CnC-CUDA: Declarative Programming for GPUs

Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar

Department of Computer Science, Rice University
{jmg3,alina,zoran,vsarkar}@rice.edu

Abstract. The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. Instead, future computer systems are expected to be built using homogeneous and heterogeneous many-core processors with 10's to 100's of cores per chip, and complex hardware designs to address the challenges of concurrency, energy efficiency and resiliency. Unlike previous generations of hardware evolution, this shift towards many-core computing will have a profound impact on software. These software challenges are further compounded by the need to enable parallelism in workloads and application domains that traditionally did not have to worry about multiprocessor parallelism in the past. A recent trend in mainstream desktop systems is the use of graphics processor units (GPUs) to obtain order-of-magnitude performance improvements relative to general-purpose CPUs. Unfortunately, hybrid programming models that support multithreaded execution on CPUs in parallel with CUDA execution on GPUs prove to be too complex for use by mainstream programmers and domain experts, especially when targeting platforms with multiple CPU cores and multiple GPU devices.
In this paper, we extend past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. CnC is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using steps that are related by data and control dependence edges, which are represented by a CnC graph. The CnC-CUDA extensions in this paper include the definition of multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution.

1 Introduction

The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. Instead, future computer systems are expected to be built using homogeneous and heterogeneous many-core processors with 10's to 100's of cores per chip, and complex hardware designs to address the challenges of concurrency, energy efficiency and resiliency. Unlike previous generations of hardware evolution, this shift towards many-core computing will have a profound impact on software.

These software challenges are further compounded by the need to enable parallelism in mainstream workloads and application domains that have traditionally not had to worry about multiprocessor parallelism in the past. Despite over four decades of research, there are few choices of high-level parallel programming models available to domain experts who are not computer science experts. Fortunately, this situation is starting to change. Systems like MapReduce [7] are succeeding based on implicit parallelism, albeit with a restricted applicability. Other systems like CUDA [20] and OpenCL [17] are partially there for GPU accelerators, providing a restricted programming model to the user but also exposing a fair amount of hardware details.

Intel's Concurrent Collections[1] (CnC) is a declarative and implicitly parallel coordination language that supports flexible combinations of task and data parallelism while retaining determinism. CnC computations are built using *steps* that are related by data and control dependence edges, which in turn are represented by a CnC *graph*. CnC is provably deterministic [2]. While this restricts CnC's scope, it is more general than other deterministic programming models including dataflow and stream-processing, and can incorporate static and dynamic forms of task, data, loop, pipeline, and tree parallelism. However, all known implementations of CnC to date have been on homogeneous multicore SMP's.

A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. As an example, NVIDIA's Compute Unified Device Architecture (CUDA) has emerged as a popular hybrid programming model for CPUs and GPGPUs [20]. While it can be fairly straightforward for mainstream programmers to write the device code for specific kernels in CUDA, the CPU-GPU interactions necessary for deploying a complete CUDA application can be complicated to implement because of the necessary control flow for launching new kernels, data flow for communicating inputs and outputs, and synchronization to ensure proper coordination between the CPU and GPU. Further, debugging the execution of a CUDA program across a multicore SMP and a GPU is especially onerous because of the loose coupling via the device interface and the lack of integrated debugging tools. For these reasons, we believe that writing and deploying full CUDA applications is beyond the scope of mainstream domain experts, from the viewpoints of both programmability and productivity.

In this paper, we extend past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. The CnC-CUDA extensions in this paper include the definition of multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps. Further, given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, we have developed a Java-based implementation of CnC which provides the foundation for the CnC-CUDA implementation. In this way, the programmer has the choice of writing CPU Steps in Java or C (since C code can be invoked from Java) and GPU steps in CUDA, and can leave all the remaining details of creating and managing parallel tasks

[1] An earlier version of CnC was called TStreams [18].

and data transfers to the CnC-CUDA framework. The CnC-CUDA extensions in this paper include the definition of *multithreaded steps* for execution on GPUs, and *automatic generation of data and control flow* between CPU steps and GPU steps. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution. To the best of our knowledge, this is the first experience with mapping the CnC model on to hybrid systems with accelerators (GPUs).

