An optimised dataflow engine for GPGPU stream processing

Marcos Paulo Rocha* and
Felipe M.G. França

Engenharia de Sistemas e Ciência da Computação,
Universidade Federal do Rio de Janeiro,
Rio de Janeiro, RJ, Brazil
Email: mrocha@cos.ufrj.br
Email: felipe@cos.ufrj.br
*Corresponding author

Alexandre Solon Nery

Departamento de Engenharia Elétrica,
Universidade de Brasília,
Brasília, DF, Brazil
Email: anery@unb.br

Leandro S. Guedes

Departamento de Informática,
Instituto Federal de Educação,
Ciência e Tecnologia de Mato Grosso do Sul,
Corumbá, MS, 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 expose and facilitate the parallelism exploitation for a wide class of applications. Thus, instead of following the program order, different operations can be run in parallel as soon as their input operands become available. 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. Moreover, new contributions in respect to concurrency analysis and the Stream processing parallel model in dataflow are presented.

Keywords: dataflow; heterogeneous systems; high-performance computing.

Reference to this paper should be made as follows: Rocha, M.P., França, F.M.G., Nery, A.S. and Guedes, L.S. (2019) 'An optimised dataflow engine for GPGPU stream processing', *Int. J. Grid and Utility Computing*, Vol. 10, No. 3, pp.248–257.

Biographical notes: Marcos Paulo Rocha Graduated in Computer Science from Universidade do Estado do Rio de Janeiro (2014), having received his MSc (2017) from Universidade Federal do Rio de Janeiro.

Felipe M.G. França graduated in Electrical Engineering from Universidade Federal do Rio de Janeiro (1982), having received his MSc in Computer Science from Universidade Federal do Rio de Janeiro (1987) and his PhD in Neural Systems Engineering from Imperial College of Science Technology And Medicine (1994). He is currently acting on the following subjects: artificial neural networks, complex systems, computer architecture, cryptographic circuits, distributed algorithms, computational intelligence, collective robotics, complex systems, intelligent transportation systems and parallel computing.

Alexandre Solon Nery graduated in Computer Science from Universidade Católica de Brasília (2006), having received his MSc (2010) and DSc (2014) from Universidade Federal do Rio de Janeiro. He is currently acting on the following subjects: reconfigurable computing, computer architecture, distributed and parallel computing (Cloud/Edge/Fog/In-Situ).

Leandro S. Guedes graduated in Computer Science from Universidade Federal de Pelotas (2013), having received his MSc (2016) in Computer Science from Universidade Federal do Rio Grande do Sul. He is currently acting on the following subjects: computer graphics, information visualisation, human-computer interaction and distributed systems.

This paper is a revised and expanded version of a paper entitled 'Dataflow Programming for Stream Processing' presented at the '2017 International Symposium on Computer Architecture and High-Performance Computing Workshops (SBAC-PADW)', Campinas, Brazil, 17–20 October 2017.

1 Introduction

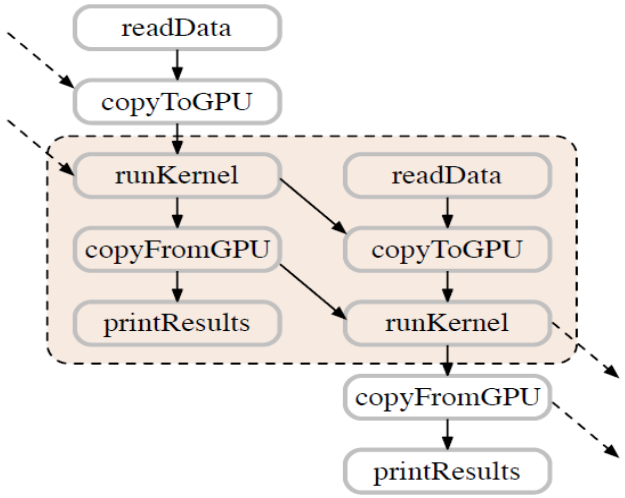
Moore (2000) predicted that the number of transistors that can be fit into an integrated chip would nearly double almost every two years. While such greater numbers of transistors over the years enabled significant advances in the processor's microarchitecture, the transistor miniaturisation trend led to insulation and heat dissipation problems, prohibiting further increases of 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 – Grasso et al. (2014)), Chip Multi-Processors (CMPs – Olukotun et al. (1996)) and custom-made accelerators in Field-Programmable Gate Arrays (FPGAs – Uliana et al. (2013)). However, building parallel programs is not a trivial task. In fact, it often requires expertise and many hours of hard work to fully optimise an application to run on a particular parallel architecture.

The Dataflow paradigm first presented in Dennis and Misunas (1975) 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 the program order. This means that independent nodes can potentially run in parallel if there are available idle processing elements. Regarding edge/fog computing services, each dataflow graph node may also work, for instance, like a fog node in the OpenFog Reference Architecture (Consortium, 2017), which is a consortium intended to help create and maintain the hardware, software and system elements necessary for fog computing.

