Accelerating calculations on

the RASC platform. A case study of the exponential function

Abstract

This paper presents results of the tests performed to determine high speed calculations capabilities of the SGI RASC platform. Different data transfer modes and memory management approaches were examined to choose the most effective combination of the Host and RASC memory adjustments. Obtained results of measurements revealed that Direct I/O mode together with DMA transfer provides the highest data throughput between the Host and RASC slice. Nevertheless, for some application multi-buffering may appear to be more suitable in terms of concurrent data transfer capabilities and FPGA algorithm execution. As an example of calculation to be accelerated the exponent function is taken for which computation speed is far beyond data transfer capability of the RASC platform.

Introduction

It has been investigated for a long time how to employ FPGA chips in HPC systems, not only as coprocessors, but also as a highly efficient computation module. It is worth enumerating several factors preventing FPGAs from being widely implemented in HPC solutions. The first one would be the need to increase effectiveness of the floating point calculations on FPGAs since this sort of computation is the predominant part of scientific applications run on HPC machines. SGI RASC RC100 [1] has been challenged on this issue by implementing double precision floating point modules as the test bench structure, results of which are presented in this paper. Another factor is GPUs [2] (Graphic Processor Unit) rising capabilities of high speed calculations, posing a challenge for FPGAs which are still struggling with interface issues within one system. Many vendors (e.g. SGI, SRC, DRC, Xtreme-Data) try to address these issues by releasing platforms which are equipped with multi-dimensional data transmutation systems to keep up with the computational demands.

This paper presents a slightly different approach: we try to evaluate RASC capability of achieving acceleration of quantum chemistry algorithms just by writing a customized VHDL code which, however, adheres to all SGI design recommendations. The quantum chemistry application [3] (Hartree-Fock method of Schroedinger evaluation) which will be partially implemented on RASC may be regarded as a certain kind of verification environment for HPRC platforms. The exponential function implemented on FPGAs offers significant acceleration in comparison to the General Purpose Processors (GPP) implementations [4]. The exponential function is commonly used in quantum chemistry algorithms. Therefore a question arise whether a single exp() function calculation can be accelerated on FPGAs; whether system overheads i.e. data transfers cause that employing FPGAs results in calculation speed-up. Consequently different data transfer modes have been tested both on the host and RASC side to provide the highest achievable data bandwidth.

SGI provides RASC together with C-language software drivers. Nevertheless no implementation results (e.g. real data transfer throughput, latency) are provided. Therefore the main idea behind this article is to find out how FPGAs configuration time, data transfer influence the whole system speed-up. Similar considerations were publish in [5] nevertheless in this paper all data transfer and configuration modes are considered. Besides FPGAs configuration time on the host side was measured and possibility of concurrent software routines execution was tested. It should be noted that for both wavelet transform [5] and exponential function, the performance bottleneck is data transfer limit rather than FPGAs calculation speed.

SGI Altix 4700 with RASC

The Altix 4700 series is a family of multiprocessor distributed shared memory (DSM) computer systems that currently ranges from 8 to 512 CPU sockets (up to 1,024 processor cores) and can accommodate up to 6TB of globally shared memory in a single system while delivering a teraflop of performance in a small-footprint rack.

The RASC communication module is based on an application-specific integrated circuit (ASIC) called TIO which attaches to the Altix system NUMAlink interconnect directly instead of being driven from a compute node. TIO supports the Scalable System Port (SSP) port that is used to connect the Field Programmable Gate Array (FPGA) to the rest of the Altix system for the RASC program.

