

HUMAN-CENTERED
COMPUTING

Caravela: A Novel Stream-Based Distributed Computing Environment

Shinichi Yamagiwa and Leonel Sousa
INESC-ID/IST, Technical University of Lisbon

Distributed computing implies sharing computation, data, and network resources around the world. The Caravela environment applies a proposed flow model for stream computing on graphics processing units that encapsulates a program to be executed in local or remote computers and directly collects the data through the memory or the network.

In the past few years, researchers have applied distributed and parallel computing to the simultaneous use of multiple resources. The message passing interface (MPI) usually supports distributed computing, and parallel processing is based on threads for concurrent programming on shared memory multiprocessors.

Modern computers, programming languages, and operating systems support these concurrency models. Universities and laboratories worldwide have built Beowulf cluster computers using commercially available commodity components. Grid computing exploits the anonymous use of idle, personally owned computing resources from around the world to create high-performance distributed systems, much like drawing on electric power from a wall outlet.

Although the CPU plays a core role in general-purpose computers, specialized processors have become increasingly important in these machines. Video encoders and graphics processing units (GPUs) are typical examples of specialized processors for stream-based computing, specifically for multimedia and graphics applications. Stream-based programming gathers data into a stream, operates on it, and then scatters it back to memory. Stream computing can also be mapped onto general-purpose CPUs by decoupling computation and memory accesses, boosting memory reads before the computation, and postponing writes of the live data to memory after the computation.¹

Researchers have developed high-level languages and interfaces, such as Cg and Brook for GPUs, to facilitate stream-processor programming. This work has generally been limited to computing small-scale problems locally on a single-stream processor. An exception is a limited set of experiments involving an MPI-supported cluster of GPUs that directly applies stream computing to the distributed programming model.² However, this type of distributed programming model is neither simple to implement nor efficient when applied to the multiple specialized processors available on a network of computers.

A proposed new *flow model* considers the more important aspects of distributed computing—including process migration, data transfer, compatibility, heterogeneity, and security—as well as the main characteristics of stream-based computing. This model makes it possible to easily share programs among computers connected to a network as well as executing them independently of their physical locations, with data flowing from and into the network through memory. The model assigns the same functionality to the network interface and to the local memory.

Caravela (www.caravela-gpu.org) is a novel distributed environment for applying the flow model on GPUs. Named after the speedy vessel commanded by Pedro Álvares Cabral, the Portuguese explorer who discovered Brazil in 1500, Caravela provides a unique toolset for programming and managing I/O data in local and remote machines. In Caravela, programmers can easily generate

and share flow-model units, each corresponding to an implementation of the model for different applications, that any system can execute locally or remotely. The environment requires only the processing resources needed to execute the flow-model unit and can be used for general-purpose computation, including numeric computing operations, physical simulations, and data mining.

To illustrate the proposed model and toolset, we use as an example an MPEG-2 video stream that can be sent together with its decoder program to a remote computer's GPU, which executes the decoder to display the frames directly on the screen. The flow model also can be used to distribute processing on large-scale computations. Toward that end, we use Caravela's tools to configure the specialized processors on different machines as a pipeline supported on local area networks (LANs) and wide area networks (WANs).

FLOW MODEL

A data stream is a sequence of data items $\alpha_0, \alpha_1, \dots, \alpha_n$ such that, on each pass through the stream, a processing unit reads the items once in an increasing order of their indices. Stream computing corresponds to applying a kernel (π) comprising several operations in sequence (usually several hundreds of them) to the single input data stream α to produce a single output data stream $\psi = \pi(\alpha)$. In general, stream computing can be applied to a set of input streams to produce a single output stream $\psi = \pi^{(i)}(\alpha^{(i)})$ or even to compute in parallel a set of output streams $\psi^{(i)} = \pi^{(i)}(\alpha^{(i)})$. This simple model does not support recursive computation because it prohibits interchanging the role of input and output streams.

The proposed flow model is generic in that it can be used for any type of stream computing. Input data items can result from previous computation (recursive computation), and the model can compute several output streams by applying a set of kernels in parallel to multiple input data streams.

Figure 1 shows the flow model's conceptual structure, which consists of several data I/O streams, input constant values, and a kernel based on a predefined instruction set. Input constant values are useful for parameterizing kernel processing. The flow model also supports recursive computation by introducing the *memory effect*—having the kernel store a few results used to generate an additional input data stream or outside the kernel.

Flow-model-based computation can map I/O streams into memory or even directly into network interfaces. The kernel only reads the input data stream and provides output data—the only computing resources it touches are the I/O streams and the constant values.

