Dataflow Programming for Stream Processing

Marcos P. Rocha, Felipe M.G. França
Systems Engineering and
Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
Email: {mrocha,felipe}@cos.ufrj.br

Alexandre S. Nery
Department of Informatics and
Computer Science
State University of Rio de Janeiro
Rio de Janeiro, Brazil
Email: anery@ime.uerj.br

Leandro S. Guedes
Department of Informatics
Federal Institute of Education, Science and
Technology of Mato Grosso do Sul - IFMS
Corumbá, Brazil
Email: leandro.guedes@ifms.edu.br

***Abstract*—Stream processing applications have high-demanding performance requirements that are hard to tackle using traditional parallel models on modern many-core architectures, such as GPUs. On the other hand, recent dataflow computing models can naturally exploit parallelism for a wide class of applications. This work presents an extension to an existing dataflow library for Java. The library extension implements high-level constructs with multiple command queues to enable the superposition of memory operations and kernel executions on GPUs. Experimental results show that significant speedup can be achieved for a subset of well-known stream processing applications: Volume Ray-Casting, Path-Tracing and Sobel Filter.**

***Keywords*—Dataflow, Heterogeneous Systems, High-Performance Computing.**

I. Introduction

Moore's Law predicted that the number of transistors that can be fit into an integrated chip would nearly double almost every two years [1]. While such greater number of transistors over the years enabled significant advances in the processor's microarchitecture, the transistor miniaturization trend led to insulating and heat dissipation problems, prohibiting further increases on the circuit's clock frequency, responsible for most of the performance improvements back then. Thus, chip designers have since then focused on selling many-core architectures to keep their business model alive. Parallel programming quickly became a game-changing method to achieve high performance on such modern parallel architectures, such as General Purpose Graphics Processing Units (GPGPUs) and Chip Multi-Processors (CMPs) [2]. However, building parallel programs is not a trivial task. In fact, it often requires expertise and many hours of hard work to fully optimize an application to run on a particular parallel architecture.

The Dataflow [3] paradigm is a trending computation model that can naturally exploit the existing parallelism in applications. A dataflow program is described as a graph, where vertexes represent tasks (or instructions) and edges depict data dependency between tasks. Nodes will be fired to run as soon as all their input operands become available, instead of following program order. This means that independent node can potentially run in parallel if there are available processing elements. Recent research proposes the dataflow in different levels of abstraction and granularities [4], [5], [6], [7], [8], [9], [10], [11], [12], as an efficient and straightforward parallel programming model. Computational units (from instructions to functions, or even entire programs) are connected in a dependency graph allowing programmers to harvest the potential of modern parallel systems

This work presents JSucuri, a dataflow programming library written in Java which extends the original Sucuri [13] for Python. JSucuri aims to enable dataflow high-performance computing on heterogeneous systems using CPU and GPU. It has been implemented in Java as (i) an alternative to writing dataflow programs other than in Python and (ii) due to Java's thread-oriented parallel programming model, making it easier to share objects within JSucuri code. Moreover, the management of threads is often cheaper in terms of processing speed and usage of resources as they don't require a separate address space, although performance comparisons between the two implementations are out of the scope of this work, being left for future works. JSucuri extends the original Sucuri by implementing the means to concurrent kernel execution and memory operations in GPU via JavaCL [14]. In this way, different nodes of the dataflow program can copy data in and out of the GPU, as well as issue kernel executions, sharing the same OpenCL context and command queues among them.

The rest of this work is organized as follows: Section II presents and discusses the state-of-art dataflow libraries and distributed solutions for heterogeneous computing using CPU and GPU. Section III describes the JSucuri library and its underlying architecture. Section **??** presents the stream processing benchmark programs used in this work and Section IV discusses the experimental results based on those programs. Finally, Section V concludes this work and presents some ideas for the future.

II. Related Works

Besides Sucuri [13], [15], other works [16] sought to use the Dataflow model as a parallel programming model using high-level languages constructs. However, these have not included support for the use of GPUs nor for asynchronous communication and concurrent kernel execution.

The work presented in [17] (rCUDA) implements a virtualization framework for remote GPUs. It allows the usage of Nvidia GPUs remotely, providing a virtualization service on clusters. In this way, some nodes which have GPUs can be accessed in a transparent way without the need to modify the code, because of a dynamic library that translates CUDA [18] calls. This framework supports version 8 of CUDA without the graphical functions. Also, it supports Remote Direct Memory Access (RDMA) through InfiniBand and TCP/IP networks.

