GPU Programming with Data-Flow Languages

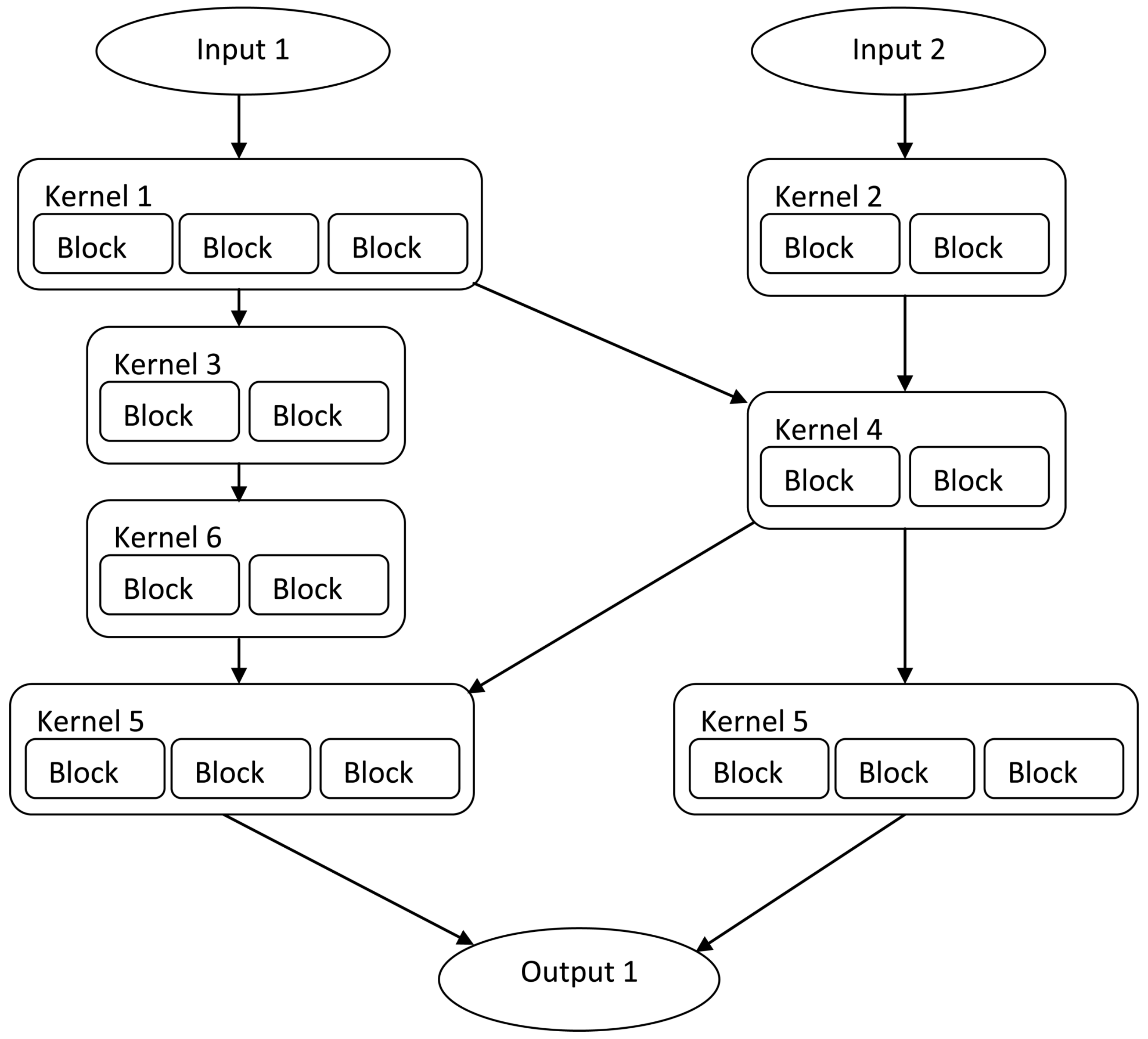
Daniel Goodman

Mikel Luján

This poster argues that the memory and synchronization restrictions when programming GPU's mean that GPU's are better served by an alternative programming style known as data-flow programming. We demonstrate that correctly constructed GPU programs map onto a coarse grained data-flow model, and that the programming of GPU's is a specific reoccurrence of the more general data-flow model.

GPUs have a high level of parallelism, on which large numbers of threads run in small groups to compute independent results without communication. These computations are controlled by the flow of data through pipelines as transferring control information between groups of threads is not possible. Currently CUDA and OpenCL are low level non-domain specific languages for programming GPUs. With these the programmer is loaded with a lot of low level details that add time and complexity to the construction of codes. This has resulted in a large number of domain specific languages being produced. These are augmented by more general purpose languages provided by companies such as the Portland Group and MathWorks. However all of these languages 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. The imperative model was originally developed for single threaded applications and lacks an intuitive way of handling the large levels of concurrency.

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


Input 1
Input 2
Kernel 1 Block Block Block
Kernel 2 Block Block
Kernel 3 Block Block
Kernel 4 Block Block
Kernel 6 Block Block
Kernel 5 Block Block Block
Kernel 5 Block Block Block
Output 1

Current GPU	Dataflow
Read Only: 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	**Read Only**: Memory which can be read by multiple threads in order to get their input.
Owner Writable: 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.	**Owner Writable:** 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 completed this memory can become read only memory.
Atomic: Memory that is protected by atomic sections, so allowing writes from multiple threads at a performance cost.	**Atomic**: Memory that is protected by atomic sections, allowing multiple threads to safely modify it, but potentially at a performance cost.
Block Local: Memory used to store temporary values used by a single block or thread within the block	**Thread Local:** Memory used to store temporary values used by a thread.

