The Function-Centric Model: Supporting SIMD Execution of Streaming Computations

Stephen V. Cole and Jeremy D. Buhler
Washington University in St. Louis
St. Louis MO, 63110, USA
E-mail: {svcole, jbuhler}@wustl.edu

***Abstract*—Wide-SIMD multiprocessors such as GPUs are an increasingly prominent architecture for streaming computing. Existing prescriptive Parallel Execution Models (PXMs) do not inherently exploit the SIMD parallelism offered by such architectures. We therefore introduce a novel PXM, the *Function-Centric* (FC) model, tailored to achieve high-throughput execution of streaming applications on a wide-SIMD multiprocessor architecture. The FC model forms the basis of our MERCATOR framework supporting modular streaming CUDA applications, which is currently under development.**

1. Introduction / Background

Emerging architectures show a trend towards increasing SIMD execution width among both traditional multiprocessors and off-chip accelerators. The essence of SIMD execution is that it applies the same function to multiple data inputs in parallel. As for many parallel platforms, writing efficient application code for wide-SIMD machines is challenging. To provide guidance to application designers, parallel platforms are often abstracted through prescriptive *parallel execution models* (PXMs), which propose a structured execution strategy conducive to achieving high performance. The closer the match between a PXM and the underlying architecture on which it is realized, the greater its potential to guide developers to high-performing implementations.

In this work, we introduce the Function-Centric (FC) PXM for mapping streaming computation to wide-SIMD architectures. Streaming computation is a widely used paradigm for processing massive data streams that can be mapped onto many different parallel architectures. The FC model keeps the data streams in a computation partitioned *by the function to be applied to the items* throughout an application's execution. This partitioning exposes the maximum potential for the processor to fill SIMD lanes at each execution step. The FC model contrasts with existing streaming computing models, which either partition data more finely and therefore isolate potentially SIMD-executable inputs, or queue data for different computations together and therefore compromise SIMD execution.

1.1. Streaming computing paradigm

We use the term *streaming computing* to denote computational models with the following characteristics:

- The input consists of an unbounded stream of *data items*.
- Computations to be performed on data items are described by a *dataflow graph* (DFG) whose nodes each perform some function on their input streams (if any) and may produce output streams.
- Each data item input to a node may generate zero or more outputs in a dynamic, data-dependent fashion.
- The computational performance metric of interest is *total throughput*, defined as number of input items consumed per unit time.

1.2. GPUs as wide-SIMD multiprocessors

GPUs are today's preeminent wide-SIMD multiprocessors. Compared to other SIMD+MIMD architectures (e.g., Intel Xeon Phi, Tilera chips), GPUs support wider SIMD execution. In this work, we focus on NVIDIA GPUs programmed in CUDA as representative of a broader class of architectures for which our FC model is appropriate.

Processor/core structure A GPU contains several (8-15) independent *streaming multiprocessors* (SMs). Fixed-sized groups (*warps*) of w threads are bound to a particular SM for scheduling and execution. An SM can be thought of as a w-lane SIMD execution unit, with threads corresponding to individual SIMD lanes. All current NVIDIA GPUs set $w = 32$. A single SM may dynamically context-switch among several active warps.

Execution GPU code is exposed for execution as one or more *kernels* that are launched from CPU code. The division of work across SMs is specified at launch time in the form of a desired number of *blocks* of work and of compute threads (hence, warps) per block. The warps of a single block are executed by a single SM for the duration of the kernel.

Execution constraints To distill the salient features of GPU execution behavior, we will constrain an application's mapping to the GPU in the following ways:

- Each application runs within a single kernel call, keeping the GPU block configuration and the assignment of blocks to SMs fixed throughout execution.
- Every thread within a GPU block executes the same code.
- GPU blocks operate on disjoint sets of input items and use independent working memory sets, which include any data-dependent DFG routing information computed during the course of execution. Hence, no synchronization among blocks is required.

Under these constraints, each block may be viewed as an independent instance of the application running on its own input data set. The *conceptual* SIMD width is increased from the minimum *physical* width w to the number of threads in an entire block, which may in practice be in the hundreds.