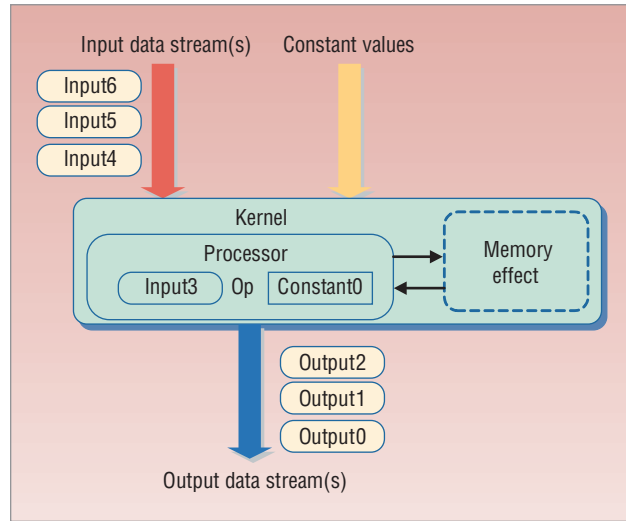


Figure 1. Flow model conceptual organization. The model's structure consists of several data I/O streams, input constant values, and a kernel based on a predefined instruction set.

This is an important property of the proposed model, as it intrinsically protects machines that locally or remotely perform flow-model-based execution.

The simple kernel calculation in Figure 2 illustrates the potential and merits of flow-model-based programming. The kernel corresponds to a finite impulse response (FIR) filter with four taps, where b_0 through b_3 are the constant coefficients and x_n through x_{n-3} represent the input samples, corresponding to Input0 to Input3 in the model, and y denotes the output filtered stream. The kernel computation involves multiplying four samples by the corresponding b values and accumulating the result in tmp_x . The tmp_x output corresponds to the y stream generated by the multiplication and addition.

The example in Figure 2 assumes a small number of filter taps. However, to accommodate an infinite number of b values in the impulsive response, the flow model should have other shapes. It can have one or more input

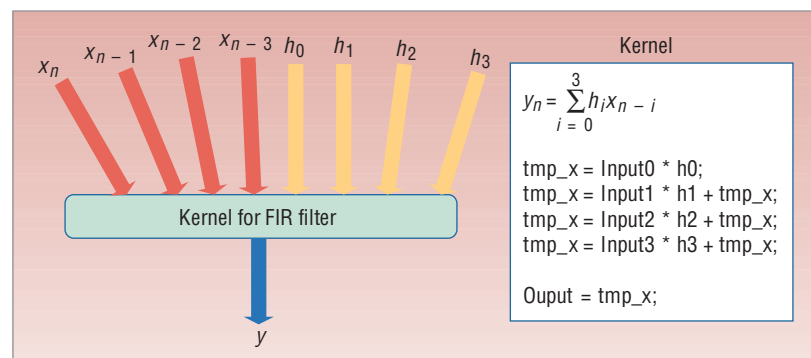


Figure 2. Example of flow-model-based computation. The kernel corresponds to a finite impulse response (FIR) filter with four taps, where h_0 through h_3 are the constant coefficients and x_n through x_{n-3} represent the input samples, corresponding to Input0 to Input3 in the model, and y denotes the output filtered stream.

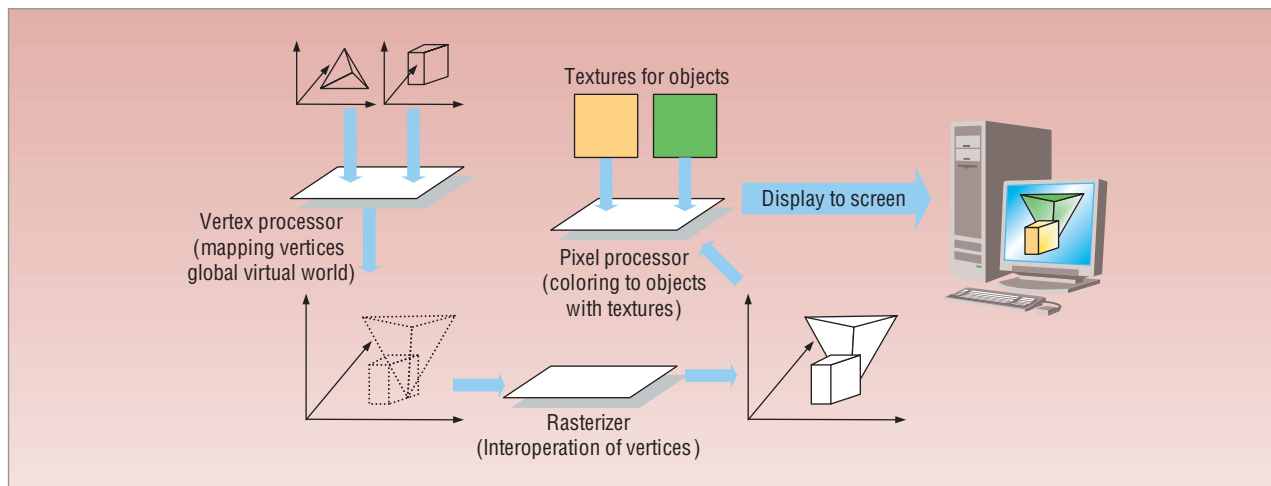


Figure 3. Processing pipeline for GPU graphics. With data generated from a rasterizer, a pixel processor creates the pixel color data streams for each point on the screen by performing texture mapping with textures for the 3D objects. The processor concurrently computes the pixel color components and sends them to the screen. Texture inputs and color data constitute I/O streams directly accessible from the GPU's external memory.