Moreover, the recent development of embedded systems and Internet of Things (IoT) technologies created new efficiency and low latency challenges on traditional centralised cloud computing systems, as shown in Ai et al. (2017). To overcome such challenges, new technology is changing the centralised cloud computing architecture to edge devices on the network border, in a trend called fog or edge computing, as presented in Cerina et al. (2017) and Li et al. (2018). Hence, while some functions are better suited for cloud computing, others are naturally more advantageous to be carried out by fog nodes, as shown in Liu et al. (2018) and Ahn et al. (2017). Thus, the focus has shifted to bring specialised computation and communication devices nearer to the user. Each special computation unit, also known as hardware accelerator, is often different from the others, i.e., they have different architectures, programming models, etc. Hence, GPGPUs are a very popular kind of hardware accelerator in heterogeneous systems and is normally employed in highly demanding applications in terms of performance and throughput. Also, CMPs have multiple processing cores that can be used together for parallel processing. More recently, FPGAs have also become popular in high-performance computing and distributed applications, due to its energy efficiency, especially when compared to GPGPUs, as presented in Tan et al. (2017). However, efficiently using such heterogeneous accelerators is still a challenge.

This work expands our previous workshop paper (Rocha et al., 2017) with new performance results and additional concurrency analysis for the Volume Ray-Casting implementation. Also, it further discusses the JSucuri dataflow programming library and runtime, as well as the state-of-the-art related works. JSucuri aims to enable dataflow high-performance computing on heterogeneous systems using CPU and GPU. Thus, ordinary dataflow nodes will be processed by the available CPU cores, while specialised dataflow nodes will be processed by the available GPU cores. 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. It is important to highlight that Java is still the leading programming language in the market according to TIOBE Software (2018), competing with C/C++ and Python. 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. Hence, despite the use of shared memory for means of communicating, JSucuri dataflow library evaluates the potential of specialised stream processing in GPUs that can be present in fog/edge/in-situ nodes of a given application dataflow graph. JSucuri extends the original Sucuri by implementing concurrent kernel execution and memory copy operations in GPU via JavaCL (Chafik, 2015), as shown in Figure 1. In this way, different nodes of the dataflow program can copy data to a GPU and retrieve its expected results. Also, different nodes can issue kernel executions that share the same OpenCL context and command queues, enabling, for instance, parallel image processing of a stream of images.

Figure 1 Concurrent kernel execution and memory operations in GPU using JSucuri dataflow programming model



The rest of this work is organised as follows: Section 2 presents and discusses the state-of-art Dataflow libraries and distributed solutions for heterogeneous computing using CPU and GPU. Section 3 describes the JSucuri library and its underlying architecture. Section 4 discusses the experimental results based on those programs. Finally, Section 5 concludes this work and presents some ideas for the future.

2 Related works

Recent research proposes dataflow processing in different levels of abstraction and granularities, as shown in Alves et al. (2011), Marzulo et al. (2014), Balaji (2015),Wozniak et al. (2013),Wilde et al. (2011), Matheou and Evripidou (2016), Giorgi et al. (2014), Duran et al. (2011) and Bosilca et al. (2012). In the last decade Dataflow has matured and developed into an efficient and straightforward parallel programming model, including edge/fog/in-Situ computing capabilities, as presented in Carvalho et al. (2017). In essence, any processing operation (e.g., instructions, functions, programs, etc.) can be connected to each other in a Dataflow dependency graph, thus allowing programmers to harvest the potential of modern parallel systems, as comprehensively presented in Alves et al. (2018). Besides Sucuri dataflow python library presented in Alves et al. (2014) and Silva et al. (2016), other works, such as in Bosboom et al. (2014), 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 Peña et al. (2014), also known as rCUDA, implements a virtualisation framework for remote GPUs. It allows the usage of Nvidia GPUs remotely, providing a virtualisation 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 (Cuda C Programming Guide (n.d.)) 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.

Tupinambá and Sztajnberg (2012) proposed the DistributedCL framework, which is similar to rCUDA. The framework uses an 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 (Alves et al., 2014). Although every thread communication is handled via shared-memory, as will be described in Section 3, JSucuri dataflow library evaluates the potential of specialised stream processing using GPUs that may be present in fog/edge nodes of a given application dataflow graph. The library explores the concurrent execution of kernels and memory copies, ranging from one to six command queues, in order to increase 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.

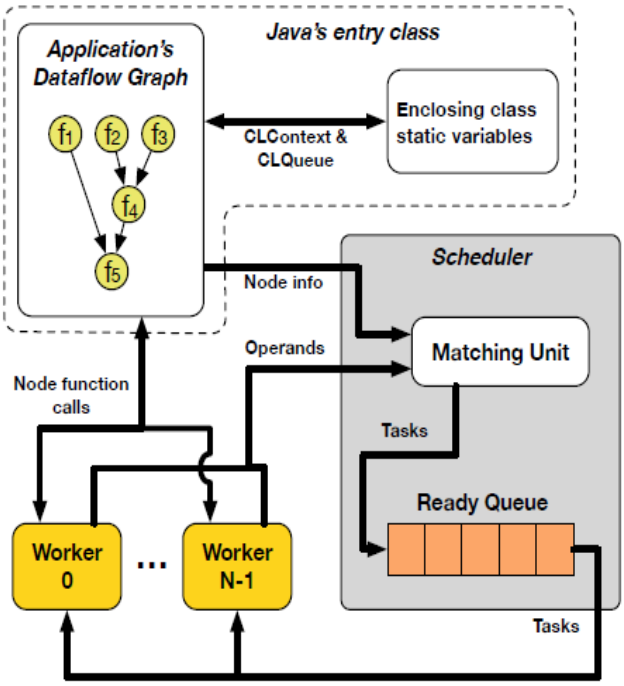
3 JSucuri

The overall architecture of JSucuri is depicted in Figure 2, 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 the Worker class, 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 synchronisation is to provide different instances of an object to each node function.