The rest of the paper is organized as follows. Section 2 briefly summarizes the CnC and CUDA programming models. Section 3 introduces the CnC-CUDA Programming Interface and Implementation. Section 4 presents preliminary experimental results for CnC-CUDA. Related work is discussed in Section 5, and our conclusions are contained in Section 6.

2 Background

2.1 Concurrent Collections Programming (CnC) Model

In this section, we give a brief summary of the CnC model, as described in [3]. As in dataflow and stream-processing languages, a CnC program is a graph of serial kernels, communicating with one another. The three main constructs in CnC are *step collections*, *data item collections*, and *control tag collections*. Statically, each of these constructs is a *collection* representing a set of dynamic *instances*. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization and communication. Control tag instances are the unit of control.

The program is represented as a graph. In textual form, the graph is denoted using `()` to suggest circles for computation steps, `[]` to suggest boxes for data items and `<>` to suggest triangles for control tags. The edges in the graph specify the partial ordering constraints required by the semantics. One type of ordering constraint arises from a *data dependence*. This relationship occurs when an instance of a step, say `(F1)`, produces an instance of an item, say `[X]`, which is later consumed by an instance of another step, say `(F2)`. Clearly the producing step instance must occur before the consuming step instance. Another type of ordering constraint arises from a *control dependence*, where one computation step determines if another computation step will execute. In that case, the controller step puts a control tag in a tag collection, which in turn *prescribes* the controllee step. The execution order of step instances is constrained only by their dynamic data and control dependences.

Control, data, and step instances are all identified by a unique *tag* within each collection. In CnC, tags are arbitrary values that support an equality test and hash function. Each type of collection uses tags as follows:

- Putting a tag into a control collection will cause the corresponding steps (in the prescribed step collections) to eventually execute. A control collection C with tag i is denoted $< C : i >$.
- Each step instance is a computation that takes a single tag (originating from the prescribing control collection) as an argument. The step instance of collection (foo) at tag i is denoted $(foo : i)$.

– A data collection is an associative container indexed by tags. The entry for a tag i, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection is necessary for determinism. An instance in data collection x with tag "i, j" is denoted $[x : i, j]$.

A CnC specification can optionally include *tag functions* [13] and use them to specify the mapping between a step instance and the data instances that it *consumes* or *produces*. A tag function can be the identity function, or can define nearest neighbor computations, a parent/child in a tree, neighbors in a graph, or any other relationship useful in the application.

2.2 Habanero-Java implementation of CnC

The Habanero-Java (HJ) is a programming language being developed in the Habanero Multicore Software Research project at Rice University [1]. We chose it for the baseline implementation of the CnC runtime system because it includes constructs that serve as a convenient target for implementing CnC primitives. We were pleasantly surprised to see how straightforward it has been to map CnC primitives to HJ, as summarized in Table 1.

CnC construct	Translation to HJ
Tag	Java *String* object or X10 *point* object
Prescription	*async* or *delayed async*
Item Collection	`java.util.concurrent.ConcurrentHashMap`
put() on Item Collection	Nonblocking *put()* on `ConcurrentHashMap`
get() on Item Collection	Blocking or nonblocking *get()* on `ConcurrentHashMap`

Table 1: Summary of mapping from CnC primitives to HJ primitives

Following are additional details for this mapping.

Tags We used the X10 *point* construct to implement Tags. A *point* in X10 is an integer tuple, that can be declared with an unspecified rank. A multidimensional tag is implemented by a multidimensional point.

Prescriptions We have optimized away all prescription tags in the HJ implementation. When a step needs to put a prescription tag in the tag collection, we perform a normal *async* or a *delayed async* for each step prescribed by that tag. The normal async statement, *async* ⟨*stmt*⟩, is derived from X10 and causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement. The delayed async statement, *async (*⟨*cond*⟩*)* ⟨*stmt*⟩, is similar to a normal async except that execution of ⟨*stmt*⟩ is guaranteed to be delayed until after the boolean condition, ⟨*cond*⟩, evaluates to true.

