SysCellC: a data-flow programming model on multi-GPU

Dominique Houzet, Sylvain Huet, Anis Rahman

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2009) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

SysCellC: a data-flow programming model on multi-GPU

Dominique Houzet, Sylvain Huet, Anis Rahman[1]

GIPSA-Lab CNRS, 961 rue de la Houille Blanche, 38402 St Martin d'Heres, France

Abstract

High performance computing with low cost machines becomes a reality with GPU. Unfortunately, high performances are achieved when the programmer exploits the architectural specificities of the GPU processors: he has to focus on inter-GPU communications, task allocations among the GPUs, task scheduling, external memory prefetching, and synchronization. In this paper, we propose and evaluate a compile flow. It automates the transformation of a program expressed with the high level system design language SystemC, to its implementation on a cluster of multi-GPU. SystemC constructs and scheduler are directly mapped to the GPU API, preserving their semantic. Inter-GPU communications are abstracted by means of SystemC channels.

Keywords: Programming Model; GPU; SystemC;

1. Introduction

High performance computing with low cost machine becomes a reality. In Q4 of 2006, Nvidia launched the G80, the first general purpose GPU. However, in order to obtain the highest performances on multi-GPUs, the programmer has to write programs that best exploit the hardware architecture. Nevertheless, this work can be simplified by using programming models hiding hardware architectural details such as inter-GPU communications, task scheduling and synchronization.

Also, explicit parallel programming models are used to describe parallel algorithms, as efficiency can be achieved only with a careful study of the parallel properties of the algorithm and the mapping on a parallel architecture. Message passing programming model is well adopted for applications exhibiting dataflow parallelism with or without pipelining (streaming) at coarse grain level. The parallelism grain targeted is a main point to study when mapping a parallel algorithm to a parallel architecture. The field of signal and image processing we are targeting corresponds to dataflow programming where an application can be divided in several steps exchanging data at a coarse grain level. This coarse grain parallel programming model is well adapted to clusters of processors as well as multicore processors allowing DMA communication between cores, like the IBM Cell processor [1] or even NoC based MPSoC on ASIC or FPGA.

For lower grain level parallelism we need a different programming model. This is the case for GPU based on data parallelism with shared memory using OpenCL language [2]. In the case of signal and image processing, one step of an application often exhibits data parallelism. The merging of parallel programming models is possible and welcome

[1] Corresponding author. Tel.: +33-476-574-361;
E-mail address: dominique.houzet@grenoble-inp.fr

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

in order to deal and benefit all the parallel capability of the applications through an exploration and optimization of the parallel aspects of an application. The aim is to better understand the parallel properties like scalability, pipelining, parallel granularity, compute/memory ratio, and so on. All those properties are very important to explore in order to obtain an efficient parallel implementation.

What we need at a coarse grain level is a programming language abstracting the communication, synchronization and scheduling of tasks. There is a compromise between abstraction and optimization. Abstraction helps to deal with true issues of parallelism by hiding tedious implementation details like dual or multi-buffering for DMA communications and memory management and synchronization. SystemC[3] is a good example of programming environment providing such abstraction.

In this paper we propose a design flow that automates the transformation of an application specification expressed with SystemC [3] to its multi-GPU implementation. This paper is organized as follows. The second section presents the context of this work: GPU architecture, programming model and SystemC. The third section explains the proposed design flow, from the SystemC specification to its implementation on the cluster of multi-GPU. The fourth section details the C code generation from SystemC. The last section illustrates the approach on three case studies: a simple producer consumer case, a Code Division Multiple Access (CDMA) software radio communication system on CPU, and a visual attention model on multi-GPU.

2. Context

2. 1. GPU architecture and programming

The newer graphics cards implement massively parallel architecture with unified shader model comprising several hundreds of scalar processors running at 1.35 GHz. It achieves the maximum utilization of the hardware computing units by launching and executing massive number of threads. A single instruction is executed across all the processors in a group that is associated to specialized hardware for texture filtering, texture addressing, cache units and fast on-chip shared memory. All these grouped processors can communicate using the shared memory space. The new design delivers impressive computational power, which is made possible by the management of numerous threads on fly along with high memory bandwidth.