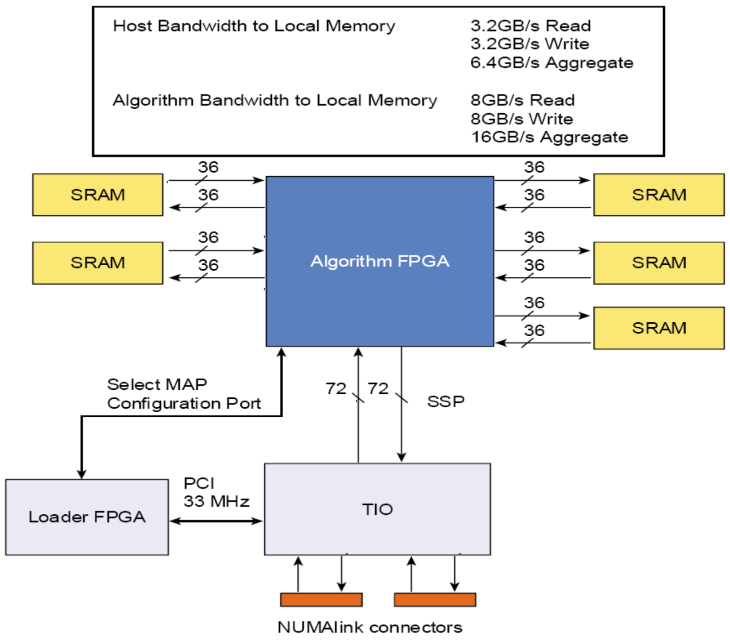

Fig. 1 Block diagram of the RASC slice (half of all resources are presented)Błąd! Nie zdefiniowano zakładki.

The RASC hardware blade contains two computational FPGAs, two TIO ASICs, and a loader FPGA for loading bitstreams into the FPGAs. The RASC blade has two Xilinx Virtx-4 LX200s on the board and memory resources with 10 synchronous static RAM modules (SSRAM DIMMs), which are grouped in three logical memory structures.

Two exp() functions have been implemented [4], each of which processes a 64-bit data chunk, so each clock cycle one 128 bit input word is fed into the implemented twin module structure and one result is obtained. SGI RASC v2.1 [1] device has four transmission/managing modes, two Host and two RASC settings. These options cover the huge spectrum of potential memory division and managing methods.

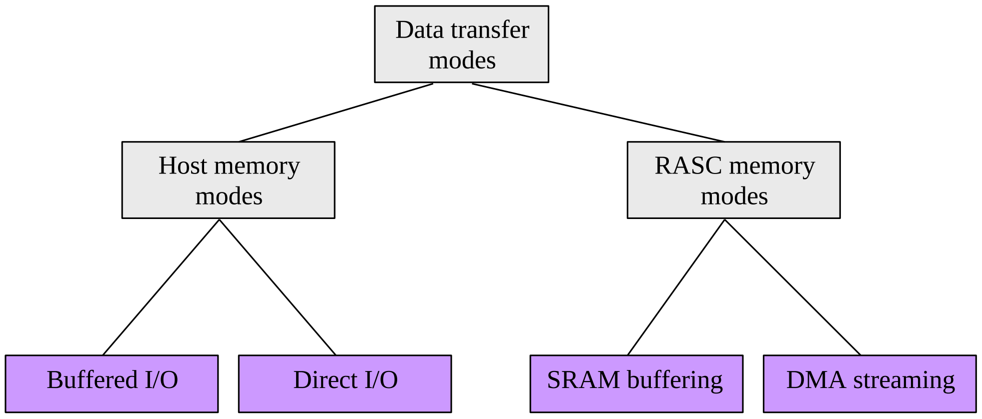

Fig. 2 Different data transfer modes available on Altix 4700 machine with the RASC slice

It is worth emphasizing that data transfer optimization is done on the Host side as well as on the RASC side. The method to apply the presented modes will be described later on in this paper.