CUDA Program Execution

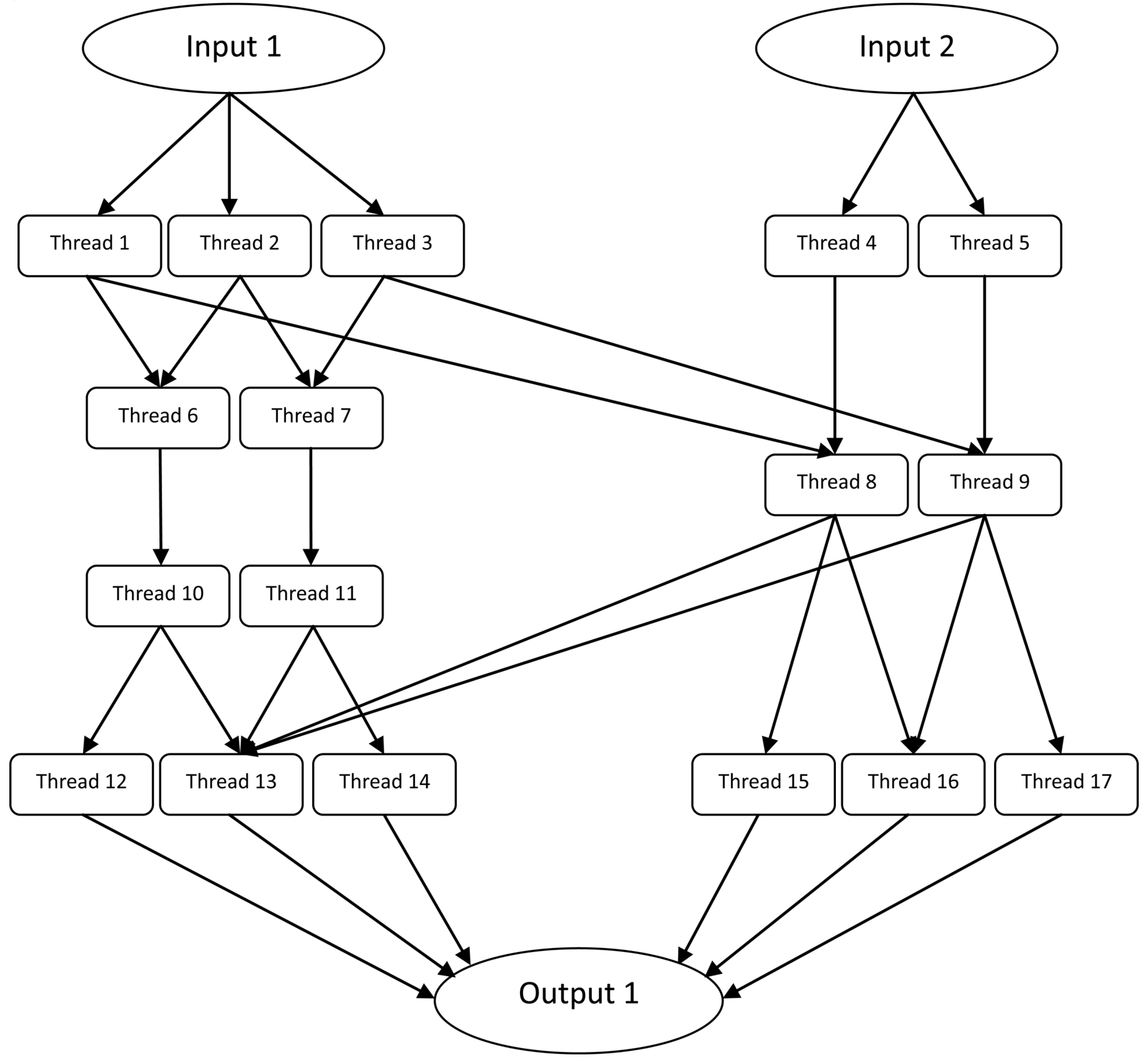
CUDA kernels are made up of Blocks. Blocks are independent of each other and have no guarantees about when they will execute, which order that they will execute in, or even that two blocks will be executing at the same time. To handle this concurrency, normally blocks will maintain separation on outputs and will only share inputs. Atomic statements can be used to ensure that interleaved of instructions do not affect the correctness of the result written to areas of memory used by multiple blocks. However, 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 and placed in separate kernels. These kernels consist of the computation before the communication and the computation after the communication. The information that the blocks wish to communicate is then 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 in addition to 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 in the above table. The mapping of these abstract types of memory map onto the physical memories can be seen in the table on the left. 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. A dependency graph constructed from multiple kernels can be seen on the left, showing how different kernels can run concurrently and depend on each other's results.

Data-Flow Languages

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. This means that once a piece of data has been written to it will remain the same for the entire execution of the program. However, the removal of mutable memory restricts the class of programs that can be described, and the way programs can be described. For example, loops have to demonstrate the same semantics as recursion, and systems such as a concurrent booking service for an airline is not possible. The restriction to loops is overcome by allowing mutable thread local memory. The restriction on a parallel booking system is because of the need for a single piece of data describing all bookings that can only be modified by a single thread at a time. Shared mutable state is needed to overcome this. Such shared state needs protection from race conditions. 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.

These restrictions leaves us with the strikingly similar set of four types of memory as can be seen in the table on the left. 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.


Input 1
Input 2
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Thread 16
Thread 17
Output 1

Conclusion

GPUs are data-flow devices, however, for scientific computing they are programmed with imperative languages 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 believe it would be better to develop a data-flow language for scientific computation on GPUs. Such a language would make it easier for the user to program codes to run on these devices, and would also allow stronger semantics that could support type checking and memory management in an efficient way. Collectively this would make GPUs easier to program, more accessible and will blur the line between the GPU and the CPU.

Acknowledgments

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