The DistributedCL [19] framework is similar to rCUDA, but it uses the existing OpenCL API to enable its use in a distributed processing environment on different GPU vendors. This framework creates the abstraction of a single OpenCL platform. It assumes that the applications that use it can perform asynchronous communication. Thus, the data can be sent simultaneously while the commands for the execution of the *kernel* are being executed. The use of asynchronous communication together with the storage of several commands before sending them through the network is pointed out as responsible for the reduction in communication overhead. Both rCUDA and DistributedCL have used asynchronous communication to reduce communication overhead in distributed environments. Furthermore, modern GPUs have features such as concurrent kernel execution and data copy superposition. Concurrent execution of a kernel can lead to an increase in the throughput of programs that may benefit from this feature.

In this work, we implemented a Java-based dataflow library inspired in Sucuri [13] to explore the concurrent execution of kernels and memory copies, ranging from 1 to 6 command queues, in order to verify if there's been an increase in the throughput of the application with concurrent execution of kernels and memory operations. Each kernel represents an iteration of a streaming application described using the Dataflow model. We have explored the use of asynchronous communication together with the Dataflow execution model in a heterogeneous multi-core environment to increase the parallelism exploitation using Dataflow and also by taking advantage of the concurrent execution of kernels and data copy.

III. JSUCURI

The overall architecture of JSucuri is depicted in Fig.1, where it is possible to observe some of the major components of the library, such as the `Scheduler`, the `Workers` and the application `Dataflow Graph`.

Most of the architectural components of JSucuri are implemented exactly as in Sucuri, except for the Worker, which is mapped onto Java threads instead of processes. Also, message passing interface (MPI) support is not implemented. Each worker thread lives on the same machine and shares the same object reference to the operand queue and the dataflow graph. While the operand queue is thread-safe, the graph is not, which means that race conditions may occur if different node functions are executing code on the same object reference at the same time. Thus, caution must be taken to ensure that different threads do not step on each other, *i.e.*, do not execute on the shared objects at the same time. One way to ensure synchronization is to provide different instances of an object to each node function.

The java library spin-off inherits the same class hierarchy of Sucuri. Because of that, the same dataflow program described in Python can be described in Java, effortlessly, with some minor differences specific to each programming language. For instance, the function code that is handed to each dataflow node to execute is specified as a Java abstract class, so that the program of each node can be declared as an anonymous class and instantiated at the same time, during the dataflow program specification. The abstract class, named `NodeFunction`, defines a single abstract method that should be overridden in order to specify the function that the node should execute. Moreover, anonymous classes have scope access to their outer class static attributes, which allows JavaCL objects to be shared among node functions, enabling each Worker to issue JavaCL commands independently. Besides, JavaCL is thread-safe. Thus, if two or more threads call the same JavaCL method, they will be synchronized.

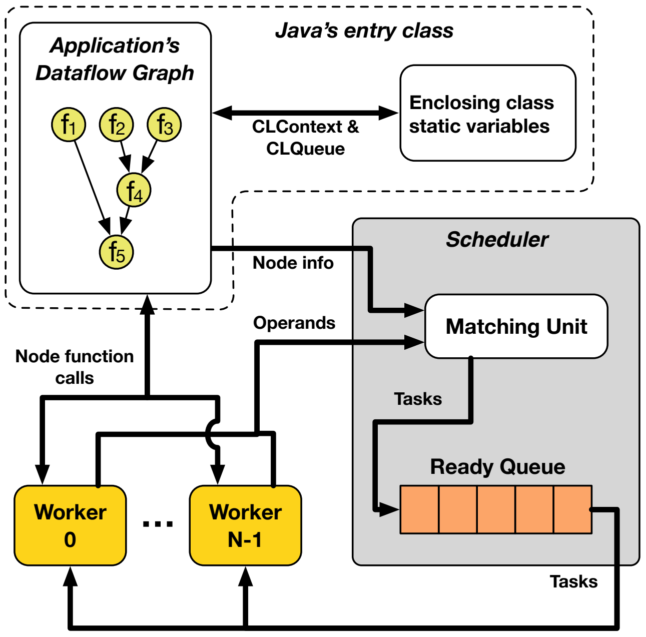


Fig. 1: The JSucuri Architecture. Each Worker is implemented as a Java Thread. Functions are handed to each node of the dataflow graph and they have scope access to the enclosing's class static variables, which is useful for sharing JavaCL objects among node functions.