Item Collections We use the `java.util.concurrent.ConcurrentHashMap` class to implement item collections. Our HJ implementation of item collections supports the following operations:

- `new ItemCollection(String` *name*`)`: create and return a new item collection. The string parameter, *name*, is used only for diagnostic purposes.
- C.`put(point` p`, Object` O`)`: insert item O with tag p into collection C. Throw an exception if C already contains an item with tag p.
- C.`awaitAndGet(point` p`)`: return item in collection C with tag p. If necessary, the caller blocks until item becomes available.
- C.`containsTag(point` p`)`: return true if collection C contains an item with tag p, false otherwise.
- C.`get(point` p`)`: return item in collection C with tag p if present; return `null` otherwise. The HJ implementation of CnC ensures that this operation is only performed when tag p is present *i.e.,* when C.`containsTag(point` p`)` = `true`. Unlike `awaitAndGet()`, a `get()` operation is guaranteed to always be nonblocking.

Put and Get Operations A CnC `put` operation is directly translated to a `put` operation on an HJ item collection, but implementing `get` operations can be more complicated. A naive approach is to translate a CnC `get` operation to an `awaitAndGet` operation on an HJ item collection. However, this approach does not scale well when there are a large number of steps blocked on `get` operations, since each blocked activity in the current X10 runtime system gets bound to a separate Java thread. A Java thread has a larger memory footprint than a newly created async operation. Typically, a single heavyweight Java thread executes multiple lightweight async's; however, when an async blocks on an `awaitAndGet` operation it also blocks the Java thread, thereby causing additional Java threads to be allocated in the thread pool[9]. In some scenarios, this can result in thousands of Java threads getting created and then immediately blocking on `awaitAndGet` operations.

This observation lead to some interesting compiler optimization opportunities of `get` operations using delayed asyncs. Consider a CnC step S that performs two `get` operations followed by a `put` operation as follows (where T_x, T_y, T_z are distinct tags):

$$S\text{: } \{\ x := C.\texttt{get}(T_x);\ y := C.\texttt{get}(T_y);\ z := F(x,y);\ C.\texttt{put}(T_z, z);\ \}$$

Instead of implementing a prescription of step S with tag T_S as a normal async like "`async` $S(T_S)$", a compiler can implement it using a delayed async of the form "`async await(`C`.containsTag(`T_x`) && `C`.containsTag(`T_y`))` $S(T_S)$". With this boolean condition, we are guaranteed that execution of the step will not begin until items with tags T_x and T_y are available in collection C.

2.3 GPU Architecture and the CUDA Programming Model

The NVIDIA CUDA programming model is an interface designed to allow programmers access to the extremely parallel hardware of programmable Graphics

Processing Units (GPUs). With an architecture originally intended for graphics rendering, the GPU's strengths lie with highly parallel applications, streaming data, and low inter-thread communication and synchronization. For instance, the GPU used in our performance evaluation, an NVIDIA GTX 480, has 480 processing cores. From this we can see that GPUs' architecture make them easily applicable to scientific and mathematical computing problems.

CUDA is an extension on the C/C++ programming language, with the CUDA runtime library providing a collection of device memory management, host-device stream synchronization, and execution control functions (among others). The general flow of a CUDA program consists of the following steps [23], where all allocation and copying of device memory is controlled explicitly or implicitly by the host using the CUDA runtime library:

1. Copy data from main memory to GPU memory
2. CPU instructs GPU to start a kernel
3. GPU executes kernel in parallel and accesses GPU memory
4. Copy the results from GPU memory to main memory

CUDA is a data parallel SIMD architecture, with the same programmer defined kernel running for all threads launched. These threads are launched in batches of blocks and grids, where blocks are collections of threads and grids are collections of blocks.

3 Programming Interface and Implementation

We have implemented several additions in the CnC programming model in order to support CUDA steps, which we outline in this section.

3.1 Graph File

Some of the features added require using new syntax in the graph file. First, we introduce a new syntax for CUDA steps. CUDA steps are declared with braces - {} - instead of parenthesis - () - for CPU steps. The graph file can now define both CPU and GPU steps.

Second, we have added support for the programmer to specify constants in the graph file using the following notation:

$|const\_name\ \ const\_value|;$

where *const_value* is of an integer type. This definition generates constant values to be used both in HJ and CUDA. These constants are used for specifying the exact type of item and tag collections that are passed to CUDA, ensuring the copying of the correct amount of data from Java arrays onto the CUDA device. For example, if each CUDA thread takes 1000 integers, this item collection would be declared as:

$[int\ \ items[1000]];$ `, or` $|size\ \ 1000|;\ \ [int\ \ items[size]];$

Using the second method allows the CnC programmer access to the constant value inside the computation step and the program's entry point, ensuring no stale values for those constants caused by multiple definitions.