OpenCL[2] is a programming model for general purpose computations on graphics hardware and multicores; providing direct control and access over graphics hardware. It is an extension to C language allowing to write programs using standard syntax, markers, and minimum set of extensions to the C language. The new model provides a way around the overhead and redundancy incurred due to the traditional graphics pipeline. The main idea is to launch numerous number of threads to use all the execution units to demonstrate the raw computational power of the graphics device. It unveils some hardware features previously not available in the graphics pipeline, for example threads can access to a common shared memory; providing very high bandwidth communication among threads. It provides more efficient interactions and data transfers between system and device memory. The linear memory addressing scheme is more effective; based on general load-store architecture, that is, allowing random memory access (both gather and scatter). Also the process of transformation of algorithm is eased by the use of GPU-specialized libraries.

2.2. Programming models

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

Programming a parallel machine is more complex than programming a single sequential processor. The programmer has to deal with interactions between the processors, e.g. communications, synchronization, task scheduling. These problems have been addressed since a long time in the parallel programming field [16].

The main programming paradigms are message passing and shared memory multiprocessing. These programming models are widely used through the Message Passing Interface standard (MPI) [10] for message passing and OpenMP [17] for shared memory. Although they are often used to program clusters of computers, their use can be extended to multicore systems like the Cell processor and multi-GPU. Whatever the targeted architecture and the programming model, overlapping communication and computation remains the Holy Grail of parallel computing. The MPI standard is crafted with such overlapping in mind. Nevertheless, due to a number of reasons it is often not accomplished on commodity computing clusters.

A number of programming models/environments have emerged for programming the multicore processors. It includes shared-memory, distributed memory and stream processing models used to implement both data-parallel approaches and task-parallel approaches [3][6-9]. The streaming model [9] is especially interesting. This paradigm relies on two concepts, (1) streams which transport data on which (2) computation kernels work. It is a very simple and thus attractive programming model for an application designer which can be efficiently implemented on multi-GPU.

In this paper we propose an approach based on the SystemC language and parallel programming model targeting the streaming programming model. This language is widely used by the hardware/software systems designer community and offers it the opportunity to easily target the multi-GPU clusters with a higher level of abstraction.

2.3. SystemC

SystemC [2][4-5] is an open source system design language based on C++. Its development started in 1999 and involved academic researchers and many Electronic Design Automation (EDA) companies. It is widely used to model hardware/software or mixed systems at different levels of abstraction, from the system specification to the gate level. SystemC provides C++ classes allowing to decompose a system hierarchically into computation modules and communication channels. SystemC also supplies a simulation kernel which is in charge of the execution of the model. It handles (1) the scheduling and synchronization between the concurrent processes describing modules and channels and (2) the passing of time. It is a discrete event scheduler which supports the concept of delta cycle. A delta cycle refers to an evaluate update step. During evaluation, the simulation kernel calls all the ready-to-run processes in an indeterminate order of execution. This step is followed by an update stage where all the outputs of the evaluated processes are updated. A called process executes until it either finishes or calls the wait() method.

SystemC supports several programming models, called Model of Computation (MoC) [18]. The relation between SystemC and the streaming model is quite straightforward. A kernel corresponds to a computation module, streams to communication channels and the synchronization between kernels and streams is managed by the SystemC simulation engine. The next section details our compile flow leading from a SystemC program to its multi-GPU implementation.

3. Design flow

Figure 1 describes the proposed top-down methodology from SystemC to the generation of the binary file for both the GPUs and the host CPUs. The following subsections describe each step.

Step 1: The design flow starts with the application code described in SystemC. The method presented in this paper focuses on SystemC models structurally described with sc_module, which models computation processes, i.e. kernels, and SystemC primitive channels, that is sc_signal or sc_fifo which corresponds to streams. We distinguish

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

two types of computation processes, the computation intensive ones mapped on GPUs and the processes dedicated to the monitoring of the application, the communication with the CPU environment (I/O) and the management of the CPU memory including GPU data prefetching. This last kind of processes is mapped on the CPUs.