1.3. Related work

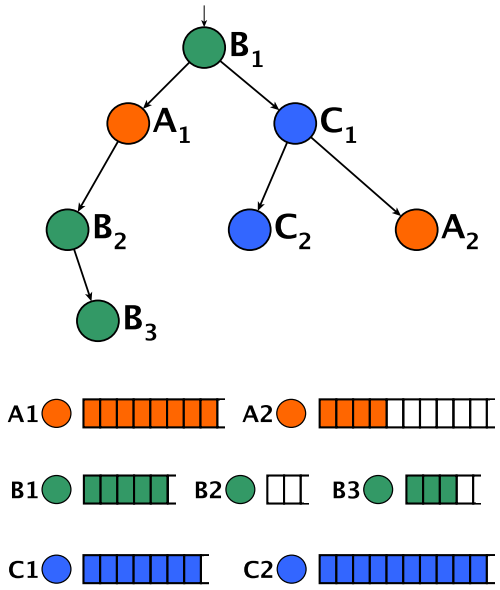
Certain classes of streaming application have been optimized for SIMD processing (e.g., tree-traversal algorithms [1] and polyhedral-friendly algorithms [2]). However, to the best of our knowledge, no general prescriptive PXM has been proposed targeting streaming SIMD computation. Although dozens of streaming applications have been ported to GPU architectures (see [3] for a sample), few implementations follow a general PXM, and the optimizations they use to achieve high throughput are therefore application-specific. We mention general frameworks that do conform to an existing PXM below.

Existing PXMs for streaming processing belong to one of two broad categories based on their work queueing schemes: Dataflow Process Network models and Shared Worklist models.

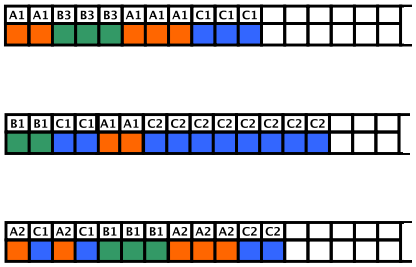
Dataflow Process Network (DPN) models Dataflow Process Networks [4] encompass both Kahn Process Networks [5] and Dataflow Models (e.g. [6]). In a DPN, as shown in Figure 1a, each edge is realized as a typed FIFO queue storing work to be performed by its downstream node, and nodes communicate only via their queues. Modular computing frameworks such as Auto-Pipe [7], RaftLib [8], and Ptolemy [9] are DPN-based, as are the GPU-based StreamIt frameworks presented in [10] and [11].

Shared Worklist (SW) models In contrast to the well-defined topology and typed queues of a DPN, the Shared Worklist model for general task-based execution, illustrated in Figure 1b, does not distinguish work by graph topology or data type. Instead, individual "tasks," each of which encapsulates a particular function to be applied to particular data, are placed on a common worklist or per-processor worklists. Strategies such as work stealing and work donation [12] are used to balance computation across resources. GPU frameworks such as those in [13] and [14] are based on the Shared Worklist model.

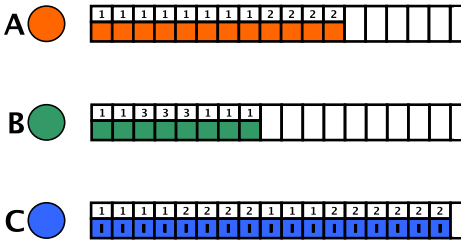
The Codelet PXM [15], [16] contains components that operate under each of the DPN and SW models. Its *codelets* are event-driven and atomically scheduled, as for nodes in



(a) **Data Process Network (DPN) models**. Each function instance (node) has its own worklist holding data to be processed by that node.



(b) **Shared Worklist (SW) models**. Globally shared worklists combine data to be processed by all nodes. Each data item on each worklist is tagged with its target node.



(c) **Function-Centric (FC) model**. Each distinct function (i.e., module type) has its own worklist holding only data to be processed by instances of that module type. Each data item on the worklist is tagged with its target module instance.

Figure 1: Dataflow graph for a streaming application composed of three functions (A, B, C), each instantiated at multiple nodes, and work queueing strategies under three streaming PXMs. The FC model resembles a modified DPN model in which worklists for nodes representing the same function have been merged.

the DPN model, while its *Threaded Procedures* (TPs) pull coarse-grained tasks from a common worklist and employ work-stealing techniques, as in the SW model.

Inadequacy of Existing PXMs While both DPN-based and Shared Worklist PXMs can be used to structure applications whose nodes internally implement SIMD computations, the models themselves do not optimally group data items that could be processed in parallel as a SIMD ensemble. A DPN system maintains separate, isolated queues even for nodes that execute identical code on their input streams. In contrast, an SW system mixes data requiring different forms of processing in a common worklist. We seek an ideal middle ground: a model that keeps separate those data items requiring different kinds of processing yet groups items requiring identical processing, thereby making it easier to form ensembles of items that can fill the lanes of a wide SIMD architecture.