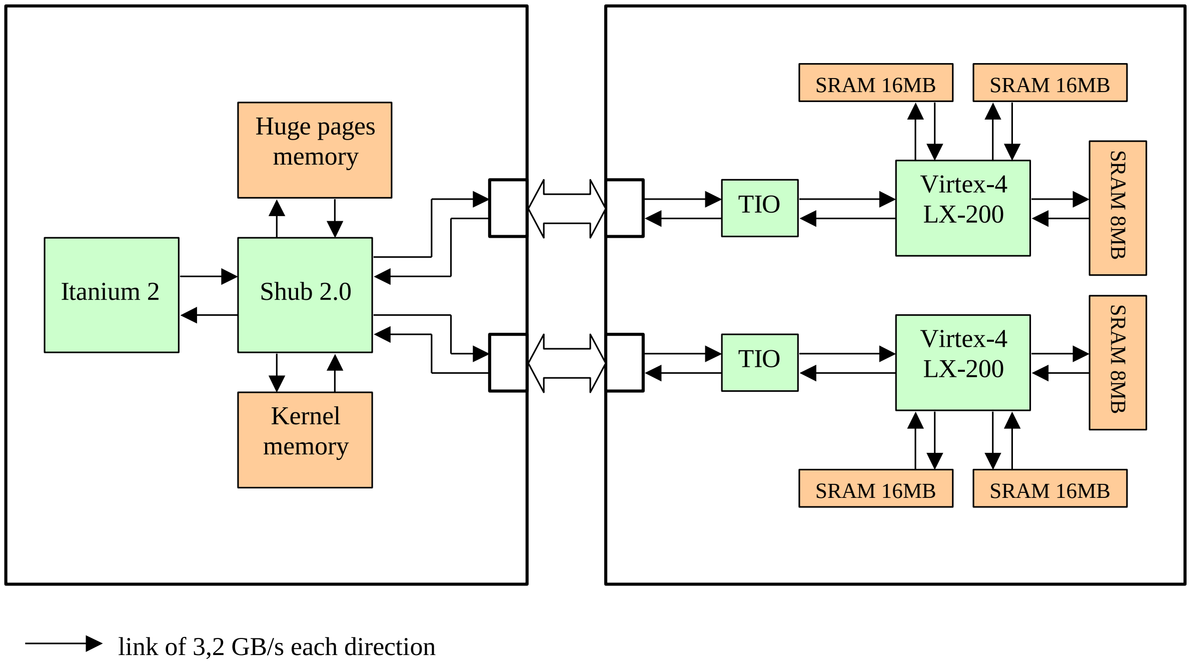

Fig. 3 Communication paths between host node and RASC slice

There are two FPGAs available on the RASC platform, but no link between them has been provided. It is possible to communicate across both FPGAs, but only by an external hub device - adding communication overhead because of delays associated with long lines and routing time.

RASC memory managing modes

SRAM buffering

This mode involves allocating data to the RASC SRAM memory. Input data must be transferred to one of the three SRAM banks, then user logic implemented in FPGA starts execution of the algorithm routine and calculations results are written back to memory. It is advisable to choose two different memory banks to make application execution more fluent.

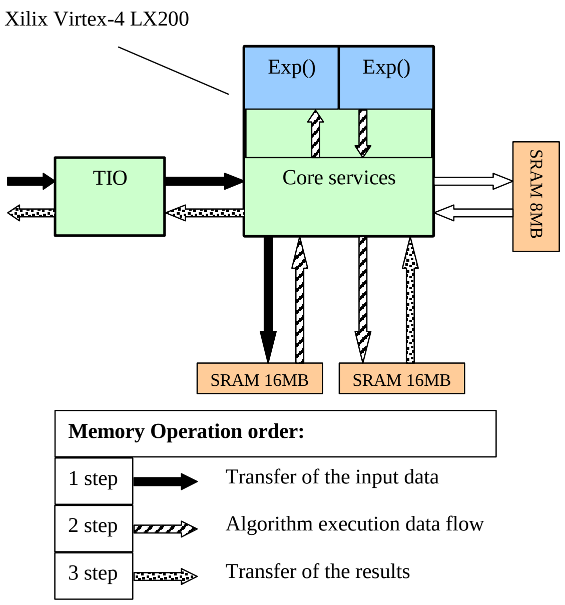

Fig. 4 Data transfer steps of the SRAM buffering mode

All the steps presented in the Figure 4 are conducted in sequence in contrast to multi-buffering mode (Fig.5). As it has been shown in Figure 4, implementation of the SRAM buffering mode leads to a three-stage data transfer operation for a single algorithm execution which in turn is time-consuming. However there is multi-buffering transfer mode which allows FPGA to concurrently transfer the data and perform calculations since *Core services* logic allows these two activities to be done simultaneously.