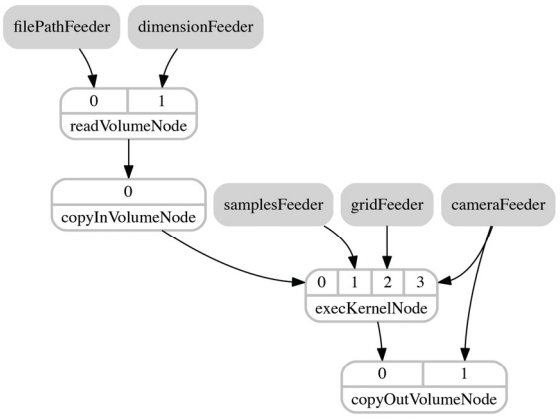
Figure 2 The JSucuri architecture. Each worker is implemented as a Java thread. Functions are handed to each node of the dataflow graph and they have scope access to the enclosing's class static variables, which is useful for sharing JavaCL objects among node functions



The proposed dataflow Java library 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 synchronised.

Figure 3 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 initialisation

(A) Dataflow Graph



(B) JSucuri Graph Code Snippet

```
1  public class RayCastJSucuriGPU {
2   static CLContext cl_context = null;
3   static CLQueue cl_queue = null;
4
5   public static void main(String args[]){
6    cl_context = JavaCL.createBestContext();
7    cl_queue = cl_context.createDefaultQueue();
8
9    //node functions code supressed
10   NodeFunction readVolume = new NodeFunction() {...};
11   NodeFunction assyncCopyIN = new NodeFunction() {...};
12   NodeFunction assyncKernel = new NodeFunction() {...};
13   NodeFunction assyncCopyOut = new NodeFunction() {...};
14
15   //reading main's args to filepath, numWorkers,
16   //imWidth, imHeight and samples, respectively
17   //instantiate eye, lookat, min and max 3D coordinates
18
19   DFGraph graph = new DFGraph();
20   Scheduler sched = new Scheduler(graph, numWorkers);
21
22   int[] dimensions = new int[]{256, 256, 256};
23   Feeder filePathFeeder = new Feeder(filepath);
24   Feeder dimensionsFeeder = new Feeder(dimensions);
25   Node readVolumeNode = new Node(readVolume, 2);
26   Node copyInVolumeNode = new Node(assyncCopyIN, 1);
27   Node execKernelNode = new Node(assyncKernel , 4);
28   Node copyOutVolumeNode = new Node(assyncCopyOut , 2);
29
30   graph.add(filePathFeeder);
31   graph.add(dimensionsFeeder);
32   graph.add(readVolumeNode);
33   graph.add(copyInVolumeNode);
34   graph.add(execKernelNode);
35   graph.add(copyOutVolumeNode);
36
37   filePathFeeder.add_edge(readVolumeNode, 0);
38   dimensionsFeeder.add_edge(readVolumeNode, 1);
39   readVolumeNode.add_edge(copyInVolumeNode, 0);
40   copyInVolumeNode.add_edge(execKernelNode, 0);
41
42   Feeder cameraFeeder = new Feeder(new Camera(imWidth,
43      imHeight, eye, lookat) );
44   graph.add(cameraFeeder);
45   cameraFeeder.add_edge(execKernelNode, 3);
46
47   Feeder samplesFeeder = new Feeder(samples);
48   graph.add(samplesFeeder);
49
50   samplesFeeder.add_edge(execKernelNode, 1);
51
52   Feeder gridFeeder = new Feeder(new Grid(min, max,
53      256, 256, 256));
54   graph.add(gridFeeder);
55   gridFeeder.add_edge(execKernelNode, 2);
56
57   execKernelNode.add_edge(copyOutVolumeNode, 0);
58   cameraFeeder.add_edge(copyOutVolumeNode, 1);
59   sched.start();
60  }
61 }
```