data streams to implement feedback by generating an input data stream from a delayed output data stream y . In this case, a flow-model unit can be viewed simply as a temporal iteration of the flow model using external memory. Thus, it easily accommodates recursive computation. Although implementation of recursive flow-model units depends on the availability of the memory effect, it need only access input and output data streams without having to touch any other resources except the kernel hardware.

APPLYING THE FLOW MODEL TO GPUS

In modern computers, GPUs are the main processing resources along with the CPUs. These GPUs are implemented as programmable units equipped with very high-bandwidth memory and integer/floating-point hardware units to operate on data streams, each consisting of an ordered sequence of primitives attributed to graphics applications. GPU performance for graphics applications has accelerated faster than Moore's law as it applies to CPUs, on average more than double per year. For example, the Nvidia GeForce 6800 Ultra GPU has a peak performance of 40 Gflops and a memory bandwidth of 35.2 Gbytes per second, compared with 6.8 Gflops and 6 Gbytes per second for a 3-GHz Pentium 4 CPU.

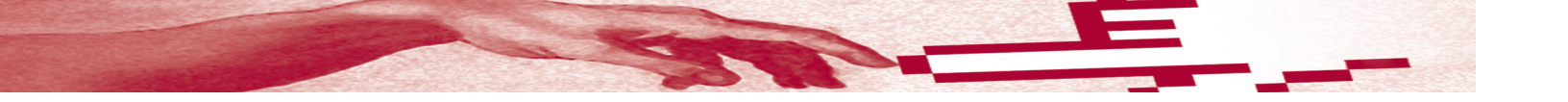
In the past few years, the development of new algorithms has enabled nongraphics applications to use GPU processing power. These GPU-based algorithms exploit the GPU's single-instruction, multiple-data (SIMD) capabilities along with its vector-processing functionalities to efficiently perform computations. GPU-based algorithms have been developed for several applications, such as database queries and data mining, numerical and scientific computation, sorting, motion estimation and planning, bioinformatics, and simulation of fluid flows.³

The General-Purpose Computation Using Graphics Hardware Web site (www.gpgpu.org) discloses information about types of software particularly specialized for molecular dynamics calculation on the GPU (<http://folding.stanford.edu>) and tools that simplify GPU programming for general-purpose uses, such as Brook for GPUs and the Sh language.

In most cases, GPUs are implemented in two programmable cores: vertex and pixel processors. Small programs developed for these processors are usually known as shaders.

As Figure 3 shows, when using a vertex processor to remap the coordinates of each modeled 3D object, designers have no access to data directly transferred to the rasterizer, preventing use of the vertex processor for general-purpose computing. With data generated from a rasterizer, a pixel processor creates the pixel color data streams for each point on the screen by performing texture mapping with textures for the 3D objects. The processor concurrently computes the pixel color components—A(lpha element), R(ed), G(reen), and B(lue)—and sends them to the screen. Texture inputs and color data constitute I/O streams directly accessible from the GPU's external memory.

The flow-model concept fits perfectly with the pixel processor's characteristics. The multiple floating-point pipelined units embedded in a pixel processor, typically four or more, provide both the SIMD capabilities and the GPU's vector-processing functionalities. They can process four input streams in parallel to produce parallel output streams. A pixel processor can be programmed in a low-level assembly language such as Direct3D, the High-Level Shader Language (HLSL) from Microsoft (www.microsoft.com/directx), or the OpenGL Shading Language (GLSL; www.opengl.org).



These languages assist in constructing a uniform programming environment and let Caravela take advantage of the hardware—for example by applying the blending-based conditional assignments to overcome the poor performance of branching instructions in the current programmable GPU pipeline.

CARAVELA

The Caravela environment consists of three main components. The *library* provides all the functions required to develop and execute an application based on the flow model in the Caravela environment. It is provided as a mass of C-based interface functions that find available sites as well as each implement the distributed flow-model management and a remote executor.

The *distributed flow-model manager* lets all the computers in the Caravela environment share a flow-model unit at any site. This manager provides the functionality to describe an atomic and independent flow-model unit in a file and also restores it from a file. The restoring function is embedded in the Caravela library.