3.2 Item Collections

All item collections are dynamically generated using the CnC Parser in order to enforce strict typing rules on items and tags, since these are passed through JNI to C code.

Item collections maintain the standard interface for adding or retrieving items, the Put and Get methods for individual items. With this approach, each of the items is put into a ConcurrentHashMap, similarly to what is done in CnC-HJ. Once a threshold number of tags have been put, the items corresponding to those tags are collected from the ConcurrentHashMap, converted to a C friendly format (i.e., java.lang.Integer→int) and passed to CUDA. While this approach correctly handles individual Put and Get operations, it also results in a significant performance overhead.

In CUDA-CnC we introduce a much more efficient alternative. The PutRegion/GetRegion primitives allow the programmer to put a (potentially multidimensional) region of integers associated with a similarly dimensioned array of items. This approach eliminates putting and then extracting individual items since the array is directly passed to the kernel. Currently we only support arrays of primitive types (int[], float[], e.t.c.), which we believe to be a reasonable limitation as the CUDA kernels are usually coded using primitive types anyway. Optimizing the individual item puts and gets, adding support for getting individual items from region puts, and implementing PutRegion and GetRegion for arrays of more complex structures (i.e., user defined classes) are beyond the scope of this paper and a subject of future research.

Currently, if an error occurs in copying an item collection (i.e., insufficient device memory) no error is reported to the user. This is simple to implement as the library written for CnC-CUDA host-device memory transfer returns error values, and will certainly be in future work in CnC-CUDA.

3.3 Tag Collection - PutRegion and GetRegion

Tag collections are automatically generated using type definitions in the graph file. Our preliminary implementation only supports integer tags, which can be easily extended to any type that can be hashed, as in traditional CnC.

As described earlier, tag collections control the execution and synchronization of computation steps. Synchronization between computation steps using tag collections is a different matter in CUDA. First, a pthread mutex is used to indicate that the device is currently in use by a computation step and inaccessible by any new computation steps for non-Fermi architecture GPUs which do not support concurrent kernel execution. Second, we limit the number of CUDA computation steps that can be prescribed by another CUDA computation step to 1, which considerably simplifies the complexity of synchronizing multiple CUDA computation steps. If a CUDA step prescribes another CUDA step (as determined by analyzing the CnC graph file), the second step is invoked immediately following the first without returning to HJ. No limitation is placed on a CUDA step prescribing multiple HJ steps. Last, synchronization between host and device computation steps is used by a call to a CUDA tag collection's Wait() method.

This call blocks until all launched CUDA kernels have returned and their output has been placed in host memory.

Tag Collections also implement the PutRegion operation, which places a region of integer tags into the tag collection. PutRegion immediately launches a CUDA kernel for all tags in the range once the required items are available. On individual tag Puts, the tag collection waits for a threshold number of tags to be put, and then launches a CUDA kernel with those tags and their associated items. We have currently empirically set this threshold to 8192 tags. Once all tags have been put into a GPU tag collection, the programmer has to issue a call to that tag collection's *Wait()* function to be certain that all CUDA threads have completed as the transferring of data to and from device memory and launching of CUDA kernels is handled by a separate CPU thread.

For more advanced CUDA programmers we introduce the option of defining a two dimensional tag:

```
<int tag:two_region>
```

This offers the opportunity of placing a tag with 2 regions on the graph. Those two regions will be interpreted as number of blocks per grid and threads per block to be used in a kernel launch. In addition, one can specify the number of desired threads in a block by compiling the graph file with the flag: *-t <number of threads>*.

We also support the item collection property One-For-All (OFA), which passes the same data to each thread on a device. This property follows the format:

```
[int item:ofa]
```

where int can be any supported data type. This can result in both considerable saving in device and host memory (less memory allocated to copy from and to) as well as better performance with less time spent copying data to the device.