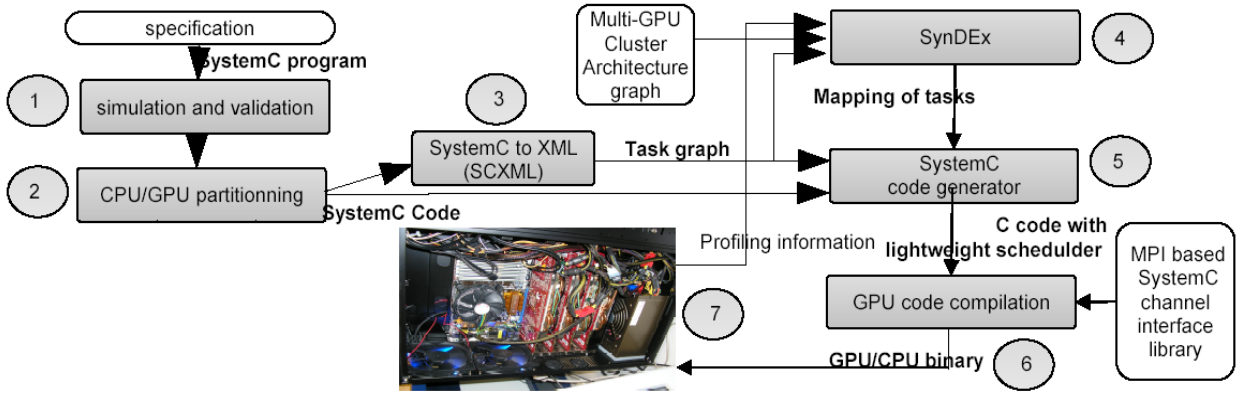

Fig. 1. SysCellC design flow

GPU data prefetching is fundamental: the GPU video memory is usually small in comparison with the amount of data to be processed. Thus the programmer has to size and tile the data in order to optimize the overlapping between CPU/GPU communications and computations. According to the application, data sometimes have to be prefetched from the CPU memory. The SystemC application must exhibit all these prefetching processes. As stated on the preceding item 2 these processes are mapped on the CPUs. As an example, the I/O processes of the streaming programming model are typically such kinds of processes; they usually tile input and output data to feed the GPUs and sometimes do pre or post-processing.

Concerning the GPUs, we impose the following limitations on the SystemC computation processes which can be mapped on GPUs:

- No wait() primitives are allowed and the processes are only sensitive to their sensitivity list.
- the processes are only sensible (through their sensitivity list) to a single signal (event) which can be viewed as a clock. Therefore, a process will only block when it reaches the end of the process.

These restrictions are familiar in the field of synchronous hardware design. They are the expression of the synchronization between concurrent executing components. From the GPU implementation point of view, a global synchronization, i.e. between several GPUs, will occur at the end of a delta cycle, i.e. when all the kernels involved in the synchronization have finished. This means that the GPUs computational load have to be well balanced. This is the role of the Syndex tool used to optimize the mapping of GPU kernels.

Step 2: The second step consists in manually partitioning the SystemC code in two parts. The computation data-parallel part is mapped on the GPUs and the other part on the CPUs. The profiling information can guide the designer's decisions.

Step 3: The step 3 is performed by our SystemC to XML (SCXML) parser tool which converts a given SystemC

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

source code into an XML intermediate representation. The chosen XML format is a subset of the standardized SPIRIT 2.0 format [13]. The application is interpreted as a set of XML files. Each XML file contains the most important characteristics of SystemC components, such as:

- name, type and size of each in/out ports, name and type of processes declared in the constructor and also the sensitivity list of each process.
- name and type of components building a hierarchical tree, the names of connections between the sub-components, and the binding with the component ports.

Considering the CPU side, the SystemC code is parsed to produce a multi-threaded C code managing CPU and GPU memory allocations and GPU memory initialization. The SystemC sc_signal and sc_fifo read and write communication methods are overloaded and implemented with the MPI version 2 (MPI-2) standard, for both intra and inter cluster nodes communication.

Step 4: Both XML files and profiling reports are parsed in order to allocate SystemC components to the different GPUs. We use SynDEx tool [14] to perform an automatic mapping of SystemC components on the different GPUs and CPUs. The different SynDEx inputs are:

- a hierarchical conditioned data-flow graph of computing operations and communication operations. The operations are just specified by the type and size of data and execution time of the components. The XML files and profiling reports are parsed to produce these inputs. The first time the tool is launched, no profiling information is available. Thus all the GPU kernels are mapped on a single GPU to produce profiling timing results at step 7.
- specification of the heterogeneous architecture as a graph composed of processors and communication medias, that is the multi-GPU cluster architecture graph. Processors characteristics are: supported tasks, their execution duration obtained through profiling, worst case transfer duration for each type of data on the interconnect (PCI-Express 2.0 for intra-node communication and Infiniband 40G for inter-node communication) [20] obtained by estimation according to the size of data. The profiling reports and cluster architecture parameters are parsed to produce these inputs.