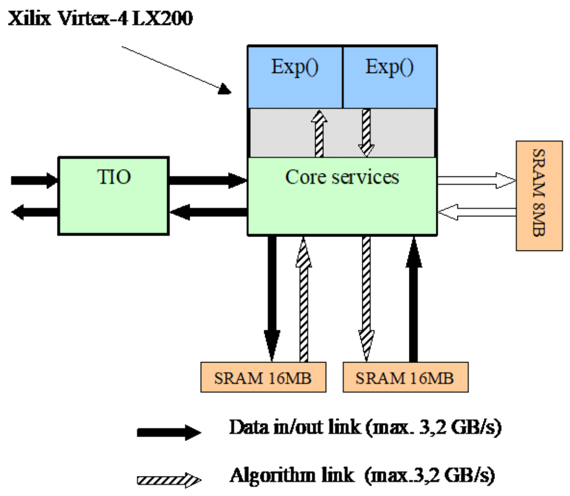

Fig. 5 Multi-buffering operation mode

Core services (Fig.4,5) maintains communication between memory, TIO ASIC and algorithm in FPGA. It is a pre-synthesized bundle of modules which must be implemented together with a user algorithm. Moreover, a dedicated wrapper within Core services is provided to facilitate design implementation, which also serves as an interface between Core Services and the algorithm.

The RASC algorithm execution procedure, from processor's perspective, is composed of several functions which reserve resources, queue commands and perform other preparation steps. It is noteworthy (Fig.6) that the time consumed by the functions remains roughly the same value, independent of the algorithm being executed. The resource reservation procedure, once conducted, allows many runnings of the algorithm – which amounts to huge time savings, since the procedure takes approximately 7.5 ms, which is roughly 99 % of overall execution time of the algorithm.

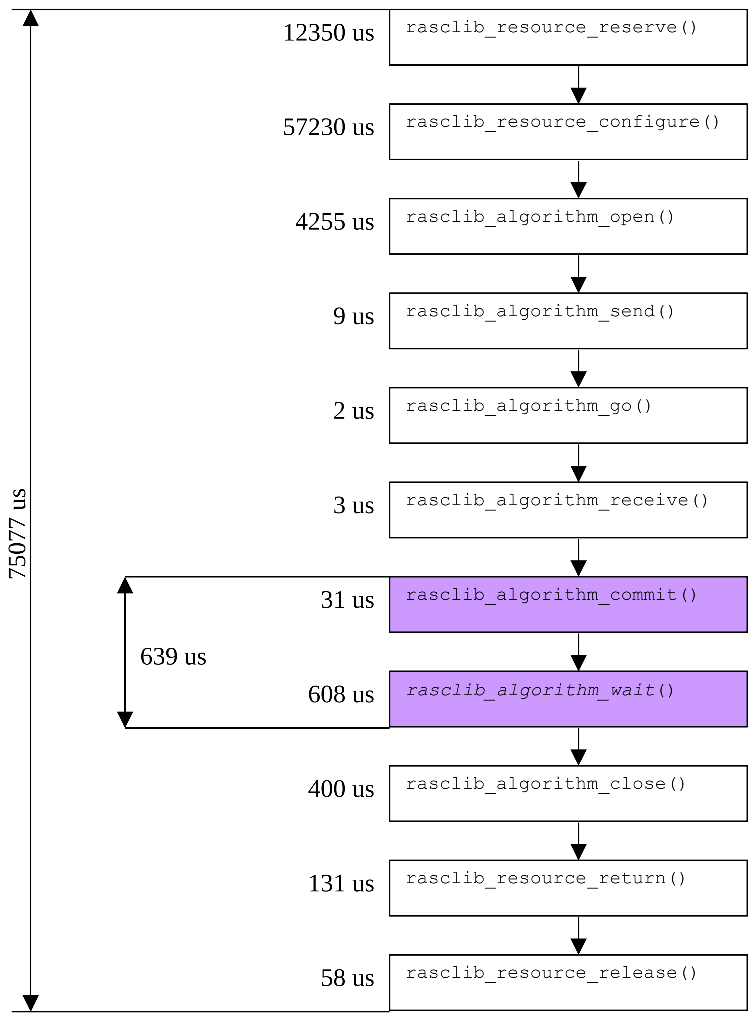
12350 us
rasclib_resource_reserve()
57230 us
rasclib_resource_configure()
4255 us
rasclib_algorithm_open()
9 us
rasclib_algorithm_send()
2 us
rasclib_algorithm_go()
3 us
rasclib_algorithm_receive()
75077 us
31 us
rasclib_algorithm_commit()
639 us
608 us
rasclib_algorithm_wait()
400 us
rasclib_algorithm_close()
131 us
rasclib_resource_return()
58 us
rasclib_resource_release()