2. Function-Centric (FC) model

To address the shortcomings of existing streaming PXMs on wide-SIMD architectures, we propose the *Function-Centric* or FC model. By organizing an application's pending work into a form that exposes maximum SIMD-friendly parallelism, the FC model automatically capitalizes on opportunities for wide-SIMD execution in a way that models targeting more general multiprocessor architectures do not.

Structure The FC model abstracts an application as a dataflow graph whose components are defined as follows.

(1) Each distinct function, as defined by its instruction stream, that operates on streaming data (possibly at one or more nodes of an application) is defined to be a *module type* (or just *module*).

(2) Each node in the application's DFG instantiates some function and is therefore a *module instance*. Multiple instances of the same module type may exist in a single DFG. Modules may be parametrized; for example, a filtering module may have many instances that implement the same filter but with different threshold values. In such cases, the functional code defining the module type is still constant across all instances of that type, while the instance to which each item is targeted is passed as a data parameter to the module type's code.

(3) Each module type has a single corresponding worklist holding all data items awaiting processing by an instance of that type. Because all instances execute the same stream of instructions on their inputs, each worklist is composed of exactly those items that require identical processing, hence can be processed in parallel in SIMD fashion!

Figure 1c illustrates the FC model's worklist strategy, while Table 1 highlights key differences between the FC, DPN, and SW models.

Execution In a GPU application organized after the FC model, a block runs the application on its input stream by executing a series of *module firing* steps. A single firing first chooses a module type to execute; all warps in the block then pull data items from that module type's worklist, forming an ensemble of items requiring identical processing. This ensemble is then processed in SIMD fashion by the module type's code. A firing may generate outputs that are placed on one or more other worklists for processing by later firings.

TABLE 1: Characteristic Features of Different PXMs

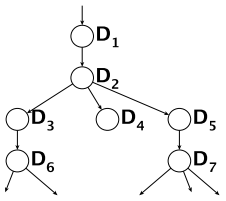
Feature	DPN	SW	FC
Worklist scope	module instance	global or processor	module type
Worklist composition	function-specific	mixed-function	function-specific
Communication	local	global	global
Typed worklist?	yes	no	yes

Execution of a full application is simply a sequence of firings, each of which processes one or more pending items from some module's worklist, until the application terminates. To preclude situations in which some firing sequences may lead to deadlock with bounded worklist storage, we restrict dynamic data-rate application in the FC model to those whose DFG topologies contain only single-input nodes (i.e. trees). General graph topologies can be supported safely if all modules' data production rates are fixed, as in Synchronous Data Flow (SDF) applications.

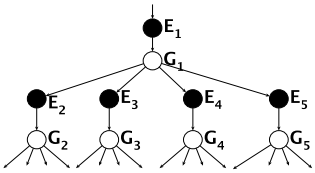
A performance-critical consequence of the FC model is that parallel SIMD execution of DFG nodes is implicitly contained in the module firings, since inputs to multiple DFG nodes of the same module type are processed simultaneously during a firing. In other words, the model automatically induces SIMD execution even without explicit coding by the application programmer. Indeed, a module's code could in principle be specified as a single-threaded function that is transparently replicated across all lanes of a SIMD processor. The dynamic scheduling capability of the FC model affords it more flexibility than a classical event-driven Data Flow execution model: an FC scheduler *may* always choose to fire modules as soon as their inputs are available, preserving the integrity of Synchronous Data Flow behavior; however, it may also choose to wait to fire an module until a certain number of inputs have been queued on its worklist in order to maximally fill SIMD lanes upon firing, increasing execution parallelism over that realized by an event-driven model. The problem of finding *throughput-optimal* firing schedules for given SIMD width will likely be a fruitful topic for research.

Example applications To demonstrate the utility of the FC model, we now show how two high-impact streaming applications may be implemented in conformity to the model. DFGs of these applications are shown in Figure 2.

In *random-forest evaluation* (Figure 2a), input items (feature vectors) are streamed through a filter cascade. At each level of the cascade, a critical value is computed from an input item's data and compared to a threshold; the result of this comparison determines the output path of the item. Each node of the filter cascade is a module instance parametrized by its threshold value, with nodes representing the same discriminator function having the same module type. When implemented in conformity to the FC model, all nodes of the same module type will share a worklist,



(a) **Random forest classifier**. Classifier-specific topology with module instances operating on items. Since all instances are of the same module type, the FC model merges data inputs to all nodes into a single worklist, enabling every module firing to execute with the maximum possible SIMD width.