Figure 4 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

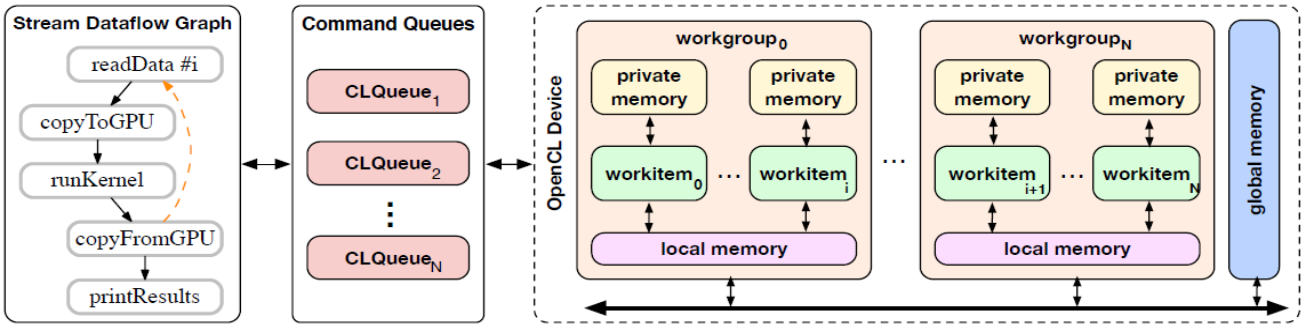


Figure 3 shows the dataflow graph for the Volume Ray-Casting application and its JSucuri code snippet, with a few nodes invoking the GPU through JavaCL. The graph is composed of five feeder nodes and four nodes with node functions assigned to each of them. The feeder nodes are the ones that simply produce 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 Figure 3 is not complete for the sake of simplicity. Also, this version of Volume Ray-Casting is very simple, as it only processes a single 3-D volume. Stream of volumes will be later described in Section 4.

Stream dataflow graphs differ from regular dataflow graphs mainly because the stream applications present a regular flow of input data, with many processing iterations, as shown in the dataflow graph depicted in Figure 4, with many queues. For instance, image processing applications apply specific operations (e.g., filters) to each frame of a video stream. Each frame is often independent of each other and can be processed concurrently or in parallel, provided that there are available resources. Therefore, stream processing graphs seek to concurrently issue multiple commands to many frames (iterations) of the video stream. Independent iterations can potentially 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.

4 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 often found in many popular benchmarks, such as in Bakhoda et al. (2009), Bienia (2011), Henning (2006), Bucek et al. (2018) and Yazdanbakhsh et al. (2016).. Such applications imposes high-demand 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 3-D model (or 3-D volume) to sample information about the objects 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 emphasise the edges of the elements (objects, people, etc.) present in an image. So it is a widely used algorithm 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 one to six command queues. Moreover, the number of JSucuri Workers varies from two to twelve (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, NVidia GeForce 750 TI GPU, running Ubuntu 14.04 OS and Java Virtual Machine (JVM) 1.6.

4.1 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 three command queues, achieving about 3000 × speedup for the Volume Ray-Casting algorithm, as shown in Figure 5, while the Path-Tracing stream dataflow implementation achieves about 640× speedup, as shown in Figure 6.

Figure 5 Speedups for the volume ray-casting stream processing application in dataflow using up to six command queues and producing up to 100 frames

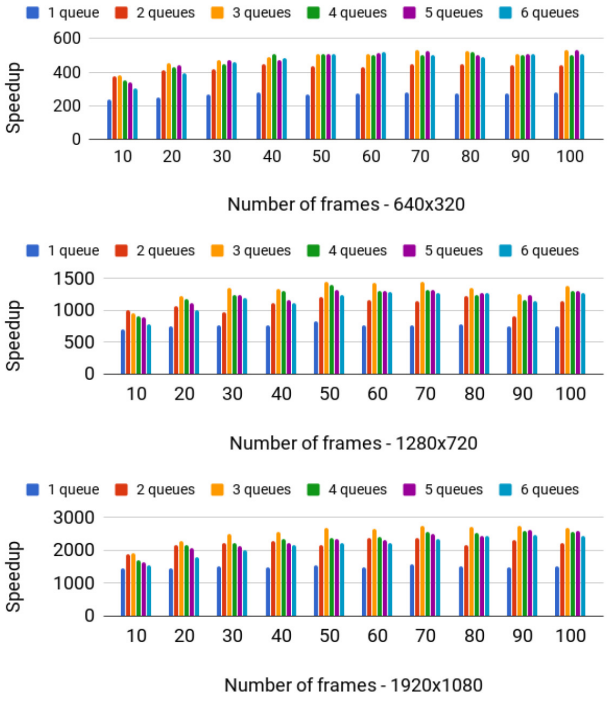
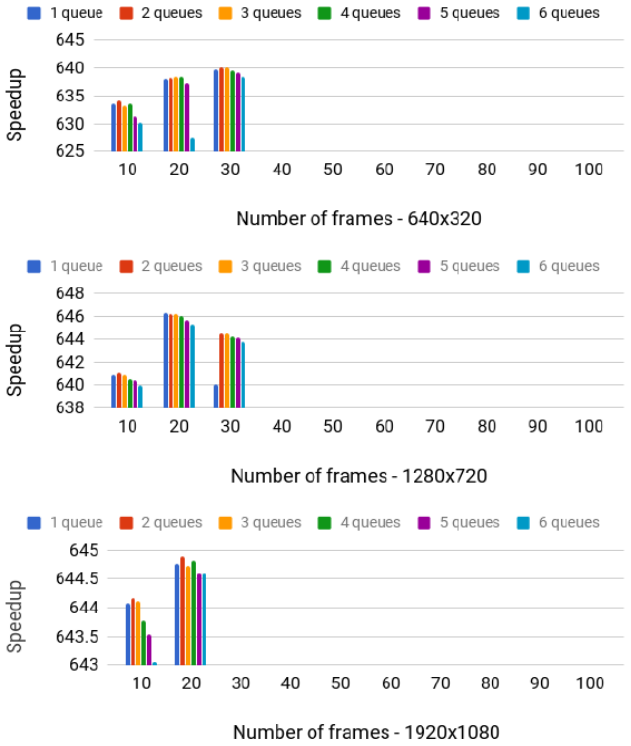