Fig. 6 Time estimations of rasclib functions execution

Rasclib_algorithm_commit() and *rasclib_algorithm_wait*() calls are considered to be the key (Fig.5) part of the RASC software execution routine. The first one activates FPGA algorithm, the second one waits for the completion flag. The period between these two commands is the transfer and algorithm execution time. All curves (Fig.8,9) reflect overall processing time of the same amount of data, but differ in size of the single data chunk, which varies from 1024x64 bit = 8 kB to 1048576x64 bit = 8 MB. It has been observed that for the bigger chunk much better results are achieved in terms of effective execution time. However above 1 MB, a decrease of effective execution time seems to indicate saturation, therefore sending data in bigger portions may not improve the performance of the system so much. The most effective execution time of single exp() function for SRAM buffering mode is 12 ns, so 9,5 ns is transport overhead due to bus delays. Theoretical calculation time of single exp()function (data transfer is not taken into account) is 2,5 ns because two exp() are implemented on the RASC and clocked at 200 Mhz.

DMA streaming

This mode does not involve allocating SRAM memory to the RASC platform. Such an approach allows to save the time that would otherwise be consumed delivering data to onboard memory and then fetch the results from SRAM.

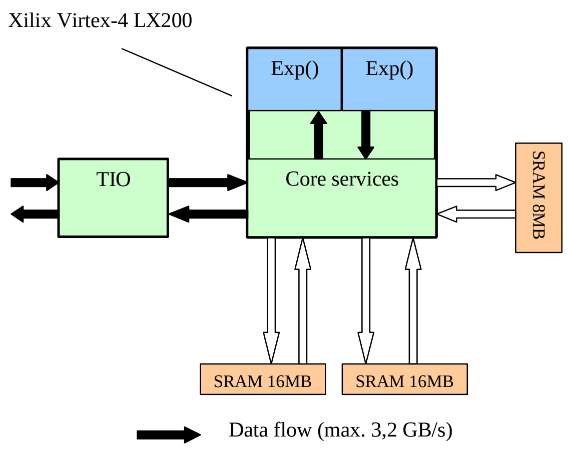

Fig. 7 DMA streaming

Data is streaming directly through the Core Services logic to the algorithm block (Fig.11. Streaming DMA reduces effective time of single exp() execution to 7,8 ns.

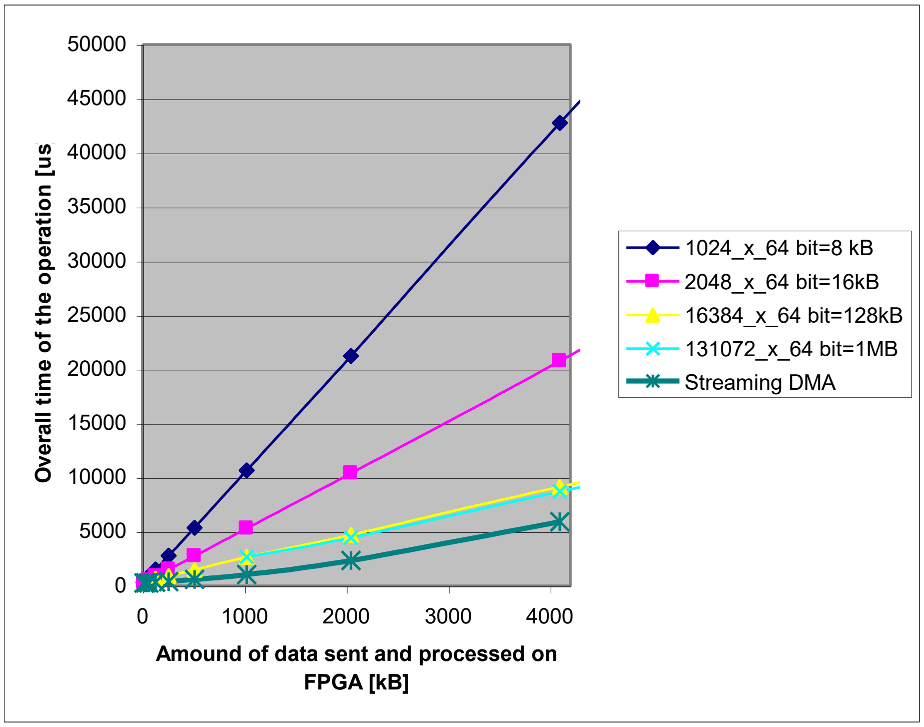

Fig. 8 Time of transferring data and executing FPGA algorithm (2 exp() functions)