Fig. 2 shows the dataflow graph for the Volume Ray-Casting application and its JSucuri code snippet, with a few nodes invoking the GPU through JavaCL. The graph is composed of 5 feeder nodes and 4 nodes with node functions assigned to each of them. The feeder nodes are the ones that simply produce a value to other nodes. Feeder nodes have their execution started immediately because they do not contain input operands. All the subsequent nodes have their execution started by a worker thread as soon as their input operands are available, as follows:

- `readVolumeNode`: the node expects the volume file path and the dimensions array to be produced in its ports 0 and 1. The node function readVolume is then executed with the respective input arguments and produces the volume data (array of floats) to copyInVolumeNode.
- `copyInVolumeNode`: with the volume data available, this node runs the `assyncCopyIn` function, which uses the JavaCL context in the outer class scope to enqueue the volume data copy into the GPU. Once enqueued, it passes on JavaCL event objects to the node responsible for kernel execution. The event

objects will be used to indicate when the kernel can start its execution, *i.e.*, whether the volume has been copied already into the GPU or not.

- `execKernelNode`: having received the values from the feeder nodes and the indication that the volume data is available in the GPU, the node function `assyncKernel` can trigger the kernel execution. It is important to observe that before the execution of the kernel there is a lot of operations that need to be executed to read and compile kernel code, which can be executed while the values are still being copied to the GPU. Thus, it represents an example of overlapping data and computation, which contributes to increasing the overall system performance.
- `copyOutVolumeNode`: finally, the node function `assyncCopyOut` waits for the kernel execution to finish and enqueue the read of the resulting data from the GPU.

Observe that the code presented in Fig. 2 is not complete for the sake of simplicity. Also, this version of Volume Ray-Casting is very simple, as it only processes a single volume. Stream of volumes will be later described in Section IV.

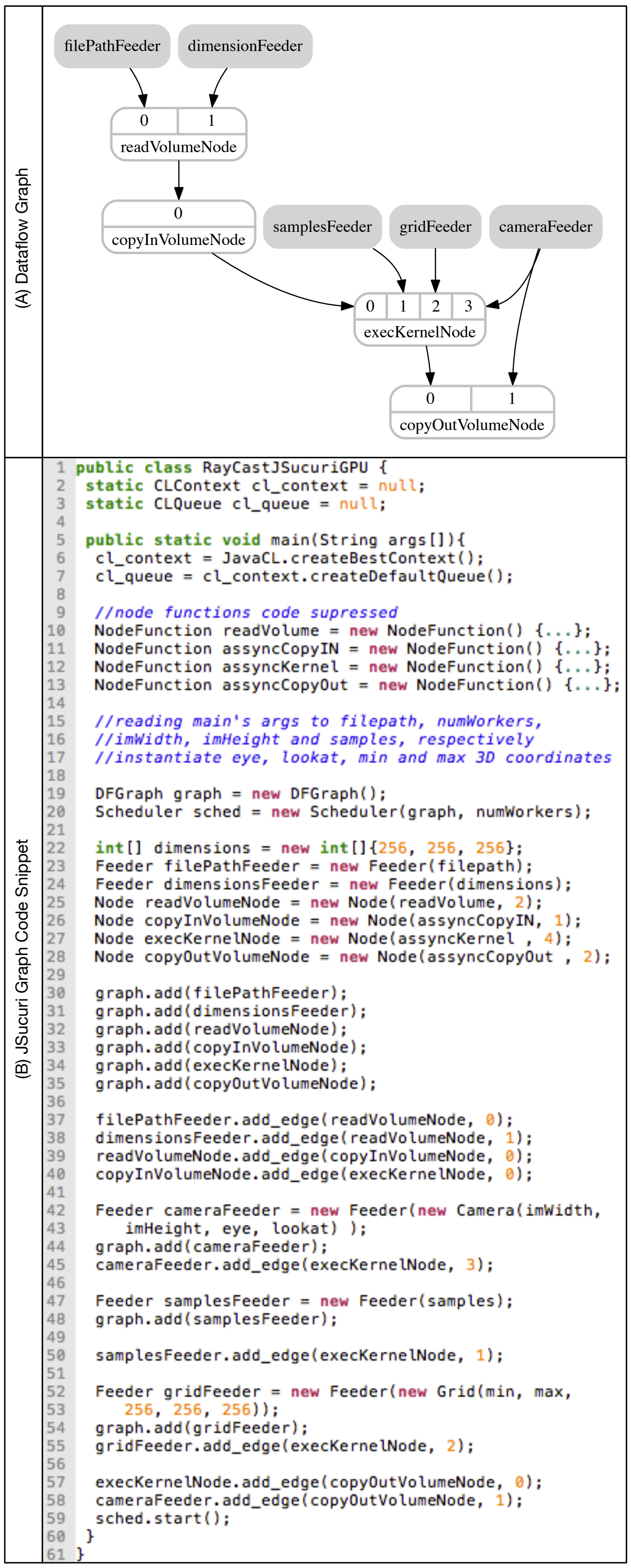
Stream dataflow processing applications present a regular flow of input data, with many processing iterations, as shown in the dataflow graph depicted in Fig. 3, with many queues. Independent iterations can be executed in parallel, with each iteration using one queue. Thus, if N queues are available, it means that N instances of the dataflow graph can be executed concurrently. Even if multiple command queues are used for a single OpenCL-enabled device, it is possible to issue commands for copy operations and kernel execution for concurrent execution.