The *flow-model executor* is a server placed in a contributing machine somewhere in the world to accept execution requests of flow models from applications and to distribute and execute a flow-model unit in a distributed environment. The library includes functions to send requests to a server and to execute a flow-model unit.

A library implementation for GPUs

The Caravela environment supported on GPUs assumes that single or multiple pixel processors are connected to the host machine. Caravela defines three different layers to specify a processing unit: The *machine* is a computer with CPU(s), memory, and hard disk(s); the *video adapter* is connected to this machine; and the *shader* is a pixel processor embedded in an adapter.

Although a conventional video adapter actually includes only a pixel processor, Caravela already supports multiple pixel processors, which is expected to be the architecture of next-generation GPUs. The environment uses a set of library functions in sequence to map flow models to shaders. These Caravela functions allow introduction of a machine in the environment, identification of a shader, generation of a flow-model unit, and mapping and firing execution of this flow-model unit in a shader.

Thus, a machine must be explicitly introduced in the Caravela environment that automatically identifies the available processors. This operation is fundamental to set up the environment, but it is independent of the generation of flow-model units that run on any GPU's pixel processor. The machine can be either local or remote. In particular, local introduction means that the flow-model

units are mapped and executed on that specific machine. Remote machines are placed in remote computers connected by the network and located anywhere around the world. After introduction of a machine, the Caravela library provides the function to find pixel processors (shaders) available in the machine.

A flow-model unit can be generated either directly through a function the library provides or by restoring it from a file written in XML. The former calls the function with arguments to dynamically create a flow-model unit directly in an application program. The main disadvantage of this approach is that the flow-model unit cannot be shared with other application programs and

machines because only an application program and its machine maintain the original code that generates the flow-model unit. If the flow-model unit is restored from an XML file, the unit can be shared among all application programs and machines. The XML file corresponding to a flow-model unit includes the number

of input streams, the number of output streams, the amount of data in I/O streams, constant values, and a program.

The Caravela library's shader-mapping function maps a flow-model unit to a pixel processor, which is required before firing its execution. The mapping function compares the flow-model unit's conditions with the corresponding processor's characteristics in terms of the maximum number of I/O streams, maximum data size, and supported data types. If they match, the mapping function returns a *fuse*, the data structure necessary to trigger the flow-model unit execution. As soon as the fuse is passed to the *fire* function, this unit starts to execute.

Caravela's current version supports the Windows environment with DirectX9. Developers currently can write a program in the DirectX assembly language or HLSL; the next version of Caravela will also support OpenGL and GLSL. When an application maps a flow model to a processor, the mapping function automatically compiles the corresponding program. The program must specify a version of a target pixel processor. Version mismatch incompatibility can cause a compilation error. In a failure, the mapping function returns an error code.

Distributed flow-model management

Caravela's applications can share flow-model units placed anywhere through XML files. As Figure 4 shows, FlowModelCreator provides a GUI for interactively generating a flow-model unit. The application packs all of a flow-model unit's information in an XML file. The XML file is readable from the Caravela function that restores the flow-model unit from a file. When a URL is passed to the function as the file name, the function contacts the

Caravela's applications can share flow-model units placed anywhere through XML files.

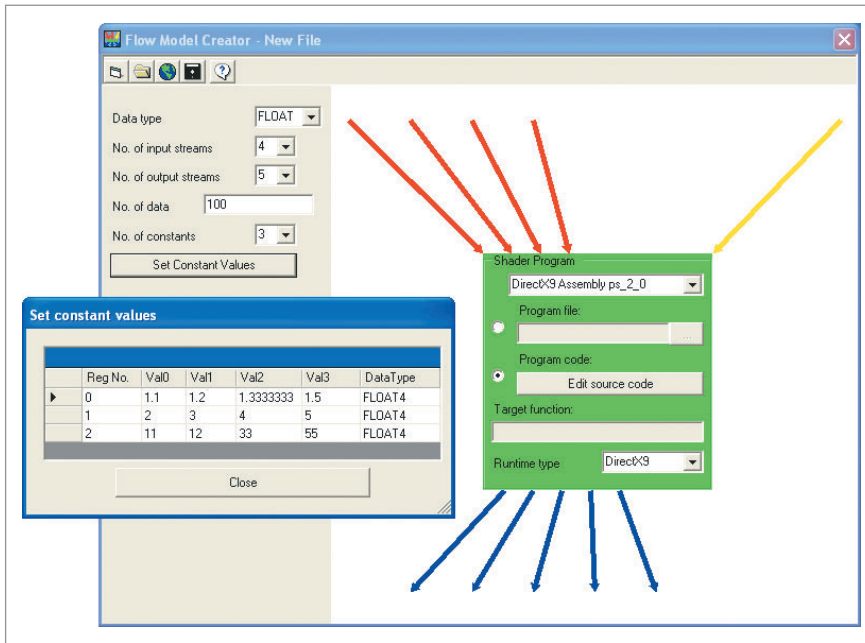


Figure 4. FlowModelCreator application. This screen shot shows the GUI for generating a flow model with four input data streams, five output data streams, and three constant values.