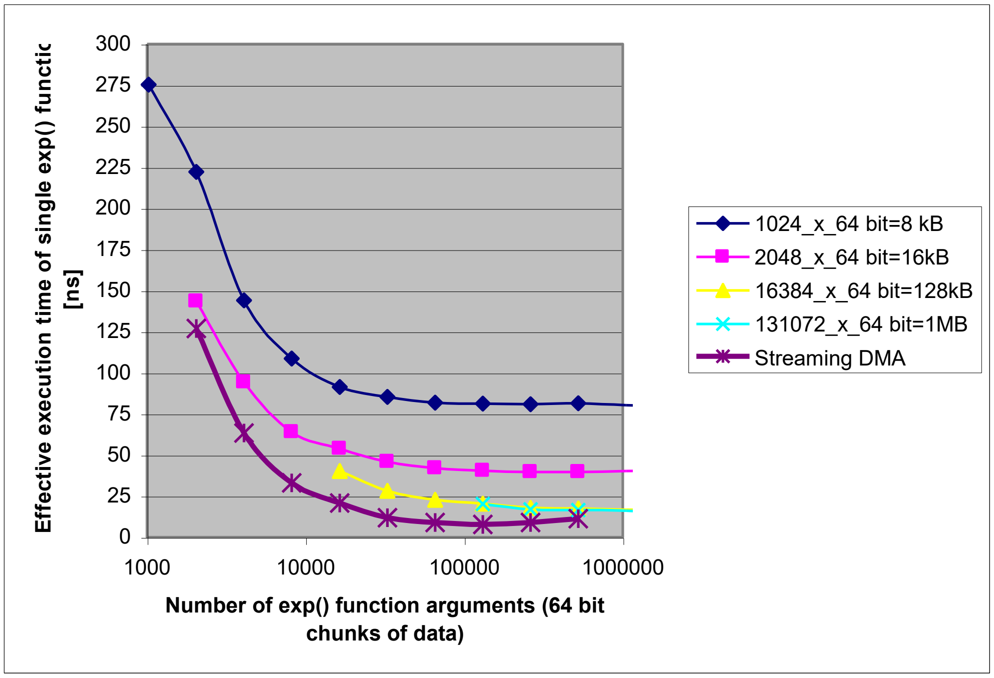

Fig. 9 Effective execution time of single exp() function

Host memory modes

Operating systems try to make the best use of the limited number of TLB (Translation Lookaside Buffer) resources. This optimization is more critical now as bigger and bigger physical memories (several GBs) are more readily available. A *hugepage* is a memory page of larger size than an ordinary page. They are usually available in multiple sizes, often up to several megabytes. Each hugepage occupies only one entry in the TLB, so the TLB coverage dramatically increases. This results in performance improvements of over 30% in many cases[6]. There are two memory modes available on the host, Buffered and Direct mode. The Direct mode takes advantage of hugepages to accelerate data transfers.

Buffered I/O mode

Buffered I/O involves allocating memory for input and output data in the user space on the host. Unfortunately transferring the data to / from the SRAM on RC100 blade using buffered I/O involves an additional copy in the kernel memory space, thus lowering the maximum throughput.

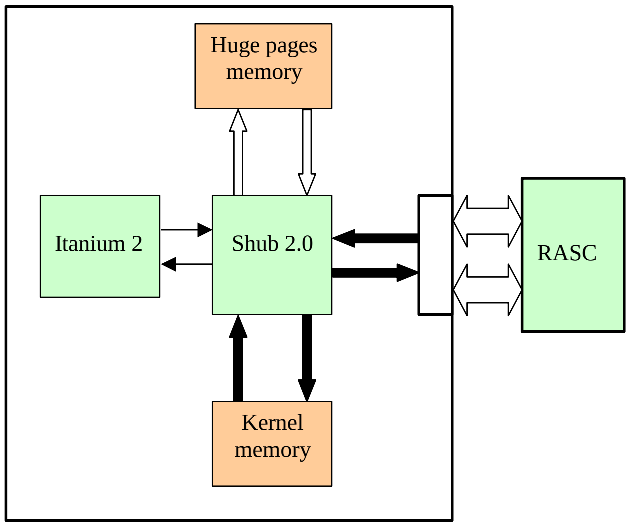

Fig. 10 Buffered I/O mode data transfer path

Kernel memory and Hugepage memory have been separated in Fig. 12 and 13 only to indicate that there are two different memory configuration options available on RASC. In fact these two memory spaces are usually combined in one physical memory.