Figure 6 Speedups for the path-tracing stream processing application in dataflow using up to six command queues and producing up to 30 frames

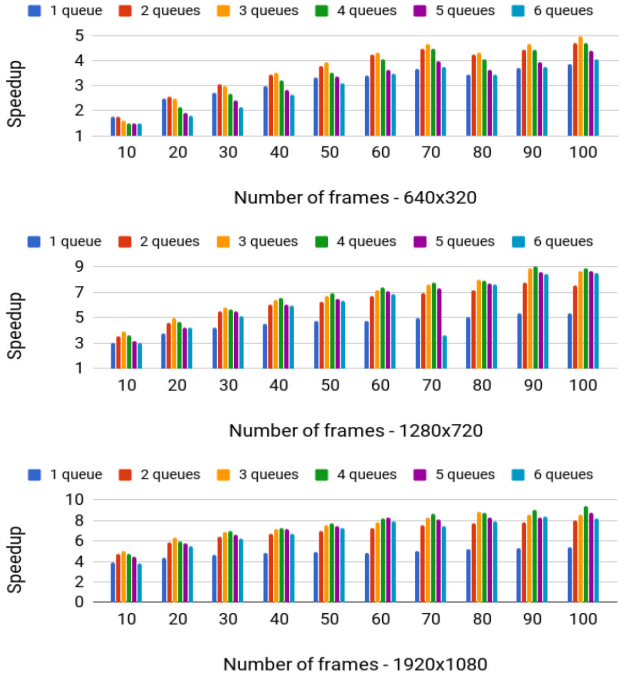


Algorithms based on Ray-Tracing rendering model fires lots of primary rays towards the 3-D scene in order to produce high-fidelity images from it. The Path-Tracing is a well-known complex computer graphics Monte Carlo rendering algorithm which simulates the path of light rays within a 3-D scene to produce photo-realistic images. The algorithm is used in 3-D animation movies and is often processed in dedicated rendering farms. Render quality depends on the number of Samples Per Pixel (SPP). A single frame can take several hours to be processed even using multiple processing units. Also, it can take more time depending on the image resolution and the number of samples per pixel. Thus, the total number of processed rays, i.e., ray-object intersection calculations, have a big impact on the algorithm's overall performance, also due to extensive floating-point computations in 64-bits representation to overcome precision problems on intersection tests. The Path-Tracing implementation fires up to 1920×1080 primary rays, i.e., camera rays, which are sampled 2048 times and can bounce up to eight times from one object to another in order to produce a decent image based on the Cornell Box 3-D model. Processing high-resolution frames also requires a large amount of memory, due to recursive function calls. On the other hand, the Ray-Casting implementation fires up to 1920×1080 primary rays that are sampled 1024 times each and does not produce any secondary rays, i.e., do not bounce on the volume's surface. Thus, the Path-Tracing algorithm needs to process far more rays than the Ray-Casting algorithm does. Yet, the Ray-Casting speedup is far higher than that of the Path-Tracing. This result comes from the fact that the Path-Tracing rendered 3-D scene is very small, yielding substantially fewer copies between the host and the GPU when compared to the other benchmark applications, as can be seen

later in Figure 8. In contrast, the Ray-Casting rendered volumes are much larger and take more time to move in and out of the GPU node. Despite that, in both algorithm implementations the rays can be processed independently, often yielding almost linear speedup as more processing elements are used in parallel. Moreover, the Path-Tracing serial implementation is very naive, i.e., does not implement any spatial subdivision techniques to optimise the number of ray-object collision tests. Thus, any effort towards making it parallel will be effective.

It was not possible to measure the serial and parallel execution times above 20 processed frames of 1920×1080 pixels and 30 processed frames in lower resolutions, because the algorithm never reached the end of its execution for such configurations and beyond. Still, further experimental results have been conducted using only eight samples for each primary ray (1280×720). Preliminary results indicate that the stream dataflow implementation was still able to achieve, on average, increasingly higher speedups ($115, 160, 200, 228, 244, 268\times$) against its serial implementation when processing more frames ($10, 20, 30, 40, 50, 60$), respectively, using three command queues. This suggests that the speedup indeed increases as the application is fed with more rays to process. More experiments will be carried out in future works.

Figure 7 Speedups for the Sobel filter stream processing application in dataflow using up to six command queues and producing up to 100 frames



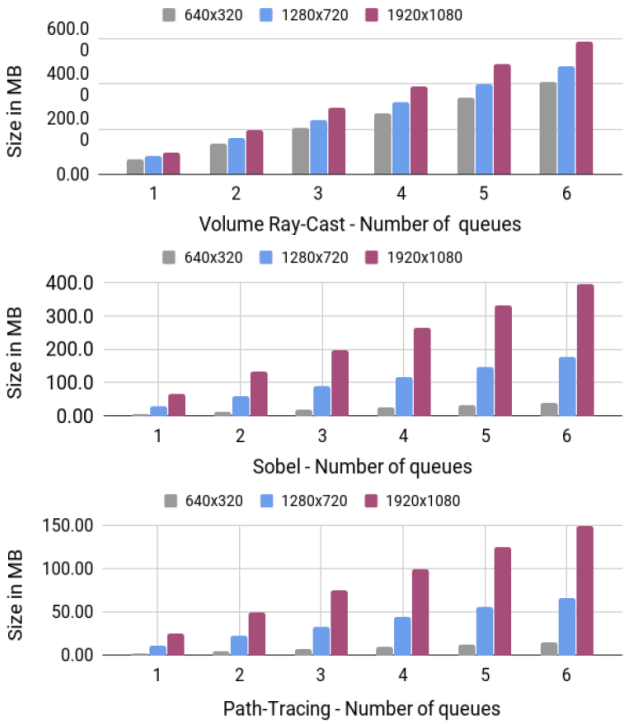
Beyond three queues or less the speedup tends to decrease for most stream applications. The reason for such slowdown is possibly due to the overhead of managing more queues than actually necessary. Also, more queues mean that more commands can be issued concurrently. However, the system's setup only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