SynDEx uses a heuristic for the mapping and scheduling of asynchronous tasks, i.e. communicating through sc_fifo, on each processor. After the implementation, a timing diagram gives the mapping of the different tasks on the GPUs and the real time predicted behavior of the system. The communication links are represented in order to show all the exchanges between GPUs and CPUs; they are taken into account in the execution time of tasks.

Step 5: The mapping and scheduling information, the tasks graph and the SystemC code are then used to generate the C code for both the CPUs and GPUs. This code embeds a lightweight SystemC scheduler on the CPUs to preserve the entire operational semantic of the SystemC model. The generated C code is architecture independent so that it can also be fully compiled on a CPU thanks to the overloading of the GPU library [12] and by the implementation of the MPI based SystemC channel interface library. A GPU kernel launcher function is used at SystemC level for GPU data-parallel components. This function get the number of threads and blocks of threads as parameters. There are two implementations, one calling a GPU kernel, and one launching a CPU multi-threaded version. This allows verifying the generated code with classical development environments.

Step 6: The validated C code is compiled with the CPU and GPU compilers to obtain a single multi-threaded binary code for all the CPUs and GPUs. We implemented the SystemC channel interfaces with the MPI standard, between CPUs and between GPUs. The SystemC to XML parser and SystemC to CPU/GPU code generator constitute the core of our C generation tool from SystemC. This tool called SysCellC tool has about 5000 C++ and Java code lines. It can be noticed that this design flow can easily be adapted to target a wide range of architectures like FPGA and Cell processor. The next section details the SystemC to C code generator and the MPI based SystemC channel interface library.

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

4. SystemC to C

4.1. sc_signal and sc_fifo semantic

As presented on sub-section 3.1, communications between sc_modules rely on sc_signal and sc_fifo channels. These channel are accessed by the sc_modules through the channels interfaces composed of read() and write() methods. These channels have the following semantic.

From the data structure point of view, due to its double buffering semantic, a sc_signal is implemented with (1) a two-element table (sc_signal_table) corresponding to its current and future value, (2) with an index (sc_signal_index) indicating the index of the future value, 0 or 1 and (3) a flag indicating if the future value has been written.

The sc_fifo channel is a blocking First In First Out (FIFO) queue with a circular buffer semantic. This means:

- a number of places is associated with this channel,
- if a process attempts to write a full sc_fifo it stalls until space is available,
- if a process attempts to read an empty sc_fifo it stalls until a data arrives.

The synchronization between writers and readers is done through the data flow and is bidirectional: either the writer or the reader can stall.

4.2. Remote Memory Access (RMA) data transfer principles

Communications through these two channel types involve a writer and a reader. The transferred information is stored at the reader's side, which means that (1) the read(data) method, from the reader point of view, consists in reading a local variable and thus do not introduces inter processing element communication overhead, (2) the write(data) method, from the writer point of view and according to the location of the reader, can either do internal processing element data transfers, that is intra GPU or intra CPU, or inter processing element data transfers. Intra processing element data transfers only consist in local memory data movements. Inter processing elements data transfers are done through DMA: the CPU processor communicates with GPUs through DMA transfers between the host memory and the GPU video memory of the targeted GPU, that is a remote memory access; the GPU to GPU data transfers are also done with DMA transfers. Whatever the kind of data transfer is, the implementation of the read and write methods is done with MPI-2 primitive calls which hide the details of the communication mechanisms.

4.3. sc_signal and sc_fifo MPI implementation

The one-sided Remote Memory Access (RMA) subset of MPI-2 is an efficient implementation solution of MPI [15][19]. Primitives presented on table 1 are well suited for implementing the presented SystemC programming model.

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

Table 1. RMA MPI subset

MPI_com_rank()
MPI_Init()
MPI_Finalize()
MPI_Put() & MPI_Wait()
MPI_Barrier()

As stated above, the sc_signal channel has a double buffering semantic. With respect to the RMA semantic, its read method only consists in reading its current value in a local variable.

The write method writes the next value of the sc_signal so that if the reader of the sc_signal is on the same processing element as the writer, this exchange only involves local variable transfers. Else, if the reader and writer are not on the same processing element, the transfer is achieved through DMA. All these mechanisms are hidden in our MPI_put() implementation. Another problem is to toggle the current value and the next value of the sc_signal. From the SystemC point of view, this is done at the end of the current delta-cycle and if and only if the signal value has changed. This last condition implies to implement an acknowledge mechanism which validates the permutation of the current value with the next value. To preserve the MPI-2 RMA semantic, we do not include the acknowledge in the MPI_put() primitive to implement the write method. The MPI_put() semantic corresponds to a simple data transfer. Thus the acknowledge is done with a second MPI_put() which, from the reader point of view, toggles the read index of the considered sc_signal.