URL through HTTP and restores the flow-model unit from the configuration described in the XML file.

With this mechanism, application designers can use all flow-model units already developed and located in any machines integrated in the Caravela environment. The most commonly used stream-based algorithms include signal processing, video processing, and data compression. They identify the URL where the XML file with the corresponding flow-model unit can be found. This mechanism eases the implementation of a service to find necessary flow-model units accessible via HTTP by typing keywords used on a flow-model unit's database. Occasionally, this database can return to the same application different flow-model units that satisfy various user requirements such as data accuracy and processing time. With the flow-model database, users can look for a suitable flow-model unit with a uniform interface through a URL.

Remote flow-model unit execution

Remote machines are categorized in two types. A *worker* machine is a remote device that provides its resources to a given application and fires the corresponding flow-model units to execute the kernels. A *broker* machine is a remote device that is responsible for routing an application to the appropriate worker machine.

Figure 5 shows the relation between worker and broker machines. While the worker machines constitute a virtual tree connection, the broker machine operates as a node at an intermediate level of the tree. A broker

machine directly connected to a worker machine is a *parent broker* of the worker machine. A worker and its parent broker can specify a different parent broker. By tracing these parent brokers from an application, a request client will reach its destination worker machine.

In Figure 5, Broker0 has the routing information for Broker1 and Broker2. Broker1 has two workers, Worker0 and Worker1. Here, Broker2 is a gateway machine. Worker2 and Worker3 are connected to Broker2. Therefore, Broker2 provides mechanisms to go over the firewall. In this example, Broker0 can collect all the information for Worker0, 1, 2, and 3 in its routing tables and thus make available those workers' GPUs.

Worker, broker, and application machines exchange requests or replies using the Simple Object Access Protocol via Web services provided by each machine in the Caravela environment. While an application machine always works as a Web services client, worker and broker machines operate not only as servers but also as clients, because they must forward requests to their parent brokers. The Web services accept a request, serve on it, and return a reply. When an application machine must send a request to a worker or broker machine, it only needs to write an XML file as the request to the server machines and to return a request ID to the application. The corresponding reply simply returns XML file content associated with the request ID.

When a worker or broker machine receives a request, Caravela activates the CaravelaSnoopServer program. The server snoops the request directory, processes new requests, and writes corresponding reply messages into the reply directory. In the server setting, a Web service's URL specifies the parent broker and local service. CaravelaSnoopServer reads the URL from the "location" attribute of each Web service's WSDL file, so processing resource contributors need only insert their WSDL file in a given URL. Thereafter, other worker or broker machines can access their desired services' URLs as their parent broker's places or as their local places.

The request/reply mechanism propagates requests through a firewall to its LAN if one of the parent brokers is placed between the WAN and a LAN. Indeed, Caravela's application can use all contributed computing resources without considering TCP port restrictions. Moreover, the access privilege of the Web services of worker and broker

machines belongs to the settings of their HTTP servers. Contributors thus can control external accesses before CaravelaSnoopServer processes requests. This mechanism provides flexibility for security settings. If desired, the contributors can accept any anonymous requests from around the world.

EXAMPLE FLOW-MODEL EXECUTION

The Caravela library functions support all the required steps for executing a flow-model unit based on the flow model in Figure 2. Figure 6a shows one possible sequence of steps using a local machine, worker machine, and worker via a broker machine.

In the first step, the CreateMachine function creates a machine that can be a local machine, a worker machine, or a worker machine via a broker. For remote worker creation, Caravela passes a URL specified in the Web services to the function. When an application requests a remote worker machine via a broker, the worker machine information must be returned to the caller via the broker. Caravela thus calls the GetRemoteMachines function to acquire remote worker machines belonging to the broker. Because this mechanism lets the broker machine provide all the worker machines' URLs, the programmer does not need to know those URLs a priori.

The next step in Figure 6a is creation of a flow-model unit, which can be done from scratch or by restoring one from an XML file. The first method creates a flow model using the CreateFlowModel function in memory, sets constant values for the model using the SetConstantsToFlowModel function, and stores the corresponding shader program code in the model. The second method simply uses the CreateFlowModelFromFile function to read the URL pointing to the flow model's XML file described in FLOWMODEL_FILE.