Compared to the workshop paper, the better volume ray-casting performance results are not due to improvements in the algorithm, but due to new input parameters which increases the computation requirements and consequently increases the quality (resolution) of the rendered volume. Sobel is not reaching greater speed-ups because it is by far not as computationally intensive as Ray-Tracing algorithms. Yet, Sobel is evaluated and presented due to its stream-processing nature. Each benchmark application was executed once and thus standard deviation results are not presented. Further execution results will be performed in the future in order to get more reliable results.

4.2 Memory copy results

The results presented in Figure 8 show the total size of input and output copies between the host machine and the GPU for varying numbers of command queues, based on the previous work (Peña et al., 2014). Observe that memory copies and kernel execution can only occur concurrently if there is enough memory available for the input/output data that the kernel requires/produces. Thus, we evaluate the memory copy requirements in this work. The Volume Ray-Casting performance results presented earlier in Figure 5 do not correspond to the results presented here. Still, 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.

Figure 8 Memory copies in Megabytes to and from the GPU for up to six command queues



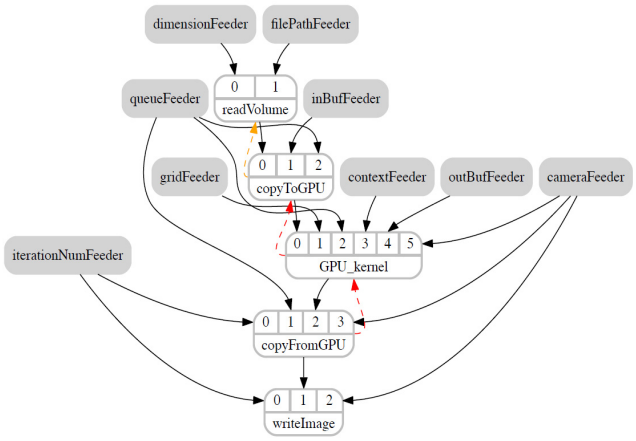
The volume ray-casting application is the one that used memory the most: up to 600 megabytes. This is due to the 3-D volume data that needs to be copied to the GPU memory for processing. Each volume consists of $256 \times 256 \times 256$ voxels of 16-bits each, yielding 16 megabytes per 3-D volume. Thus, using six 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 nine 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.

4.3 Concurrency profiling

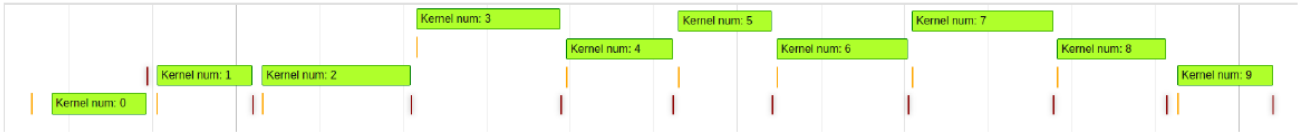
This section provides an analysis of the concurrent kernel execution of the Volume Ray-Casting stream processing implementation shown in Figure 9. The highlighted nodes are feeder nodes, i.e., the JSucuri dataflow runtime can start their execution immediately, as they do not depend on receiving any input data. These nodes are often used to introduce initial parameters to the dataflow program. The other nodes are scheduled to execute based on the availability of their input operands. Finally, the dashed edges are depicted only to simplify the dataflow graph description, avoiding the need to replicate the nodes to represent a concurrent execution as shown way back in Figure 1. Such edges represent the computation of a different volume of the application, overlapping copy and processing operations. Therefore, as soon as the GPU rendering kernel finishes its execution, the results are sent to the next processing node (copyFromGPU), which will copy the results from the GPU back to the host machine. Concurrently, upon the end of the GPU kernel execution a new copy of volume data into the GPU will be executed as soon as all the input operands (from zero to five in GPU kernel node) are available, including the dashed edge which will indicate that the GPU memory is ready to receive new rendering information from the kernel processing, as shown in GPU kernel node in Figure 9.