As stated on sub-section 3.1 the processes are only sensible to a signal which can be considered as a clock. The permutation of a sc_signal current value with its next value only occurs at the end of a delta cycle. From the MPI-2 RMA subset point of view, we implement the SystemC wait() with the MPI_wait() primitive and the clock event is implemented with a MPI_barrier() which synchronizes all the waiting processes.

At last, a sc_signal can have multiple readers. Broadcasting is achieved with as much MPI_put() couples calls as sc_signal readers (one for the data transfer and another for the index toggling).

To conclude this sc_signal topic, the sc_signal expresses a double buffering communication mechanism which can be efficiently implemented with the MPI-2 RMA subset, whereas MPI-2 RMA does not natively support this double buffering semantic.

The sc_fifo data transfers are handled by the same MPI_put() primitive, but compared to the sc_signal, the sc_fifo synchronization mechanism differs. Synchronization is done through the data flow, not through a global clock. Each time a sc_fifo blocking read or blocking write is done the MPI_wait() primitive is called. The sc_fifo channel provides a circular buffering communication mechanism.

To end this sub-section, the sc_signal and sc_fifo communication channels have a multibuffering communication semantic, which can be efficiently implemented with the MPI-2 RMA subset. Their implementation allows the overlapping of communication and computation since while a process is working on a given data, the previous data can be transferred.

4.4. GPU MPI-2 RMA implementation

As all the GPU kernels are launched from the same binary code, during the MPI_com_rank() call, the CPU launches as many CPU threads as GPUs on the platform. Each CPU thread manages its own GPU and launches the

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

corresponding kernels on it.

The MPI_Put primitive, is implemented with the OpenCL *clEnqueueReadBuffer()* and *clCreateBuffer()* functions for GPU to GPU or GPU to CPU DMA transfers. For GPU to GPU DMA, we use two calls with an intermediate CPU memory copy. All we need to carry-out the DMA between GPUs is that the CPU code provides the memory pointer address of the destination GPU, which is done at the initialization of each GPU kernel by the CPU code. The MPI_Barrier() implementation among multiple GPUs is based on the signal register barrier algorithm presented in [12]. The CPU threads wait the completion of all the kernels involved in synchronization. The MPI_wait() and MPI_Finalize() primitives are platform independent.

4.5. Computation processes scheduling

The last point concerns the scheduling of the CPU threads and GPU kernels (the computation processes) when several sc_modules are mapped on the same CPU or GPU. Two solutions are possible: with an Operating System (OS) on the CPUs or with a dedicated scheduler, that is to say some kind of Application Specific Lightweight Scheduler (ASLS). We implemented the second solution. In our design flow, we generate an ASLS which respects the semantic of the original SystemC code. In that way it is possible to map several components on the same GPU, with the same restrictions as the MPI microtask approach [8] that is tasks with a data size and binary code size small enough to be loaded on the GPU memory. This ASLS is called from each MPI_wait() which corresponds to each SystemC wait() or at the end of a SC_METHOD, that is a SystemC thread with no wait(). The ASLS is also called from each sc_fifo blocking read() or write() instructions. This scheduler manages the synchronization of event sensitive tasks with the MPI_barrier().

4.6. C code generation

The C code generation consists:

-inlining the SystemC specification,

-replacing the sc_fifo and sc_signal read and write calls by the corresponding primitives of our MPI based SystemC channel interface library, e.g. write_signal, read_signal,

- including binding information on these calls,

- generating our ASLS.