3.4 CUDA Kernel

The advantage of using CnC CUDA is that the user need not worry about the allocating and copying of data, but just about writing the actual CUDA kernel. The translator is the one responsible for generating stub codes that will allocate memory and copy the data structures to the device before a step is executed as well as free the device memory after these finish. During the execution of a kernel, no puts or get are done on GPUs. The actual step code is written as a CUDA __device__ function by the CnC programmer in a file named "XXXXXKernel.cu" where 'XXXXX' is the CUDA step name as declared in the graph file. The step function needs to be defined as a __device__ function because the __global__ entry point to the device is auto-generated by the translator to protect against unwanted threads entering the programmer-defined kernel (i.e. 513 tags are put, but 2x512 threads are launched. The auto-generated __global__ function will ensure the upper 511 threads of the second block do not enter the programmer-defined kernel). Also, CUDA kernels are limited to putting a single item on each output item collection. This limitation is a result of CUDA requiring preallocation of all device memory before kernel launch. Future work will allow the programmer to specify the number of items output from a kernel.

3.5 Implementation Details

The CnC-CUDA execution model requires a Java-to-native code interface. The approach outlined below builds on our past experience with the JCUDA system [23].

Java provides the *native* keyword which is used to define functions that are implemented in native code. Our implementation creates a libJVMToC.so library which contains CUDA and C code needed to communicate the data to and from the device and execute the kernel. This library encapsulates the C and CUDA code auto-generated by the translator and is generated by the compiler. In CnC-CUDA, all of the complexity of inter-language function-calls and device memory management is hidden from the CnC programmer and auto-generated at compile time, allowing them to focus on developing and implementing the algorithms in their application.

Habanero Java also offers the *extern* keyword — similar to *native* in Java — which greatly simplifies programming with native code. Compiling an HJ class with extern functions generates C stubs which will be included in the file with the native function implementation, named *ClassName_FunctionName*.

The C and CUDA code that are generated by our CnC translator are responsible for creating a data collection for every data structure declared in the graph file, copying the said data structure to the GPU before launching the kernel and from the device back to the CPU afterwards, and actual launching of the CUDA computation step(s). The CnC graph is analyzed to determine which item and tag collections are only accessed from the device (i.e., Put from one CUDA step and Get from another), and this analysis is used to remove extraneous device memory copies from the generated code.

4 Preliminary Experimental Results

4.1 Experimental Setup

In order to compare the performance of CnC-CUDA to CnC-HJ (available at [6]) and other programming models and languages we used three benchmarks from the Java Grande Forum (JGF) benchmark suite [15], as well as the Heart Wall Tracking program from the Rodinia benchmark suite [5]. The three benchmarks from the JGF suite are Fourier coefficient analysis (Series), successive over-relaxation (SOR), and IDEA encryption (Crypt). Each was run on varying data sizes using CnC-CUDA, CnC-HJ, Serial C, hand-coded CUDA, and the original single-threaded JGF Java benchmark. Additionally, the Crypt benchmark was run using CnC-CUDA and CnC-HJ computation steps running in parallel. The timing of each benchmark was started just before the first set of tag puts were performed to launch the CnC graph or the function call to launch the actual computation for non-CnC benchmarks. Timing was stopped when all CnC steps completed (as detected by the HJ finish construct) or the core function completed. For GPU execution, this included the overhead of copying data to and from device memory.

For these evaluations, we used an NVIDIA GTX 480 GPU. The GTX 480 has 480 processing cores and 1.6 GB of memory. The CPU host of this GPU

is a AMD Phenom 9850 Quad-Core Processor with a 1.25 GHz clock, 512 KB cache, and 8 GB of memory. The installed software includes a Java HotSpot 64-bit virtual machine from version 1.6.0_20 of the Java Development Kit (JDK), a GNU C compiler v. 4.1.2, and version 3.1 of the NVIDIA CUDA Toolkit.

There are a few limitations in the current CnC-CUDA implementation which will need to be addressed in future work. First, the current implementation limits tags to only be integers. Second, both parent and CUDA steps are assumed to always have the same block/grid structure; giving the CnC programmer the ability to change the number of threads used across parent-child CUDA computation steps could further increase the flexibility of our CnC-CUDA implementation.

4.2 Evaluation and Analysis

Tables 2, 3, and 4 display the average execution times across ten runs of the respective benchmarks in different programming models or languages. In the CnC versions (HJ or CUDA), the CnC Parser was used to auto-generate all the glue code, leaving the programmer to only provide the CnC step code in CUDA or HJ and the code for launching the CnC graph in the main HJ program. The CnC-CUDA measurements on the GPU were compared to CnC-HJ runs of the same benchmarks on the CPU and the results listed in the Speedup column of each table. Each CUDA kernel launch was performed with a constant 256 threads per block, with the grid size determined by the iteration size[2]. Each CnC-HJ execution was performed using 16 worker threads. (Over-provisioning the number of worker threads per CPU degraded performance for the benchmarks and hardware studied in this paper.)