IV. Experimental Results

This section presents experimental results of performance and memory consumption on a set of benchmark applications chosen for Stream Dataflow implementation in JSucuri: Volume Ray-Casting, Path-Tracing and Sobel Filter. They are relevant graphics processing applications with high-demanding performance constraints that often benefits from the stream processing model of computation.

The Volume Ray-Casting and Path-Tracing algorithms are well-known 3-D rendering algorithms. The first is able to render 3-D volume information captured from CT-Scan and MRI machines, while the latter is able to render 3-D virtual scenes with even more realistic light effects, including shadow smoothing effects and indirect lighting, like 3-D animation movies. Both algorithms operate by firing rays towards the model to be rendered. However, the Path-Tracing algorithm produces one or more secondary beams in different directions from diffuse surfaces, while Volume Ray-Casting does not produce secondary rays at all. Thus, the color of each *pixel* in Path-Tracing is calculated by taking into account several pieces of information gathered by each ray-object collision, *i.e.*, intersection, all over the 3-D scene.

The Sobel filter is an image processing algorithm used to emphasize the edges of the elements (objects, people, etc.) present in an image. So it is a widely used algorithm



```java
public class RayCastJSucuriGPU {
 static CLContext cl_context = null;
 static CLQueue cl_queue = null;

 public static void main(String args[]){
  cl_context = JavaCL.createBestContext();
  cl_queue = cl_context.createDefaultQueue();

  //node functions code supressed
  NodeFunction readVolume = new NodeFunction() {...};
  NodeFunction assyncCopyIN = new NodeFunction() {...};
  NodeFunction assyncKernel = new NodeFunction() {...};
  NodeFunction assyncCopyOut = new NodeFunction() {...};

  //reading main's args to filepath, numWorkers,
  //imWidth, imHeight and samples, respectively
  //instantiate eye, lookat, min and max 3D coordinates

  DFGraph graph = new DFGraph();
  Scheduler sched = new Scheduler(graph, numWorkers);

  int[] dimensions = new int[]{256, 256, 256};
  Feeder filePathFeeder = new Feeder(filepath);
  Feeder dimensionsFeeder = new Feeder(dimensions);
  Node readVolumeNode = new Node(readVolume, 2);
  Node copyInVolumeNode = new Node(assyncCopyIN, 1);
  Node execKernelNode = new Node(assyncKernel , 4);
  Node copyOutVolumeNode = new Node(assyncCopyOut , 2);

  graph.add(filePathFeeder);
  graph.add(dimensionsFeeder);
  graph.add(readVolumeNode);
  graph.add(copyInVolumeNode);
  graph.add(execKernelNode);
  graph.add(copyOutVolumeNode);

  filePathFeeder.add_edge(readVolumeNode, 0);
  dimensionsFeeder.add_edge(readVolumeNode, 1);
  readVolumeNode.add_edge(copyInVolumeNode, 0);
  copyInVolumeNode.add_edge(execKernelNode, 0);

  Feeder cameraFeeder = new Feeder(new Camera(imWidth,
     imHeight, eye, lookat) );
  graph.add(cameraFeeder);
  cameraFeeder.add_edge(execKernelNode, 3);

  Feeder samplesFeeder = new Feeder(samples);
  graph.add(samplesFeeder);

  samplesFeeder.add_edge(execKernelNode, 1);

  Feeder gridFeeder = new Feeder(new Grid(min, max,
     256, 256, 256));
  graph.add(gridFeeder);
  gridFeeder.add_edge(execKernelNode, 2);

  execKernelNode.add_edge(copyOutVolumeNode, 0);
  cameraFeeder.add_edge(copyOutVolumeNode, 1);
  sched.start();
 }
}
```

Fig. 2: Dataflow graph for the Volume Ray-Casting benchmark and its JSucuri code snippet. Feeder nodes (with no input ports) are presented in gray, while the other nodes are presented in white. For the sake of simplicity, the code for each node function is not presented, as well as some variables declaration and initialization.