The program shown in Figure 6b is written in assembly language according to ShaderModel 2.0 of DirectX. This shader assembly language assigns constants b_0 through b_3 to the corresponding constant registers c_0 through c_3 . Expecting the input data streams to be provided as textures, the code uses the texld instruction to load them into r_0 through r_3 . Because each input stream sample includes four data elements, register r_4 (tmp_x in Figure 2b) accumulates four results in parallel due to the

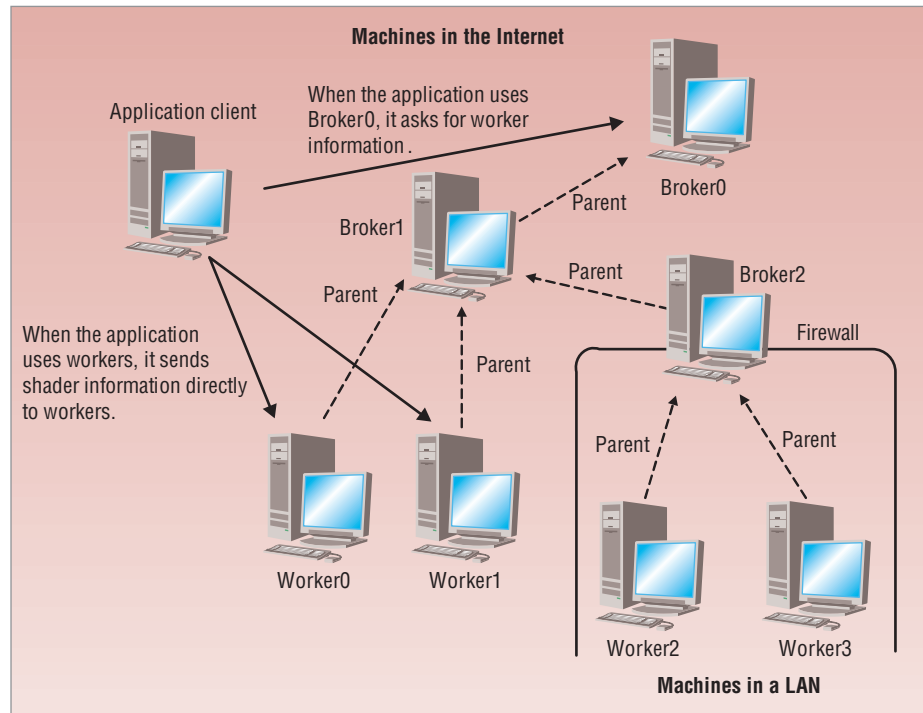


Figure 5. Communication in the Caravela environment. Workers belong to their parent broker, and each broker can have its parent broker.

GPU architecture's characteristics. The register oC_0 is the program's—and thus the flow model's—output stream.

After creating the flow model, the program tries to find processors available in the worker machine by calling the QueryShader function. The program maps the flow-model unit to a processor through the MapFlowModelIntoShader function. The GetInputData function then gets memory regions for input streams and initializes the input values.

Finally, the FireFlowModel function executes the flow model. After the execution, the GetOutputData returns the output data stream y . During an execution, the Caravela library has a single interface for executing a flow model—independently of being executed locally, directly on a remote worker, or on a remote worker via a broker. Moreover, a flow model can be created separately from its execution steps in the code. Flow-model units can be shared via network by specifying XML files through HTTP. Thus, the Caravela environment accesses all the GPU resources and lets designers share all the GPU algorithms distributed globally.

The Caravela package includes sample kernels for 1D and 2D FIR filtering. The Caravela team is programming several other applications and will be deploying the corresponding flow-model units at its Web site. For example, it takes about 10.5 seconds to process an application program of a 3×3 FIR filter to an image with $1,024 \times 1,024$ pixels and 100 iterations in an Nvidia GeForce 7300GS graphics card, connected to the PC through a PCI Express bus.