We illustrate the C generation process on the Producer/Consumer example. The CPU C equivalent code of the SystemC Producer is shown on the right side of figure 3. The write_signal() primitive uses four parameters: F2 is the sc_signal from the top level that connects the consumer and the producer together through a2 port.

```
class producer2 :
public sc_module {
public:
  sc_out <int> a2;
  sc_in <bool> clk;
  SC_HAS_PROCESS(producer2);
  producer2(
    sc_module_name name) :
    sc_module(name) {
```

```
{
  int nb2;
  nb2=2;

  while(1) {
    if ((nb2%2)==0)
      write_signal( F2,1,0,nb2++);
    else {
      write_signal(F2,1,0,nb2);
```

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

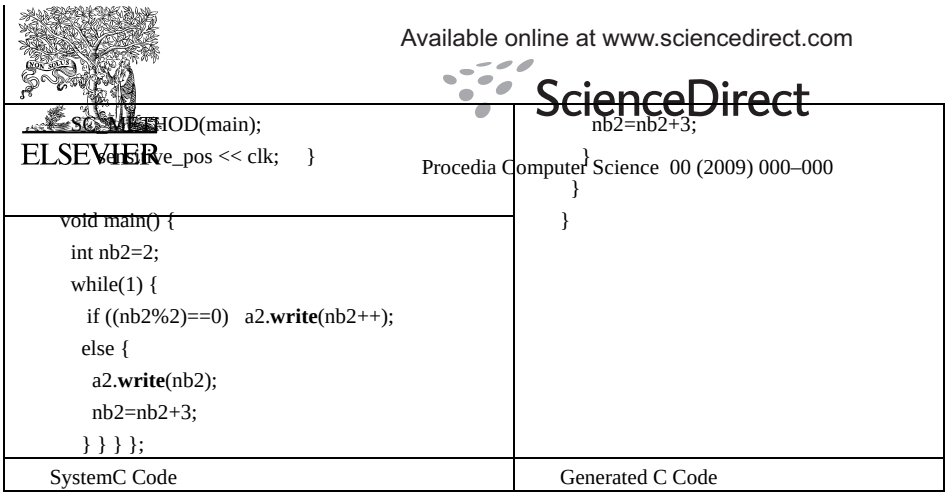

Fig. 3. From SystemC code to C code

The second parameter is the channel number. The third parameter is a priority and the last one is the expression assigned to the channel. The C code generation is fully architecture independent. Only the implementation of the MPI-2 RMA primitives is architecture dependent. This allows targeting a variety of architectures: we just need to have an MPI-2 RMA implementation for it. Nevertheless, for performance reasons, the designer can introduce target specific constructs in his source code, e.g. vectorized programming for the GPUs. However, if the designer wants to preserve the generality of his code he can define an API which can be implemented with the target specificities. This allows him to efficiently program a wide range of machines from a simple workstation, e.g. to validate the generated C code, to homogeneous or heterogeneous grid computers, with the same C code. For GPU code, we use only SystemC threads with no wait(). Thus, the thread code (corresponding to a GPU kernel) has to finish before getting back to the CPU executing the ASLS scheduler, which performs the data transfers and synchronization.

5. Case study

In order to evaluate the proposed flow on a single CPU, three examples have been experimented in this section: first a consumer/producer case with two to twelve SystemC components linked together used to validate the approach, second, a more realistic case: a CDMA radiocommunication system to validate the ASLS and then a visual attention model to validate the multi-GPU use. These three examples applications use the non blocking sc_signal channel. The consumer/producer with two producer modules that send data to the same consumer module has a system description of about 86 SystemC code lines. The CDMA system description has about 976 SystemC code lines; it includes 7 modules with 8 concurrent processes. The visual attention model is more complex with many C functions. It includes 15 processes.

The 7 modules of the CDMA system are: the top module builds the system hierarchy, one module that generates samples, three modules that compute the QPSK modulation, the THR (Hadamard transform) and the interleaving, one that implements the real environment channel behavior by introducing noise, and the last ones that do the reverse treatment that is de-interleaving, ITHR and demodulation. All the modules work in a pipelined dataflow way. Table 2 shows the code size of the different codes on two different configurations with their binary size and their average execution time per processing iteration. SystemC Linux refers to the SystemC code which is the entry point of our design flow, executed on a CPU. Its Binary code size and execution time (on a Linux workstation) result from the compilation of this code with the SystemC library. ASLS refers to the C code generated with our design flow. It has been compiled and executed on a Linux workstation also, with no GPU. The results collected in table 2 show the advantages of our ASLS.

Table 2. prod/cons and CDMA results

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

	[B1]Line number		Binary code size		Execution time	
	SystemC	ASLS C	SystemC	ASLS C	SystemC	ASLS C
Prod/Cons	86	136	592 K	13K	5.3 µs	0.15 µs
CDMA	976	490	1.8 M	66K	21 µs	2.16 µs

The execution time of the ASLS C code is nearly ten times faster than pure SystemC execution. So, it is more time efficient to validate a SystemC code with the C code generated with our SysCellC design flow rather than compiling and executing it with the whole SystemC simulation engine. Besides, the generated binary code sizes are significantly reduced with our ASLS compared to SystemC. This point is important for GPU applications with limited memory sizes. Finally, the sizes of the generated C source code and the original SystemC source code are similar. Moreover, this generated C code is entirely "readable".