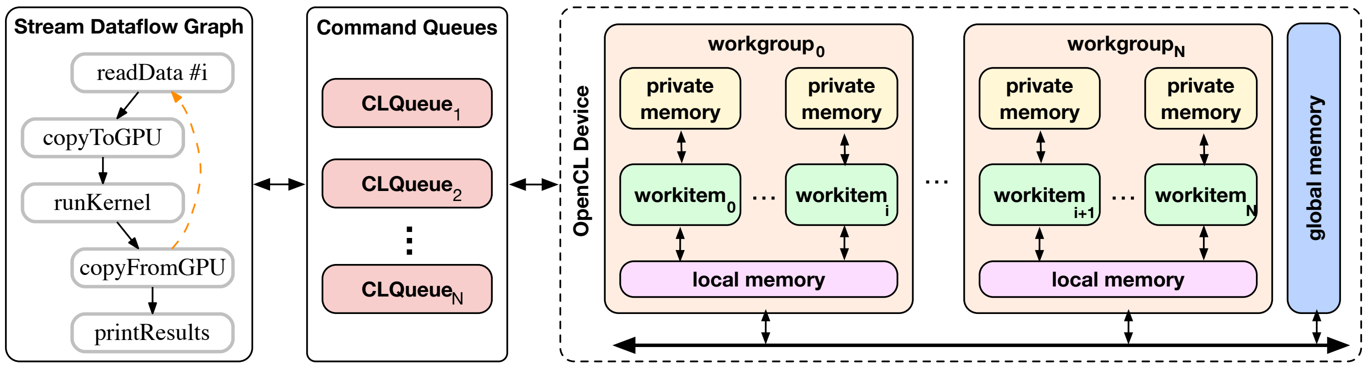


Fig. 3: Stream dataflow programs present a regular flow of input data (readData #i), with many processing iterations. Using multiple queues to issue OpenCL commands to the device increases data throughput and performance by issuing memory operations and kernel execution in parallel.

in computer vision applications. The filter determines the intensity value of each *pixel* by calculating the gradient vector along the x and y axis. Thus, two convolution matrices are used for each axis. In this dataflow implementation, the filter is applied to a sequence of frames extracted from a movie to reproduce the behavior of a stream application.

A stream dataflow graph was implemented for each benchmark application varying from 1 to 6 command queues. Moreover, the number of JSucuri Workers varies from 2 to 12 (twice the number of command queues for each configuration). This is to ensure that there will be enough Workers to issue concurrent copies and kernels to the GPU. The baseline for performance comparison is the sequential implementation of each benchmark application. The experimental setup consists of an AMD FX 8-core processor with 8GB of RAM, GeForce 750i Nvidia GPU, running Ubuntu 14.04 OS and Java Virtual Machine (JVM) 1.6.

A. *Performance Results*

In general, regardless of the image resolution and number of processed frames, it is possible to observe that the stream dataflow applications perform better than their serial implementation when using around 3 command queues, achieving about 8× and 9× speedup for Volume Ray-Casting and Sobel stream dataflow algorithms, respectively, as shown in Fig. 4 and Fig. 5. Besides, Sobel performs better with 4 queues for higher resolution images, possibly because it increases the computational load of the algorithm, which is much lower than that of the Volume Ray-Casting and Path-Tracing algorithms.

Beyond 3 queues or less the speedup tends to decrease for most stream applications. The reason for such slowdown is possibly due to the overhead of managing more queues than actually necessary. Also, more queues mean that more commands can be issued concurrently. However, the system's setup only disposes of a single GPU, possibly flooding it with concurrent commands that in the end need to be synchronized (serialized) by the thread-safe JavaCL API.