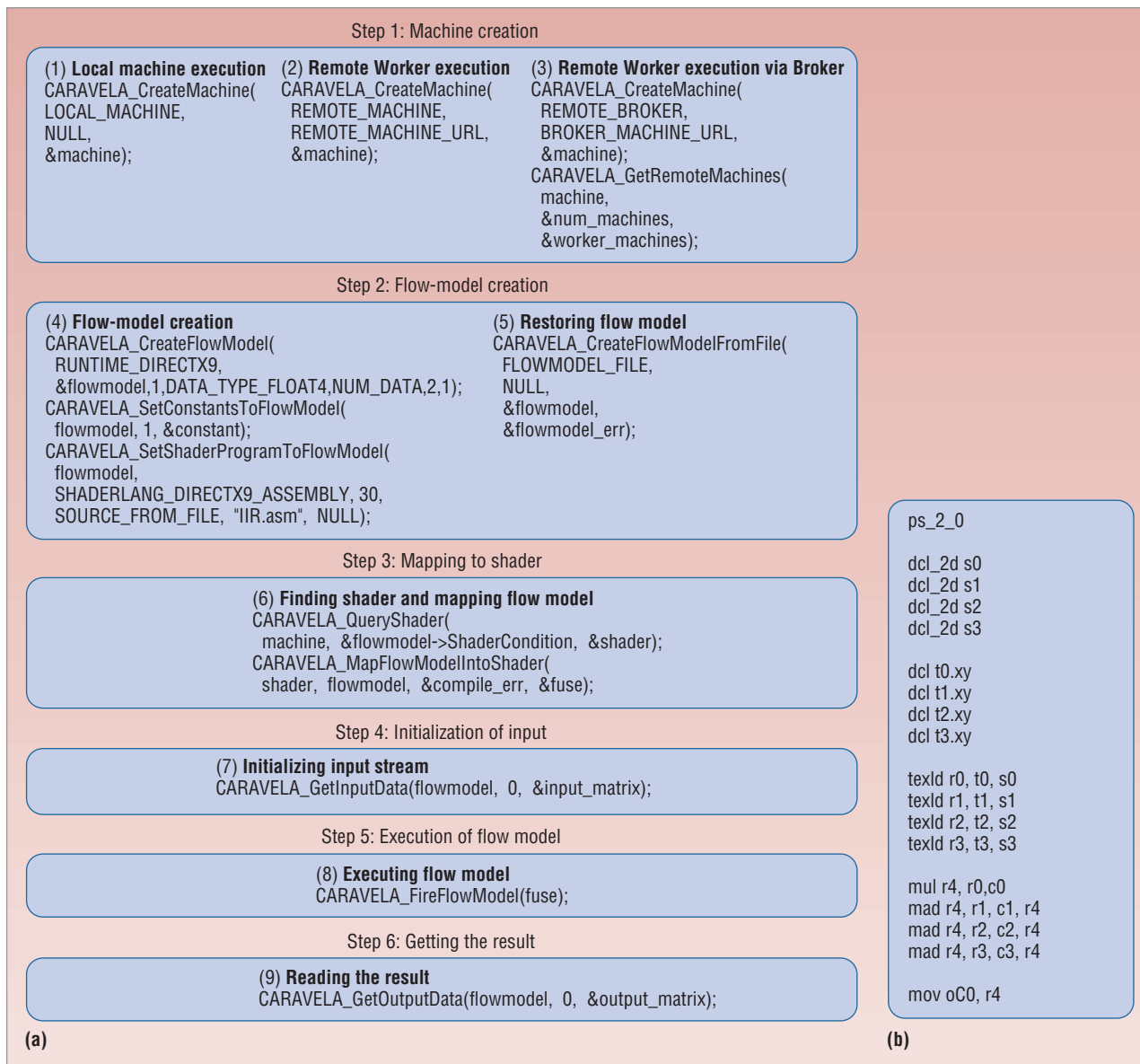
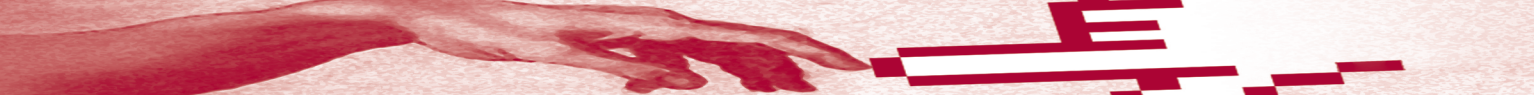


Figure 6. Flow-model execution. (a) Program flow using the Caravela library; (b) a shader stored in the FIR filter's flow-model unit.

Data transfer between the host memory and the video memory (VRAM) is currently one of the bottlenecks of GPU-based applications because data transfer occupies approximately 70 percent of the total execution time. This is also an issue to be considered in recursive computing, as such applications may have to use the host memory to place output data into the GPU's input.⁴ In the host machine, an AMD Opteron running at 2 GHz and with 2 Gbytes of RAM, the execution time is approximately 23.5 seconds, 2.2 times slower than in a GPU.

Although Caravela currently supports remote execution based on individual flow models, we are extending the environment to implement distributed computing based on *meta-pipelining*.

Caravela will organize multiple processors on different machines in a pipeline over a network, where all participating flow-model units will run in a pipeline fashion. In this meta-pipelining, each flow-model unit resides in a different virtual pipeline stage and passes data streams from one stage to the next stage as it processes the data.

We anticipate some interesting applications for pipeline-model-based processing. For example, Caravela can process video encoding by assigning a flow-model unit to each step of the video encoding scheme and having those flow-model units process the sequence of input frames in a pipeline manner. The Caravela environment makes it possible to create and manage a virtual computational meta-pipeline for such applications formed by workers around the world. ■

Acknowledgments

This work was partially supported by the Portuguese Foundation for Science and Technology through the FEDER program. We thank Munehiro Fukuda at the University of Washington, Bothell, and the anonymous referees for their helpful comments.