Direct I/O mode

Data to be processed by the algorithm is located on the hugepages by issuing dedicated API functions. The data is streamed from the hugepage's memory on the host to the SRAM on the RC100 Blade, which results in significantly shorter transfer time in contrast to Buffered I/O mode.

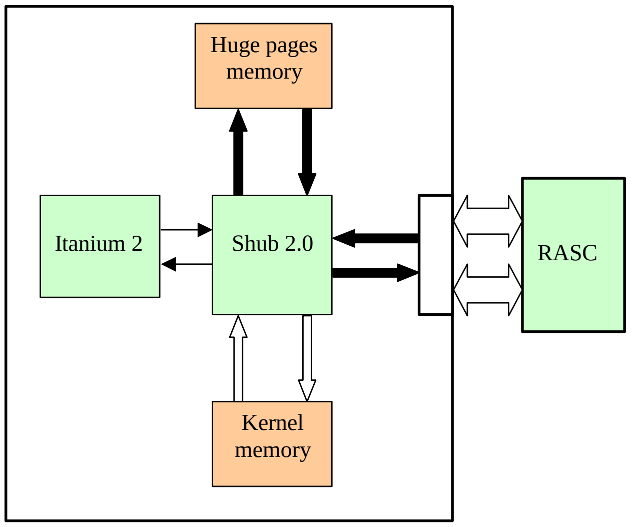

Fig. 11 Direct I/O mode data transfer flow

Host memory management plays a crucial role, therefore it is advisable to take advantage of the hugepages by choosing the Direct I/O mode to achieve the highest acceleration. Switching between Buffered I/O and Direct I/O mode is quite a simple operation and requires only a change of the flag in rasclib_open function.

Summary

SGI RASC is the modern platform providing a handy tool to design HPRC applications. According to the results of the tests that were carried out to verify SGI transfer rate estimations, 3.2 GB/s as the top bandwidth has not been reached. Several measurements that have been conducted, applying different adjustments of the transfer parameters, have shown that DMA streaming together with Direct I/O is the most effective data transfer mode for the quantum chemistry application that we aim to accelerate.

The multi-buffering mode is also considered to be effective in terms of data transfer and algorithm execution time optimization. Certain applications have large input data sets that have the same processing performed on subsets and only require that new input data be available in order to maintain continuous processing. These requirements bring up the notion of multi-buffering of a large data set through an algorithm. Multi-buffering data provides a continuous and parallel flow of data to and from the algorithm. In order for the hardware-accelerated algorithm to run efficiently on large data sets, it is recommended to overlap data loading and unloading with algorithm execution. Algorithm Block can execute one segment while the next data segment is loaded by the Read DMA Engine and the previous output data segment is unloaded by the Write DMA Engine.

[1] Silicon Graphics, Inc. Reconfigurable Application-Specific Computing User's Guide, Ver. 005, January 2007, SGI

[2] M. Giles, *GPU's - the next big advance in HPC?*, Oxford University, Reconfigurable Supercomputing Conference (MRSC), 1-3 April 2008, Belfast, Northern Ireland

[3] G.Mazur, M.Makowski, *Development and Optimization of Computational Chemistry Algorithms*, KDM'2008, March –2008, Zakopane, Poland

[4]M.Wielgosz, E. Jamro, K. Wiatr, *Highly Efficient Structure of 64-Bit Exponential Function Implemented in FPGAs*, ARC 2008, Lecture notes in Springer-Verlag, London LNCS 4943, pp. 274 – 279

[5]A. Mitra, G. Yao, W. Najjar **Performance Analysis of SGI RASC RC100 Blade on 1-D DWT, Proceedings of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07),** July 17-20, 2007 USA, Urbana

[6] Adam G. Litke, *"Turning the Page" on Hugetlb Interfaces*, IBM agl@us.ibm.com, Proceedings of the Linux Symposium, June 27th–30th, 2007 Ottawa, Ontario Canada