No special CUDA memory (e.g., texture, shared, constant) was used in the execution of these benchmarks.

These results demonstrate that performance potential of GPUs can be made accessible to non-expert programmers through CnC-CUDA. Without any knowledge of CUDA's memory or threading model, a CnC-CUDA programmer can go from working with CnC-HJ to exploiting the computational power of a GPU using CnC-CUDA quickly and easily, achieving a magnitude of performance better than a quad-core CPU. For example, building the SOR benchmark from a CnC-HJ version required 3 hours of time to port the step code to CUDA, and resulted in a 35× speedup. Auto-generation of CUDA code using techniques such as those reported in [19] could result in a further productivity boost for non-expert programmers.

We observe that the speedup of CUDA over HJ increases as the size of the data set increases, with the maximum average speedup (406.00×) observed for the embarrassingly parallel Series benchmark at its largest data size. The minimum average speedup of 1.77× was observed for SOR executed with its smallest data size. While not observed in these results, it is of course possible for a GPU version of an application to run slower than a CPU version, when the relative overheads of host-device data transfers, CUDA initialization, or of control flow divergence lead to performance degradation on the GPU.

[2] An evaluation of alternate grid/block sizes is a subject for future work.

Crypt	GPU Performance		CPU Performance			
Data Size (bytes)	CnC-CUDA	CUDA (16 cores)	CnC-HJ	Serial C	Serial Java	Speedup
50,000,000 (JGF Size C)	0.886	0.161	2.067	7.367	2.92	2.33
75,000,000	1.208	0.253	3.239	11.033	4.387	2.68
100,000,000	1.488	0.341	4.460	14.678	5.818	3.00
150,000,000	2.311	0.550	6.903	22.039	8.716	2.99

Table 2: Execution times in seconds of JGF Crypt benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ.

Series	GPU Performance		CPU Performance			
Data Size	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Serial C	Serial Java	Speedup
10,000 (JGF Size A)	0.332	0.0095	3.587	3.157	6.777	10.80
100,000 (JGF Size B)	0.441	0.116	36.588	31.832	69.074	60.78
1,000,000 (JGF Size C)	1.411	1.279	572.86	321.553	N/A	406.00

Table 3: Execution times in seconds of JGF Series benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ.

SOR	GPU Performance		CPU Performance			
Data Size (Dim)	CnC-CUDA	CUDA	CnC-HJ (16 cores)	Serial C	Serial Java	Speedup
1,000 (JGF Size A)	0.403	0.021	0.714	1.691	1.247	1.77
1,500 (JGF Size B)	0.448	0.045	3.015	3.811	3.872	6.73
2,000 (JGF Size C)	0.498	0.078	5.400	6.769	6.891	10.84
3.000	0.602	0.186	12.079	15.309	15.677	20.06
4,000	0.795	0.475	21.512	27.262	27.658	27.06
5,000	0.952	0.813	33.547	42.600	43.129	35.24

Table 4: Execution times in seconds of JGF SOR benchmark implemented in several programming models, and Speedup of CnC-CUDA relative to CnC-HJ.

Crypt (150,000,000 bytes)	Hybrid Performance			
Percent of Data on GPU	Average	Slowest	Fastest	Speedup (Relative to CnC-CUDA)
10	3.042	3.806	2.493	0.76
20	3.066	3.765	2.727	0.75
30	2.720	3.048	2.223	0.85
40	2.289	2.750	1.878	1.01
50	2.139	2.397	1.973	1.08
60	2.035	2.242	1.538	1.14
70	2.076	2.799	1.755	1.11
80	2.189	2.511	1.883	1.06
90	2.143	2.344	1.968	1.08

Table 5: Execution times in seconds, and Speedup of a hybrid CnC-CUDA/HJ version of Crypt against only CnC-CUDA.