Figure 9 Volume ray-casting stream processing graph in JSucuri

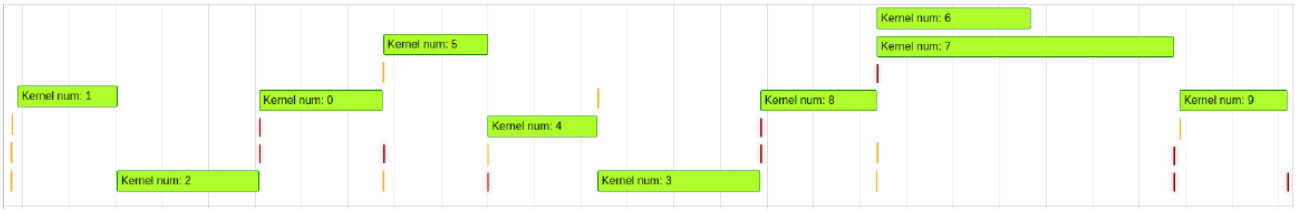


The concurrent execution of each GPU kernel interlaced with concurrent input/output GPU data copies is presented in Figures 10(a) and 10(b), for one and three command queues, respectively. It is possible to observe that there is no parallel execution when using only one command queue. Thus, all the input and output GPU copies are placed in-between each kernel execution. On the other hand, using three command queues enables the parallel execution of not only memory copies, but also of kernels. This is because the GPU used in the experiments allow parallel kernel execution. These results indicate that more command queues might further increase the dataflow program parallelism, improving its performance.

Figure 10 Volume ray-casting stream processing graph execution using one and three command queues, as time progresses from left to right. Each bar represents a GPU operation and its length represents the time it took it to complete. Moreover, the yellow and red bars corresponds to copies in and out the GPU, respectively, while the green bar corresponds to a kernel execution. Each kernel represents the rendering process of a different 3-D volume



(a) Execution using one command queue



(b) Execution using three command queues

5 Conclusions and ideas for future work

This work presented JSucuri, a dataflow programming library for high-performance computing on heterogeneous systems, which are at the core of most cloud/edge/in-situ modern architectures. The combination of CPUs and GPUs is very common in such distributed systems, especially to speedup the kernel part of an application on the GPU-side, leaving the least intensive parts of the application on the CPU-side. JSucuri implements high-level GPU constructs using JavaCL to enable the superposition of memory operations and kernel executions issuing multiple command queues to tasks of one or more GPUs present in the same machine, further increasing parallelism exploitation using each GPU as a hardware accelerator that operates concurrently within the machine idle CPU cores. The JSucuri library greatly helps to leverage the programming complexity of stream processing applications.

A set of relevant graphics processing applications was chosen for Stream Dataflow implementation in JSucuri, yielding significant speedups when using a configuration of multiple command queues. The key idea is to reduce the processing time of each stream application via concurrent kernel and memory copies, all issued by the JSucuri dataflow model. While such stream applications can only be executed in a shared-memory machine, they can still benefit from dataflow heterogeneous parallelism exploitation and can serve as a preliminary analysis on stream processing acceleration for distributed systems.

In the future, experimental results for more than one GPU should be performed in order to evaluate actual parallel kernel execution besides concurrent kernel and memory copy superposition. Also, a comparative analysis on the performance of Java-Sucuri against different Java GPU APIs will be considered for future iterations of this work. More benchmarks applications also need to be implemented using the stream dataflow model to further experiment JSucuri on different classes of applications. Moreover, a message passing interface would allow nodes at distinct networks to transmit data in and out all the available CPUs and GPUs on the distributed system. A comparative analysis on the performance of Java-Sucuri vs. Python-Sucuri is expected in future iterations of this work, as it would be unfair at this point of the Java-Sucuri implementation to compare to its Python-based Sucuri implementation, as the first only supports shared-memory systems so far.

References

Ahn, S., Gorlatova, M. and Chiang, M. (2017) 'Leveraging fog and cloud computing for efficient computational offloading', *Proceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC)*, pp.1–4.

Ai, Y., Peng, M. and Zhang, K. (2017) 'Edge cloud computing technologies for internet of things: a primer', *Digital Communications and Networks*, Vol. 4, No. 2, pp.77–86.

Alves, T., Marzulo, L., Kundu, S. and França, F.M.G. (2018) 'Concurrency analysis in dynamic dataflow graphs', *IEEE Transactions on Emerging Topics in Computing*, pp.1–1.

Alves, T.A., Marzulo, L.A., Franca, F.M. and Costa, V.S. (2011) 'Trebuchet: exploring TLP with dataflow virtualisation', *International Journal of High Performance Systems Architecture*, Vol. 3, Nos. 2/3, pp.137–148.

Alves, T.A.O., Goldstein, B.F., França, F.M.G. and Marzulo, L.A.J. (2014) 'A minimalistic dataflow programming library for python', *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshop*, pp.96–101.

Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H. and Aamodt, T.M. (2009) 'Analyzing cuda workloads using a detailed gpu simulator', *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, pp.163–174.

Balaji, P. (2015) *Intel Threading Building Blocks*. MIT Press

Bienia, C. (2011) *Benchmarking Modern Multiprocessors*, PhD Thesis, Princeton University.

Bosboom, J., Rajadurai, S., Wong, W-F. and Amarasinghe, S. (2014) 'StreamJIT: a commensal compiler for high-performance stream programming', *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ACM, Vol. 49, pp.177–195.

Bosilca, G., Bouteiller, A., Danalis, A., Hérault, T., Lemarinier, P., and Dongarra. J. (2012) 'Dague: a generic distributed dag engine for high performance computing', *Parallel Computing*, Vol. 38, Nos. 1/2, pp.37–51.

Bucek, J., Lange, K-D. and v. Kistowski, J. (2018) 'Spec epu2017: next-generation compute benchmark', *Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering* (ICPE'18), ACM. New York, NY, USA, pp.41–42.