The visual attention model described figure 4 mimics the human visual perception from retina to cortex [21]. This model is compute-intensive, as it uses both static and dynamic information to calculate the output saliency map. The mapping of the static and dynamic pathways is done based on computational times of the different kernels and data transfers between GPUs. We targeted here a 3-GPU platform illustrated in figure 5. After optimization we obtained a suitable cut of the dynamic path just after the recursive Gaussian filters that are followed by motion estimator. Each half of this cut takes almost 50ms, which is half of the entire dynamic pathway.

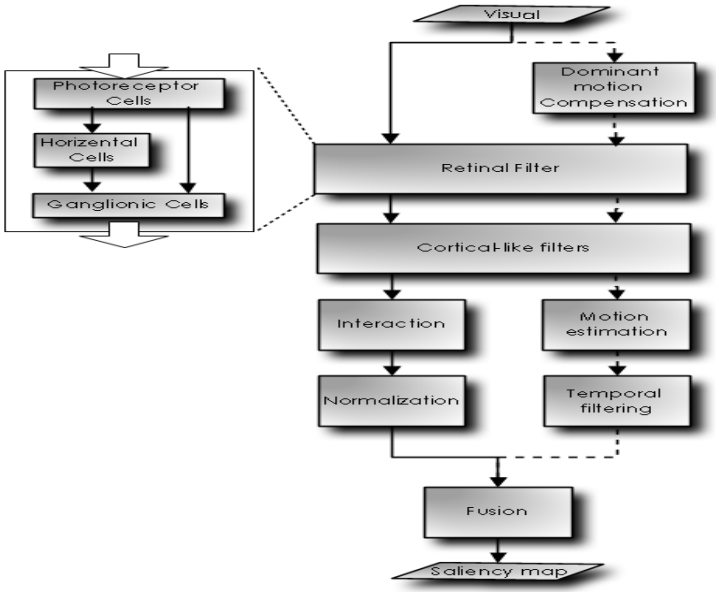

Fig. 4. The bottom-up visual saliency model

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer ence
/locate/procedia

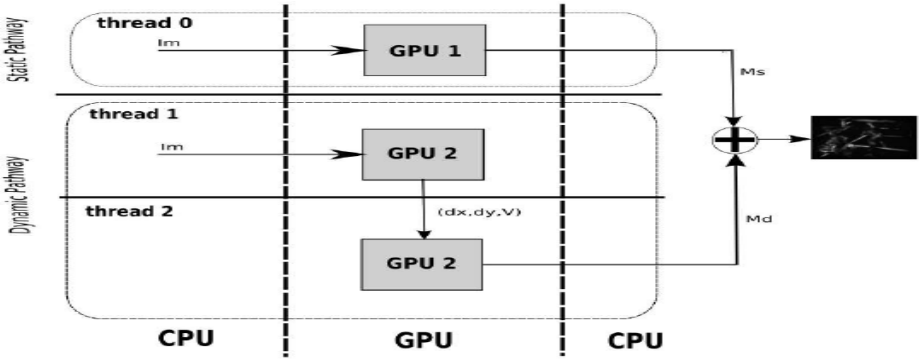

Fig. 5. Block diagram of multi-GPU pipeline

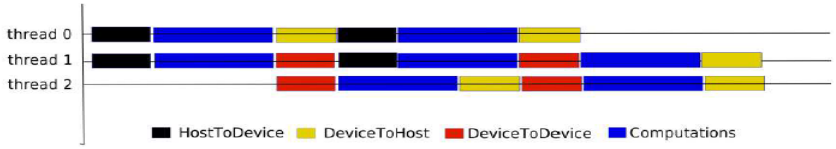

Fig. 6. Time-line of the kernels scheduling

The inter-GPU communication between thread 1 and thread 2 involves the transfer of a N-level pyramid for the input image treated with retinal filter and cortical filter, which can be overlapped by computation. The data used between other kernels is significantly greater. Afterward, thread 2 is responsible for the estimation and temporal filtering. Finally, the static and dynamic salliency maps from thread 0 and thread 2 are fused together into the final visual saliency map on the CPU. Consequently, the simplified time-line in figure 6 shows establishment of a pipeline that cuts off the time to calculate the entire visual saliency model at a rate of one image every 50ms instead of 150ms for single GPU solution, leading to a 35 speedup factor compared to the single CPU solution.

