changing the description level to that of specific blocks instead of specific kernels leaving the compiler or runtime to group these into kernels for execution. This would require the system to determine the appropriate parameters with which to call the created kernels. However, as there is already a great need for the addition of auto-tuners to simplify the selection of optimal parameters for kernel invocations, this could be more of an opportunity than a drawback. Within the block, the construction of the language extensions would need careful consideration in order to protect the vector efficiency of the warps, but the prize for this construction would be the automatic detection of the required locations of synchronization points.

The stronger semantics gained by the construction of a higher level language can then provide the opportunity for the language to be integrated with a more advanced the type system. This coupled with the explicit passing of data between data-flow threads mean that it is now possible to produce a reference counting managed memory for the GPU. The managed memory can be included in with the scheduler to ensure that the variable amount of memory on across different cards is automatically handled. This removes from the user the challenges of ensuring that there are no memory leaks, and the passing around of data is not restricted.

6. CONCLUSION

In this abstract we have observed that GPUs are data-flow devices designed for the concurrent handling of the computation of large numbers of pixels without the need for coherency in the execution model. However, for scientific computing they are programmed using CUDA which is an imperative language developed from C. This results in many natural actions in the language being illegal, and forces the user to comprehend the difficult problem of how to write data-flow code in an imperative language. Instead we argue that it would be better to develop a data-flow language for scientific computation on GPUs. Such a language would not only make it easier for the user to program codes to run on these devices, but it would also allow for the construction of stronger semantics that could support type checking and memory management in an efficient way. Collectively this will make GPUs easier to program, more accessible and will blur the line between the GPU and the CPU.

7. ACKNOWLEDGMENTS

Many thanks to Chris Kirkham, Ian Watson, Salman Khan and Berham Khan for their thoughts. Dr. Lujan is supported by a Royal Society University Research Fellowship

8. REFERENCES

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[2] CUDA Fortran Programming Guide and Reference. *CUDA Fortran - Programming Guide and Reference*, 1.3 edition, May 2010.

[3] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10:349–366, December 1990.

[4] M. Giles and G. Mudalige. Op2: an open-source library for unstructured grid applications. In *2nd UK GPU Computing Conference*, December 2010.

[5] D. Grewe and A. Lokhmotov. Generating and automatically tuning opencl code for sparse linear algebra. In *2nd UK GPU Computing Conference*, December 2010.

[6] Khronos OpenCL Working Group. *The OpenCL Specification*, August 2008.

[7] R. S. Nikhil. The parallel programming language id and its compilation for parallel machines. *International Journal of High Speed Computing*, 5(2):171–223, 1993.

[8] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 edition, June 2008.

[9] R. J. Rost, A. Central, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language*. Addison Wesley, 3rd edition, 2009.

[10] G. Sharma and J. Martin. MATLAB: A Language for Parallel Computing. *International Journal of Parallel Programming*, 37(1):3–36, February 2009.