Carvalho, C.B.G., Ferreira, V.C., França, F.M.G., Bentes, C., Alves, T.A.O., Sena, A.C. and Marzulo, L.A.J. (2017) 'Towards a dataflow runtime environment for edge, fog and in-situ computing', *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp.115–120.

Cerina, L., Notargiacomo, S., Paccanit, M.G. and Santambrogio, M.D. (2017) 'A fog-computing architecture for preventive healthcare and assisted living in smart ambients', *Proceedings of the IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*, pp.1–6.

Chafik, O. (2015) *Javacl: Opencl bindings for java*. Available online at: `https://github. com/nativelibs4java/JavaCL` (accessed on 9 September 2017).

Consortium (2017) *Openfog reference architecture for fog computing*. Available online at: `www.OpenFogConsortium.org`

Cuda c Programming Guide (n.d.) Available online at: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4idxrP1Vq` (accessed on 30 May 2017).

Dennis, J.B. and Misunas, D.B. (1975) 'A preliminary architecture for a basic data-flow processor', *Proceedings of the 2Nd Annual Symposium on Computer Architecture* (ISCA'75), New York, NY, USA, ACM, pp.126–132.

Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X. and Planas, J. (2011) 'Ompss: a proposal for programming heterogeneous multi-core architectures', *Parallel Processing Letters*, Vol. 21, pp.173–193.

Giorgi, R., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., Fechner, B., Gao, G.R., Garbade, A., Gayatri, R., Girbal, S., Goodman, D., Khan, B., Kolia, S., Landwehr, J., Nhat Minh N., Li, F., Luj`an, M., Mendelson, A., Morin, L., Navarro, N., Patejko, T., Pop, A., Trancoso, P., Ungerer, T., Watson, I., Weis, S., Zuckerman, S. and Valero, M. (2014) 'TERAFLUX: harnessing dataflow in next generation teradevices', *Microprocessors and Microsystems*, pp.976–990.

Grasso, I., Radojkovic, P., Rajovic, N., Gelado, I. and Ramirez, A. (2014) 'Energy efficient hpc on embedded socs: optimization techniques for mali gpu', *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, pp.123–132.

Henning, J.L. (2006) 'Spec cpu2006 benchmark descriptions', *SIGARCH Computer Architecture News*, Vol. 34, No. 4, pp.1–17.

Li, J., Jin, J., Yuan, D. and Zhang, H. (2018) 'Virtual fog: a virtualization enabled fog computing framework for internet of things', *IEEE Internet of Things Journal*, Vol. 5, No. 1, pp.121–131.

Liu, L., Chang, Z., Guo, X., Mao, S. and Ristaniemi. T. (2018) 'Multiobjective optimization for computation offloading in fog computing', *IEEE Internet of Things Journal*, Vol. 5, No. 1, pp. 283–294.

Marzulo, L. A., Alves, T.A., França, F.M. and Costa, V.S. (2014) 'Couillard: parallel programming via coarse-grained data-flow compilation', *Parallel Computing*, Vol. 40, No. 10, pp.661–680.

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

Moore, G.E. (2000) *Readings in computer architecture*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter Cramming More Components Onto Integrated Circuits, pp.56–59.

Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K. and Chang, K. (1996) 'The case for a single-chip multiprocessor', *IEEE Computer*, pp.2–11.

Peña, A.J., Reaño, C., Silla, F., Mayo, R., Quintana-Ortí, E.S. and Duato, J. (2014) 'A complete and efficient cuda-sharing solution for hpc clusters', *Parallel Computing*, Vol. 40, No. 10, pp.574–588.

Rocha, M.P., França, F.M.G., Nery, A.S. and Guedes. L.S. (2017) 'Dataflow programming for stream processing', *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp.103–108.

Silva, R.J.N., Goldstein, B., Santiago, L., Sena, A.C., Marzulo, L.A.J., Alves, T.A.O. and França, F.M.G. (2016) 'Task scheduling in sucuri dataflow library', *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp.37–42.

Tan, T.H., Ooi, C.Y. and Marsono, M.N. (2017) 'hpfog: a fpga-based fog computing platform', *Proceedings of the International Conference on Networking, Architecture, and Storage (NAS)*, pp.1–2.

TIOBE *Software* (2018) *TIOBE programming community index*. Available online at: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (accessed on 24 July 2018).

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

Uliana, D., Kepa, K. and Athanas, P. (2013) 'Fpga-based hpc application design for non-experts', *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, pp.9–15.

Wilde, M. Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S. and Foster, I. (2011) 'Swift: a language for distributed parallel scripting', *Parallel Computing* (Emerging Programming Paradigms for Large-Scale Scientific Computing), Vol. 37, No. 9, pp.633–652.

Wozniak, J.M., Armstrong, T.G., Wilde, M., Katz, D.S., Lusk, E. and Foster, I.T. (2013) 'Swift/t: Large-scale application composition via distributed-memory dataflow processing', *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp.95–102.

Yazdanbakhsh, A., Mahajan, D., Lotfi-Kamran, P. and Esmaeilzadeh, H. (2016) *Axbench: A Benchmark Suite f or Approximate Computing Across the System Stack*. Technical Repot. Available online at: http://hdl.handle.net/1853/54485