Heart Wall Tracking	GPU		CPU		
Data Size (# of frames)	CnC-CUDA	CUDA	Serial	Open MP 16 cores	CnC-HJ 16 cores
1	0.427	0.985	0.005	0.005	0.246
104	4.4842	3.6133	156.977	13.863	11.058

Table 6: Execution times in seconds on GPU - CnC-CUDA and hand-coded (Rodinia) CUDA versions - and CPU - Serial, OpenMP on 16 cores and CnC-HJ on 16 cores - of the Heart Wall Tracking benchmark.

The results in Table 5 shows the potential for performance improvement using hybrid CPU-GPU execution in the CnC model. For consistency, the hybrid CUDA/HJ tests were performed using 256 threads per block for GPU execution. These results were obtained by evaluating different load distributions between the CPU and GPU for the Crypt benchmark with its largest size, for which the average speedup of the GPU over the CPU in Table 2 was 1.82×. In Table 5, we see that an additional 1.14× speedup can be obtained over the pure CnC-CUDA version by a hybrid execution in which 60% of the load is placed on the GPU and 40% of the load on the CPU. An interesting topic for future research is to extend the CnC runtime to perform this load distribution adaptively and automatically, allowing for a single CnC-CUDA graph to dynamically and efficiently handle a wide range of data set sizes.

Finally, Table 6 shows the execution times in seconds for the CnC-CUDA and hand-coded CUDA versions of the Heart Wall Tracking benchmark (the hand-coded version was obtained from the Rodinia benchmark set [5]). When comparing the fastest times, we see that both versions have comparable performance when processing a single frame, but the CnC-CUDA version is 1.25× slower than the hand-coded version for 104 frames thereby reflecting the extra coordination overhead in CnC involved in sequencing the computation across frames.

5 Related Work

We discuss related work according to their attributes in three dimensions: *Declarative*, *Deterministic* and *Efficient*. A number of lower-level programming models in use today — *e.g.,* Intel TBB [22], .Net Task Parallel Library, OpenMP [4], Nvidia CUDA, Java Concurrency [21] — are non-declarative, non-deterministic, and efficient[3]. Deterministic Parallel Java [10] is an interesting variant of Java; though imperative (non-declarative), it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions.

Higher-level languages such as High Performance Fortran (HPF) [16], X10 [8], and Linda [14] contain hybrid combinations of imperative and declarative programming in different ways. HPF combines a declarative language for data distribution and data parallelism with imperative (procedural) statements, X10 contains a functional subset that supports declarative parallelism, and Linda is a coordination language in which a thread's interactions with the tuple space is declarative. Linda was a major influence on the CnC design, but CnC also differs from Linda in many ways. For example, an in() operation in Linda atomically removes the tuple from the tuple space, but a CnC get() operation does not remove the item from the collection. This is a key reason why Linda programs can be non-deterministic in general, and why CnC programs are provably deterministic. Further, there is no separation between tags and values in a Linda tuple; instead, the choice of tag is implicit in the use of wildcards. In CnC, there is a separation between tags and values, and control tags are first class constructs like data items.

[3] We call a programming model efficient if there are known implementations that deliver competitive performance for a reasonably broad set of programs.

Both streaming and dataflow languages have also had major influence on the CnC design. The CnC semantic model is based on dataflow in that steps are functional and execution can proceed whenever data is ready, without unnecessary serialization. However, CnC differs from dataflow in some key ways. The use of control tags elevates control to a first-class construct in CnC. In addition, item collections allow more general indexing (as in a tuple space) compared to dataflow arrays (I-structures). CnC is like streaming in that the internals of a step are not visible from the graph that describes their connectivity, thereby establishing an isolation among steps. A producer step in a streaming model need not know its consumers; it just needs to know which buffers (collections) to perform read and write operations on. However, CnC differs from streaming in that *put* and *get* operations need not be performed in FIFO order, and (as mentioned above) control is a first-class construct in CnC. We observe that CnC's dynamic put/get operations on data and control collections is a general model that can be used to express many kinds of applications (such as Cholesky factorization) that would not be considered to be dataflow or streaming applications.