On the other hand, the Path-Tracing stream dataflow implementation achieves the best speedup: around 645× faster than its serial counterpart, as shown in Fig. 6. The reason for such astonishing acceleration, especially when using less queues, is because the algorithm's rays can be processed independently, often yielding almost linear speedup as more processing elements are used in parallel, which explains why the speedup is close to the number of CUDA-Cores available in the GPU that is being used in the experiments. Also, the Path-Tracing serial implementation is very naive, *i.e.*, does not implement any spatial subdivision techniques to optimize the

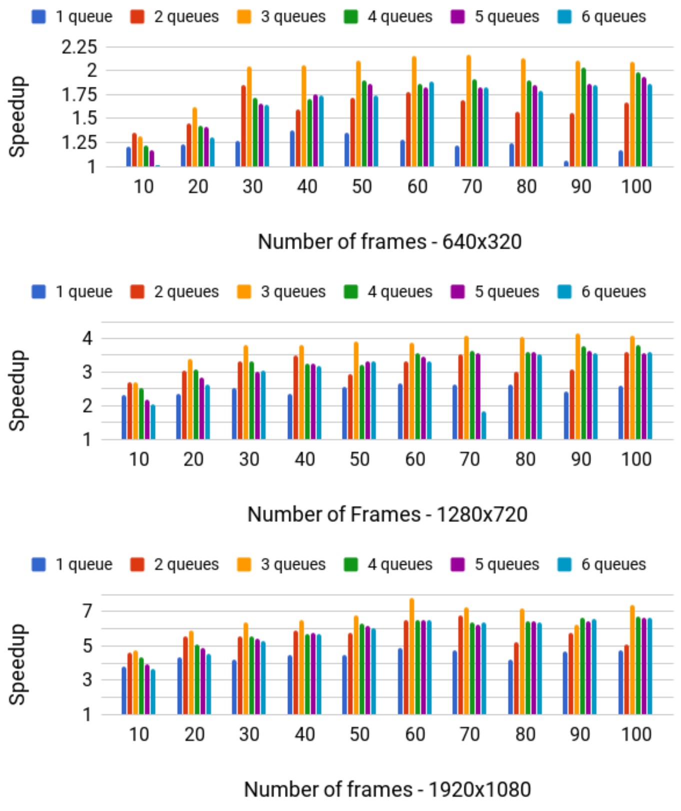


Fig. 4: Speedups for the Volume Ray-Casting stream processing application in dataflow using up to 6 command queues and producing up to 100 frames.

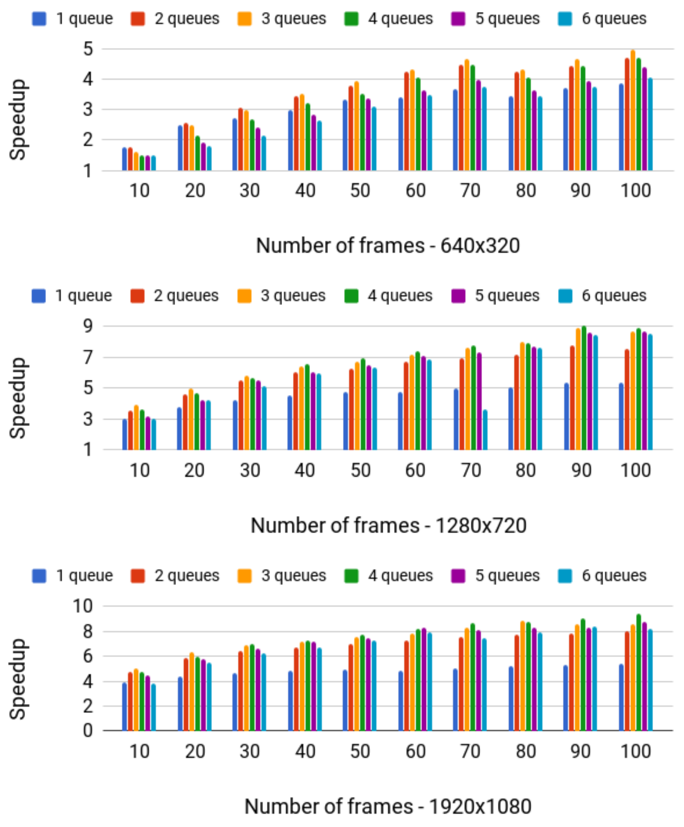


Fig. 5: Speedups for the Sobel filter stream processing application in dataflow using up to 6 command queues and producing up to 100 frames.