References

1. J. Gummaraju and M. Rosenblum, "Stream Programming on General-Purpose Processors," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, 2005, pp. 343-354.
2. Z. Fan et al., "GPU Cluster for High-Performance Computing," *Proc. 2004 ACM/IEEE Conf. Supercomputing*, IEEE CS Press, 2004, pp. 47-58.
3. J.D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," *Eurographics 2005, State of the Art Reports*, Aug. 2005, pp. 21-51.
4. S. Yamagiwa, L. Sousa, and D. Antão, "Data Buffering Optimization Methods Toward a Uniform Programming Interface for GPU-Based Applications," to appear in *Proc. 2007 ACM Int'l Conf. Computing Frontiers*, ACM Press, 2007.

Shinichi Yamagiwa is a researcher at Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento (INESC-ID), Technical University of Lisbon. His research interests include parallel and distributed computing, especially using GPU resources, and network hardware and software for cluster computers. Yamagiwa received a PhD in engineering from the University of Tsukuba, Japan. He is a member of the IEEE. Contact him at yama@inesc-id.pt.

Leonel Sousa is an associate professor in the Department of Electrical and Computer Engineering at Instituto Superior Técnico and a researcher at INESC-ID, Technical University of Lisbon. His research interests include computer architectures, high-performance computing, embedded systems, and multimedia signal processing. Sousa received a PhD in electrical and computer engineering from the Technical University of Lisbon. He is a senior member of the IEEE and a member of the ACM. Contact him at las@inesc-id.pt.

ADVERTISER INDEX MAY 2006

Advertiser	Page Number	Advertising Personnel	
Compsac 2007	Cover 3	Marion Delaney	Sandy Brown
CSDP-Training	7	IEEE Media, Advertising Director	IEEE Computer Society, Business Development Manager
e-Science and Grid Computing Conference 2007	69	Phone: +1 415 863 4717	Phone: +1 714 821 8380
Fairfield University	79	Email: md.ieeemedi@ieee.org	Fax: +1 714 821 4010
IEEE Computer Society Awards	13	Marian Anderson	Email: sb.ieeemedi@ieee.org
IEEE Computer Society Membership	82-84	Advertising Coordinator	
IEEE Member Digital Library	81	Phone: +1 714 821 8380	
IPDPS 2007	Cover 2	Fax: +1 714 821 4010	
John Wiley & Sons, Inc.	5	Email: manderson@computer.org	
Seapine Software, Inc.	Cover 4		
WorldSciNet	22		
Classified Advertising	78-80		
Advertising Sales Representatives			
Mid Atlantic (product/recruitment)	Midwest (product)	Midwest/Southwest (recruitment)	Northwest/Southern CA (recruitment)
Dawn Becker	Dave Jones	Darcy Giovingo	Tim Matteson
Phone: +1 732 772 0160	Phone: +1 708 442 5633	Phone: +1 847 498-4520	Phone: +1 310 836 4064
Fax: +1 732 772 0161	Fax: +1 708 442 7620	Fax: +1 847 498-5911	Fax: +1 310 836 4067
Email: db.ieeemedi@ieee.org	Email: dj.ieeemedi@ieee.org	Email: dg.ieeemedi@ieee.org	Email: tm.ieeemedi@ieee.org
New England (product)	Will Hamilton	Southwest (product)	Southeast (product)
Jody Estabrook	Phone: +1 269 381 2156	Steve Loerch	Bill Holland
Phone: +1 978 244 0192	Fax: +1 269 381 2556	Phone: +1 847 498 4520	Phone: +1 770 435 6549
Fax: +1 978 244 0103	Email: wh.ieeemedi@ieee.org	Fax: +1 847 498 5911	Fax: +1 770 435 0243
Email: je.ieeemedi@ieee.org	Joe DiNardo	Email: steve@didierandbroderick.com	Email: hollandwfh@yahoo.com
New England (recruitment)	Phone: +1 440 248 2456	Northwest (product)	Japan
John Restchack	Fax: +1 440 248 2594	Peter D. Scott	Tim Matteson
Phone: +1 212 419 7578	Email: jd.ieeemedi@ieee.org	Phone: +1 415 421-7950	Phone: +1 310 836 4064
Fax: +1 212 419 7589	Southeast (recruitment)	Fax: +1 415 398-4156	Fax: +1 310 836 4067
Email: j.restchack@ieee.org	Thomas M. Flynn	Email: peterd@pscottassoc.com	Email: tm.ieeemedi@ieee.org
Connecticut (product)	Phone: +1 770 645 2944	Southern CA (product)	Europe (product/recruitment)
Stan Greenfield	Fax: +1 770 993 4423	Marshall Rubin	Hilary Turnbull
Phone: +1 203 938 2418	Email: flynttom@mindspring.com	Phone: +1 818 888 2407	Phone: +44 1875 825700
Fax: +1 203 938 3211		Fax: +1 818 888 4907	Fax: +44 1875 825701
Email: greenco@optonline.net		Email: mr.ieeemedi@ieee.org	Email: impress@impressmedia.com