With respect to our experimental results, we are not claiming that the GPU by itself offers a certain speedup [12], rather that the speedup we get is from taking advantage of the high amount of data parallelism in our test applications, having 256-512 GPU threads run in parallel instead of 16-32 on a CPU, and an easy to use programming model that hides the use of resources from the user while offering lower execution times. The overhead of copying data from and to the device is hidden by a much larger number of tasks run in parallel. The innovation we offer is an easy way for a programmer to specify the algorithm while taking advantage of the available resources. Like Oregami [11], CnC is based on a graph description of the algorithm, however in Oregami the programmer needs to design the program as a "set of parallel processes that communicate through explicit message passing. The identity of all of the processes are known at compile time [...]". Such restrictions are not applicable to CnC-CUDA.

In summary, CnC has benefited from influences in past work, but we are not aware of any other parallel programming model that shares CnC's fundamental properties as a coordination language, a declarative language, a deterministic language, and a language amenable to efficient implementation. To the best of our knowledge, this is the first experience with mapping the CnC model on to hybrid systems with accelerators (GPUs).

6 Conclusions and Future Work

In this paper, we extended past work on Intel's Concurrent Collections (CnC) programming model to address the hybrid programming challenge using a model called CnC-CUDA. The CnC-CUDA extensions in this paper include the definition of multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps. Further, given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, we have developed a Java-based implementation of CnC which provides the foundation for the CnC-CUDA implementation. In this way, the programmer has the choice of writing CPU Steps in

Java or C (since C code can be invoked form Java) and GPU steps in CUDA, and can leave all the remaining details of creating and managing parallel tasks and data transfers to the CnC-CUDA framework. The CnC-CUDA extensions in this paper include the definition of *multithreaded steps* for execution on GPUs, and *automatic generation of data and control flow* between CPU steps and GPU steps. Experimental results show that this approach can yield significant performance benefits with both GPU execution and hybrid CPU/GPU execution.

There are multiple opportunities for future research. We would like to support richer (non-primitive) element data types in the PutRegion and GetRegion primitives. In addition, we would like to extend the types accepted for tags to more than integers. There is a large amount of overhead incurred by transfers to and from device memory, and further experimentation with transfer patterns may yield better performance. Finally, our longer-term plan is to extend the CnC-CUDA implementation to serve as a unified runtime for heterogeneous combinations of CPUs, GPUs, and FPGAs in the CDSC project.

Acknowledgments

We would like to thank all members of the Habanero team at Rice University and the CnC team at Intel for valuable discussions and feedback related to the Concurrent Collections programming model. We gratefully acknowledge support from an Intel award during 2009-2010. This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127. Finally, we would like to thank Tim Warburton for providing access to the GPU systems used to obtain the performance results reported in this paper.

References

1. Habanero Multicore Software Project. http://habanero.rice.edu.
2. Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. The CnC Programming Model. *SIAM PP10, Special Issue on Scientific Programming*, 2010.
3. Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The Concurrent Collections Programming Model. David Padua (Ed.), Encyclopedia of Parallel Computing, Springer New York, to be published 2011.
4. Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Programming in OpenMP*. Academic Press, 2001.
5. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, October 2009.
6. Concurrent Collections in Habanero-Java (HJ). http://habanero.rice.edu/cnc-download, 2010.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
8. P.Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 519–538, 2005.

9. R.Barik et al. Experiences with an smp implementation for x10 based on the java concurrency utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006*, September 2006.
10. Robert L. Bocchino et al. A type and effect system for Deterministic Parallel Java. In *Proceedings of OOPSLA'09, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 97–116, 2009.
11. V. M. Lo et al. Oregami: Tools for mapping parallel computations to parallel architectures. *IJPP: International Journal of Parallel Programming*, 20(3):237–270, June 1991.
12. Victor W Lee et al. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *ISCA 2010: ACM IEEE International Symposium on Computer Architecture*, June 2010.
13. Z.Budimlić et al. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: the workshop on Declarative Aspects of Multicore Programming*, pages 47–58. ACM, 2008.
14. David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
15. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande.
16. Ken Kennedy, Charles Koelbel, and Hans P. Zima. The rise and fall of High Performance Fortran. In *Proceedings of HOPL'07, Third ACM SIGPLAN History of Programming Languages Conference*, pages 1–22, 2007.
17. Khronos OpenCL Working Group. The OpenCL Specification - Version 1.0. Technical report, The Khronos Group, 2009.
18. Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.
19. Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09*, pages 101–110, New York, NY, USA, 2009. ACM.
20. John Nickolls, Ian Buck, Michael Garland, Nvidia, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.
21. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
22. James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.
23. Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.