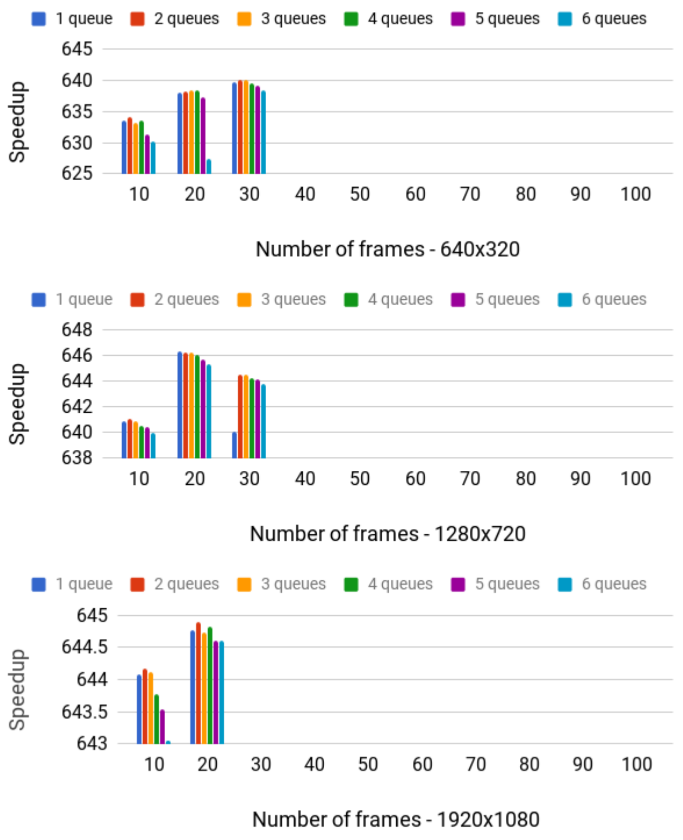


Fig. 6: Speedups for the Path-Tracing stream processing application in dataflow using up to 6 command queues and producing up to 30 frames.

number of ray-object collision tests. Thus, any effort towards making it parallel will be effective.

Moreover, each primary ray, *i.e.*, camera ray, is sampled 2048 times and can bounce up to 8 times from one object to another, which roughly yields a total of $1920 \times 1080 \times 2048 \times 8$ ray-collision tests per processed frame. Because of that amount of work, it was not possible to measure the algorithm's serial and parallel execution times beyond 20 processed frames of 1920×1080 pixels and 30 processed frames in lower resolutions.

B. *Memory Copy Results*

The results presented in Fig. 7 show the total size of memory copies in and out the GPU for varying numbers of command queues. It is possible to observe that as more queues are used the higher the total size of data that is being transferred, for all benchmark applications. Also, higher image resolutions collaborate to increase the total size of data copies, because each image is produced inside the GPU's global memory and later on transferred to the host's machine.

The volume ray-casting application is the one that used memory the most: up to 600 Megabytes. This is due to the 3-D volume data that needs to be copied to the GPU memory for processing. Each volume consists of $256 \times 256 \times 256$ voxels of 16-bits each, yielding 16 Megabytes per 3-D volume. Thus, using 6 queues means that up to 6 volumes can be copied to the GPU concurrently. The rest of the data consists of several arrays required for the algorithm's kernel execution and the respective rendered images at the specified resolution.

On the other hand, the path-tracing application is the one that used memory the least: up to 150 Megabytes. Once more, this is due to the 3-D scene data that needs to be copied to the GPU. It consists of 9 spheres which are used to compose the scene. Each sphere consumes no more than 416 Kilobits of information. The rest of the data also consists of many arrays required for the algorithm's kernel execution and the respective rendered images at the specified resolution.

V. Conclusion & Ideas for Future Work

This work presented JSucuri, a dataflow programming library for high-performance computing on heterogeneous systems using CPU and GPU. It implements high-level constructs with multiple command queues to enable the superposition of memory operations and kernel executions on GPUs using JavaCL. A set of graphics processing applications was chosen for Stream Dataflow implementation in JSucuri, yielding significant speedups when using a configuration of multiple command queues.