(b) **WOODSTOCC**. Alternating layers of module instances of two different types implement dynamic-programming and tree-traversal calculations, respectively. In this case, the FC model creates two separate worklists, one for each type, and each module firing processes work from one of the two worklists in SIMD.

Figure 2: Dataflow graphs of example applications. Only the top few levels of the graphs are shown; in practice, they may extend to many more levels depending on the inputs to be processed.

and inputs to all of these nodes can be processed in SIMD fashion.

A second application is *DNA short-reads alignment* (Figure 2b), which performs approximate string matching of many short DNA strings against a common long reference string. The search is performed one character at a time, first computing a row of a dynamic programming matrix for each successive reference character, then computing the next character to align based on a virtual tree traversal. The two functions of dynamic programming calculation and tree traversal are the module types of the application.

Our WOODSTOCC application [17] implements short-reads alignments in conformity to the FC model, and it therefore contains one worklist corresponding to each function type. Empirical testing on NVIDIA GPUs [18] revealed a performance improvement of $> 50\%$ with the introduction of these worklists, which are themselves themselves managed using SIMD operations, compared to a sequential data management scheme.

3. Conclusion and Future Work

We have introduced the Function Centric model for streaming application execution on wide-SIMD multiprocessors. By dividing an application's data streams according to the function implemented at the node receiving each stream, the FC model naturally promotes grouping of items to be processed into wide SIMD ensembles, facilitating efficient execution of applications on GPUs and other SIMD architectures. The model can expose SIMD parallelism across data items even without explicit effort by the programmer to exploit parallelism within each node's computation.

We are currently implementing a realization of the FC model, based on a generalization of the worklists used in the WOODSTOCC application, in our MERCATOR framework.

Acknowledgment This work was supported by NSF award CNS-0905368 and Exegy, Inc.

References

[1] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *IEEE Int'l Symp. Code Generation and Optimization*, 2013, pp. 1–10.

[2] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *Proc. 34th ACM Conf. Programming Language Design and Implementation*, 2013, pp. 127–138.

[3] W.-m. W. Hwu, *GPU Computing Gems Emerald Edition*, 1st ed. San Francisco: Morgan Kaufmann, 2011.

[4] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. Int'l Fed. Information Processing Cong.*, vol. 74, 1974, pp. 471–475.

[6] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[7] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-pipe and the X language: A pipeline design tool and description language," in *Proc. Int'l Parallel & Distributed Processing Symp.*, 2006.

[8] J. C. Beard, P. Li, and R. D. Chamberlain, "Raftlib: A C++ template library for high performance stream parallel processing," in *Programming Models and Applications on Multicores and Manycores*, 2015.

[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[10] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *Proc. Int'l Symp. Code Generation and Optimization*, 2009, pp. 200–209.

[11] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. S. Goh, "Automated architecture-aware mapping of streaming applications onto GPUs," in *IEEE Int'l Parallel & Distributed Processing Symp.*, 2011, pp. 467–478.

[12] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[13] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU Task-Parallel Model with Dependency Resolution," *IEEE Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[14] M. E. Belviranli, C.-H. Chou, L. N. Bhuyan, and R. Gupta, "A Paradigm Shift in GP-GPU Computing: Task Based Execution of Applications with Dynamic Data Dependencies," in *Proc. 6th Int'l Wkshp. Data Intensive Distributed Computing*, 2014, pp. 29–34.

[15] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines: Position paper," in *Proc. 1st Int'l Wkshp. Adaptive Self-Tuning Computing Systems for the Exaflop Era*, 2011, pp. 64–69.

[16] J. Suettlerlein, S. Zuckerman, and G. Gao, "An implementation of the codelet model," in *Euro-Par 2013 Parallel Processing*, 2013, vol. 8097, pp. 633–644.

[17] S. V. Cole, J. R. Gardner, and J. D. Buhler, "WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU," in *IEEE 13th Int'l Symp. Parallel & Distributed Computing*, 2014.

[18] Cole, Stephen V. and Gardner, Jacob R. and Buhler, Jeremy D., "WOODSTOCC: Extracting Latent Parallelism from a DNA Sequence Aligner on a GPU," *Washington University Tech Report WUCSE-2015-004*, Sep 2015.