5. Conclusion

In this paper we propose a design flow which automates the implementation of a SystemC specification of an application to its multi-GPU implementation. We show that the SystemC programming model is well suited to specify applications with the streaming programming model and SystemC semantic, leading to an optimized implementation. The sc_signal and sc_fifo SystemC constructs are a convenient way to express multi-buffering communications. We show that, from the implementation point of view, these communication schemes allow to optimize the overlapping of communications and computations. Besides, the C code we generate to program the multi-GPU cluster is architecture independent. The architecture dependent code is hidden in the MPI-2 RMA primitives used for communication and synchronization. This allows to easily target any platform if we dispose of a MPI-RMA implementation for it.

Moreover we embed in our C code an application specific lightweight SystemC scheduler which is very efficient, in terms of code size and task switching time, compared to the SystemC scheduler, at the price of some language restrictions. Future work concerns the extension of the accepted SystemC subset: we schedule to support the

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2009) 000–000
Procedia Computer Science
www.elsevier.com/locate/procedia

sc_event synchronization mechanism. We also plan to integrate a data tiling methodology to help the designer with data sizing. With this compile flow we will also target heterogeneous computer grids with FPGA and Cell processors.

References

1. Peter Hofstee. Introduction to the Cell Broadband Engine. Technical report, IBM Corp., 2005.
2. Khronos Group, "Khronos Launches Heterogeneous Computing Initiative". Press release. 2008.
3. SystemC, http://www.systemc.org/
4. Black, D.C. & Donovan, J. SystemC: From the ground up, Eklectic Ally, 2005.
5. Grotker, T. System Design with SystemC Kluwer Academic Publishers, 2002.
6. Ian Buck et al. Brook for GPUs: Stream Computing on Graphics Hardware, SIGGRAPH 2004.
7. Tom R. Halfhill, PARALLEL PROCESSING WITH CUDA, MICROPROCESSOR Report, 2008.
8. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. IBM Sys. J., 45(1):85–102, 2006.
9. Stream processing, http://en.wikipedia.org/wiki/Stream_processing
10. Gropp, W.; Lusk, E. & Skjellum, A. Using MPI: portable parallel programming with the message-passing interface MIT Press, 1999.
11. IEEE Standard SystemC Language Reference Manual.
12. John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu MCUDA: An Efcient Implementation of CUDA Kernels on Multi-cores,Technical Report, University of Illinois at Urbana-Champaign, 2008
13. SPIRIT Consortium, SPIRIT V2.0 Alpha release, 2006.
14. C. Sorel and Y. Lavarenne, From Algorithm and Architecture Specifications to Automatic Generation of Distributed Real-Time Executives : a Seamless Flow of Graphs Transformations, In *Formal Methods and Models for Codesign Conference, France*, June 2003.
15. Ziavras, S.G.; Gerbessiotis, A.V. & Bafna, R. Coprocessor design to support MPI primitives in configurable multiprocessors Integr. VLSI J., Elsevier Science Publishers B. V., 2007, 40, 235-252
16. Andrews, G. Foundations of Multithreaded, Parallel, and Distributed Programming Addison-Wesley, 2000.
17. Chapman, B.; Jost, G. & Pas, R.V.D. Press, M. (ed.) Using OpenMP: Portable Shared Memory Parallel Programming Mit Press, 2007.
18. Herrera, F. & Villar, E. A framework for heterogeneous specification and design of electronic embedded systems in SystemC ACM Trans. Des. Autom. Electron. Syst., ACM, 2007, 12, 1-31.
19. Velamati, M.K.; Kumar, A.; Jayam, N.; Senthilkumar, G.; Baruah, P.K.; Sharma, R.; Kapoor, S. & Srinivasan, A. Optimization of Collective Communication in Intra-Cell MPI HiPC, 2007, 488-499
20. Dana Schaa and David Kaeli, Exploring the Multiple-GPU Design Space, IPDPS 2009.
21. S. Marat, T. Ho Phuoc, et al. Modelling spatio-temporal saliency to predict gaze direction for short videos. Int. J. Comput. Vision, 82:231–243, 2009.