In the future, experimental results for more than one GPU should be performed in order to evaluate actual parallel

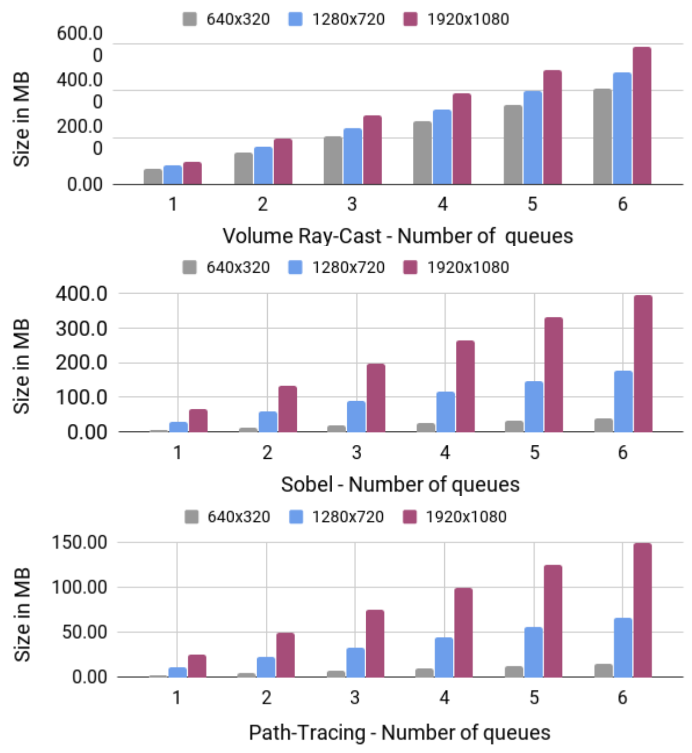


Fig. 7: Memory copies in Megabytes to and from the GPU for up to 6 command queues.

kernel execution besides concurrent kernel and memory copy superposition. More benchmarks applications also needs to be implemented using the stream dataflow model to further experiment JSucuri on different classes of applications. Finally, a message passing interface would allow nodes at a distinct networks to transmit data in and out the available GPUs on the system.

Acknowledgment

The authors would like to thank CAPES, CNPq and FAPERJ for the financial support given to the authors of this work.

References

[1] G. E. Moore, "Readings in computer architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," in *IEEE Computer*, 1996, pp. 2–11.

[3] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ser. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. [Online]. Available: http://doi.acm.org/10.1145/642089.642111

[4] T. A. Alves, L. A. Marzulo, F. M. Franca, and V. S. Costa, "Trebuchet: exploring TLP with dataflow virtualisation," *International Journal of High Performance Systems Architecture*, vol. 3, no. 2/3, p. 137, 2011.

[5] L. A. Marzulo, T. A. Alves, F. M. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *Parallel Computing*, vol. 40, no. 10, pp. 661 – 680, 2014.

[6] "Tbb flowgraph," https://www.threadingbuildingblocks.org/tutorial-intel-tbb-flow-graph, accessed on September 9, 2017.

[7] J. Wozniak, T. Armstrong, M. Wilde, D. Katz, E. Lusk, and I. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 95–102.

[8] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633 – 652, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing.

[9] G. Matheou and P. Evripidou, "FREDDO: an efficient framework for runtime execution of data-driven objects," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, 2016, pp. 265–273.

[10] R. Giorgi, R. M., F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R., A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, pp. –, 2014, available online 18 April 2014.

[11] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011-03-01 2011.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Hrault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing." *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[13] T. A. O. Alves, B. F. Goldstein, F. M. G. Frana, and L. A. J. Marzulo, "A minimalistic dataflow programming library for python," in *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, Oct 2014, pp. 96–101.

[14] O. Chafik, "Javacl: Opencl bindings for java," https://github.com/nativelibs4java/JavaCL, 9 2015, accessed: 2017-09-09.

[15] R. J. N. Silva, B. Goldstein, L. Santiago, A. C. Sena, L. A. J. Marzulo, T. A. O. Alves, and F. M. G. Frana, "Task scheduling in sucuri dataflow library," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct 2016, pp. 37–42.

[16] J. Bosboom, S. Rajadurai, W.-F. Wong, and S. Amarasinghe, *StreamJIT: A commensal compiler for high-performance stream programming*. ACM, 2014, vol. 49, no. 10.

[17] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "A complete and efficient cuda-sharing solution for hpc clusters," *Parallel Computing*, vol. 40, no. 10, pp. 574–588, 2014.

[18] "Cuda c programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4idxrP1Vq/, accessed: 2017-05-30.

[19] A. Tupinambá and A. Sztajnberg, "Distributedcl: a framework for transparent distributed gpu processing using the opencl api," in *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*. IEEE, 2012, pp. 187–193.